Deterministic Planning in Incompletely Known Domains with Local Effects

by

Vitaliy Batusov

Bachelor of Applied Science, University of Toronto, 2009

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2014

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my dissertation may be made electronically available to the public.

Deterministic Planning in Incompletely Known Domains with Local Effects

Master of Science 2014

Vitaliy Batusov

Computer Science

Ryerson University

Abstract

Conformant planning has been traditionally studied in the form of classical planning extended with a mechanism for expressing unknown facts and/or disjunctive knowledge. Despite a sizable body of research, most approaches do not attempt to move beyond essentially propositional planning. We address this shortcoming by defining conformant planning in terms of the situation calculus semantics and use recent advances in the fields of first-order knowledge base progression and query answering to develop a sound and complete conformant planning algorithm capable of handling knowledge defined in an expressive fragment of first-order logic. We implement a prototype planner and evaluate its performance on several existing domains.

Contents

List of Appendices

Chapter 1

Introduction

Loosely formulated, planning is a way for an agent to come up with a course of action that, once executed, would achieve the agent's goals. Automated planning has been one of the central problems of AI for decades. It is possible to distinguish many flavours of automated planning, each based on its own set of assumptions: the world in which the agent operates could be described using formal languages with varying levels of expressiveness and semantic clarity; the description of the world could be complete or it could contain unknown facts; the effects of the agent's actions could be deterministic or otherwise; the agent could be unique or there could be multiple agents acting simultaneously; the agent could be limited to finding simple sequences of actions, or it could be allowed to perform multiple actions in parallel; actions could be not only physical, but also sensory — the number of the possible sets of assumptions is immense. In this thesis, we focus on *conformant planning* — planning with incomplete information but no sensing actions: the agent is acting in the presence of uncertainty, but is unable to acquire information to resolve it except by altering the state of the world. Specifically, we address the problem of conformant planning in domains formulated in an expressive fragment of first-order logic. To simplify the treatment, we avoid the unnecessary complications such as multiple agents, parallel actions, non-determinism and the like.

Since planning has been a major area of AI research for many years, there exist numerous approaches to it. The approaches differ in the kind of compromise they make between abstraction and practicality and can be classified by the underlying formal framework. The frameworks, in turn, can be associated with formal languages on which they are based. On one end of the spectrum of formal languages is propositional logic, which allows for tractable reasoning, but can only deal with Boolean combinations of structureless facts. On the other end are higher-order logics which are able to quantify over objects, relations, and functions at the price of undecidability and incompleteness. Likewise, in the range of existing planning frameworks, the two extremities are STRIPS [11] and situation calculus [30]. The former, referred to as "classical planning", historically corresponds to a set of inexpressive procedural approaches with a focus on computational efficiency but with little regard for formal semantics, while the latter is based on expressive logics with an emphasis on semantically sound reasoning about actions and situations.

Common to all approaches, the description of the world in which an intelligent agent operates consists of three parts: the initial state, the desired state, and the dynamics of the world. The latter includes a set of actions and a description of how each of them affects the state of the world. The output of a planning algorithm—a plan—can take many forms. In the most basic form, a plan is a sequence of actions which achieves the agent's goals. In conformant planning, the sought action sequence needs to be able to achieve the desired state of the world regardless of the uncertainty about the initial state; that is, it needs to be a correct plan for every initial world state that is possible within the bounds of the given uncertainty. Due to this fact, conformant planning is considerably harder than classical planning.

In this thesis, we express the problem of conformant planning in its most general form using the semantics of situation calculus and use recent advances in the areas of query answering and knowledge base progression to propose an algorithm which implements a sound and complete conformant planner for domains formulated in an expressive fragment of first-order logic. In the process, we formulate and prove a set of existing and new results, which serve as the theoretical basis for our implementation. Specifically, we prove the existence, under certain constraints, of (1) finite representations for infinite query answer sets and (2) finite propositional representations for knowledge bases which are equivalent to possibly infinite sets of clauses. After formalizing conformant planning as a problem of logical entailment, we prove that it is possible, within the outlined constraints, to solve it using a simple iterative deepening search in conjunction with a generic SAT solver. We describe a prototype implementation of a conformant planner based on these results and evaluate its performance. The distinguishing feature of our algorithm is its ability to work with more expressive knowledge bases than traditionally accepted in conformant planning literature; specifically, our formalism does not enforce domain closure.

We start by outlining, in Chapter 2, the historical background behind automated planning in general and conformant planning specifically. Then, we provide the standard axiomatization of situation calculus and describe the semantics of conformant planning in that context. We review the notions of knowledge base regression (reasoning backward) and progression (reasoning forward) and focus on the latter, summarizing the latest advances in that area. We proceed by addressing query answering with respect to expressive theories. In the conclusion of Chapter 2, we arrive at a set of constraints which we are willing to apply to action theories in order to produce a conformant planning algorithm while preserving all of the semantics and much of the expressive power of situation calculus.

In Chapter 3, we describe the design and implementation of a conformant planning algorithm based on the preceding background and developments, followed by an experimental evaluation of its theoretical abilities. We confirm that the algorithm's performance is adequate for the tasks that it is addressing.

We conclude our work with a discussion of the results and future work in Chapter 4.

Chapter 2

Background

The term *conformant planning* was first introduced in [40] to denote *conditional planning without sensing*. Conditional planning is planning in non-deterministic and possibly incompletely known domains where information about the world can be acquired at runtime using sensing actions. Thus, a conditional plan is a tree whose branches are sequences of actions. A particular branch is chosen during plan execution depending on the outcome of sensing actions carried out previously. A conditional plan is successful if it achieves the goals regardless of what the actual initial state is as long as it is consistent with the given incomplete description of the initial state. Likewise, a successful conformant plan must achieve the goals in every initial state that is possible within the given uncertainty. However, without sensing actions, there can be no branching, and a conformant plan is a simple sequence of non-sensing actions.

Most of the existing practical approaches to conformant planning are based on *classical planning*, a body of research centered around a simple historical formalism for describing planning domains. For this reason, we begin with a brief overview of classical planning followed by propositional approaches to conformant planning. Then, we overview a more general account of planning based on situation calculus. We outline the essential reasoning problems associated with situation calculus and review the most prominent approaches to them. After providing the necessary preliminaries and motivation, we focus on the notions of knowledge base progression and query answering in a restricted class of action theories and survey important recent results in those areas. Among the contributions of this chapter are proofs of new results derived from existing work.

This chapter uses the following basic terminology. An *atom* is a formula that uses no logical connectives and no quantifiers. In propositional logic, an atom is a propositional symbol; in first-order logic, an atom is a formula of the form $P(t_1, \ldots, t_n)$ where P is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms. A *literal* is an atom or its negation. A *clause* is a finite disjunction of literals, commonly represented as a set of literals. A *unit clause* is a clause consisting of a single literal. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. A *ground expression* (i.e. term or formula) is an expression that contains no free variables.

2.1 Propositional Planning

2.1.1 Classical Planning

Classical planning refers to a range of approaches based on the STRIPS ("Stanford Research Institute Problem Solver") representation, which is an early and simple formalism for specifying planning domains [11, 5]. A STRIPS problem is a tuple $\langle P, I, O, G \rangle$ of sets, where

- P is a set of propositional symbols (fluents) which represent simple facts that can be true or false.
- $I \subseteq P$ is the initial state database which specifies the truth values of all propositional symbols in P. If an atom is in I, it is true in the initial state. Otherwise, it is assumed to be false. This is known as the closed-world assumption (CWA), as opposed to the open-world assumption (OWA) where the true and false values have to be specified explicitly, and the rest are unknown.
- O is a set of operators (actions). An action is a quadruple of sets of propositional symbols. The first two sets represent the action's preconditions: those required to be true and those required to be false, respectively, for the action to be executable. The remaining two sets, commonly called the *add list* and the *delete list*, represent the action's effects. The propositions in the *add list* (resp., *delete list*) become true (resp., false) as the result of the action execution.
- $G \subseteq P$ lists the goals to be made true.

It is easy to see that every subset of P, like I, describes a valid world state. Applying actions (that are possible according to their preconditions) in a world state yields another world state. A STRIPS plan is simply a sequence of actions; a STRIPS plan is successful if every proposition appearing in G becomes true as the result of executing the plan starting with the state described by I. It is natural to model a STRIPS planning problem as a search in a graph where nodes are world states and edges are actions. However, in a STRIPS instance with n fluents, there could be a total of 2^n reachable world states. Thus, the main objective is to find a procedure that can efficiently deal with such a large search space. Multiple such procedures have been investigated and implemented. This graph-based direction of research culminated in the development of Graphplan [3], an elaborate, sound and complete planner which departs from the notion of state space, but is still based on graph search algorithms.

An alternative to the graph-based approach was given in [16], where STRIPS planning is reduced to propositional theorem proving. As outlined in [6], propositional reasoning is an important task that arises in many areas of Computer Science, and, as a result, numerous efficient theorem provers, called SAT solvers, are readily available. This fact makes it possible to focus on efficient encodings of STRIPS problems as satisfiability problems and outsource the computation to the state-of-the-art in propositional theorem proving. This approach was shown to be more effective by orders of magnitude compared to contemporary graph-based approaches and has been extended to other kinds of planning, including conformant planning.

More recently, a new generation of classical planners, e.g. Fast Downward [14], have far surpassed the abilities of purely SAT-based planners.

2.1.2 Conformant Planning

In conformant planning, the state of the world at any given point is allowed to contain uncertainty. In STRIPS, this usually means, at the very least, dispensing with the closed-world assumption in the description of the initial state. Some approaches also allow ways to express *disjunctive knowledge*: e.g., the values of propositions P and Q are unknown, but $P \vee Q$ is known to hold. Other kinds of uncertainty are also possible. An incompletely specified world state is called a *belief state*; it can be thought of as a set of all deterministic states that are possible within the given uncertainty. Then the task of conformant planning can be seen as a search in the *belief state space*, the space of all belief states.

The belief state space is much larger than the already large classical state space. Thus, propositional approaches to conformant planning tend either (1) to look for compact explicit representations for possible worlds and store only a small portion of the search space at a time, or (2) to not store possible worlds at all, instead computing them on demand. In the terminology of [35], the first method models the belief state space at the *world level* and works directly with sets of possible worlds, whereas the second method operates at the *knowledge level* and works with logical formulas that represent belief states.

An early example that assumes an explicit graph-based representation of belief states is Bonet and Geffner [4]. As far as sensorless planning is concerned, the approach overall and the resulting planner GPT rely solely on search heuristics for the standard search algorithm A*. Although the explicit representation allows other techniques such as probabilistic reasoning to be used, like it is done in [15], it is ineffective compared to knowledge-level planners.

An advanced approach by Cimatti *et al.* [2, 8] uses techniques from both camps: the possible worlds are still explicit, but they are represented in a compact symbolic manner. The planning domain is an extension of the traditional STRIPS domain with a certain level of non-determinism. In particular, the authors dispense with CWA, and allow actions with conditional and uncertain effects. The planning domain is converted from a domain description language to an automaton whose states correspond to planning states and whose transitions correspond to actions. The approach is focused on implementing an effective search strategy by borrowing from the area of symbolic model checking the idea of binary decision diagrams (BDD). BDDs are an efficient representation for propositional formulas and allow easy application of Boolean operators. At the core of the approach is a choice of BDD-based data structures and optimal manipulation thereof. The search is accelerated by a simple domain-independent heuristic based on the observation that the cardinality of a belief state is inversely proportional to the amount of knowledge that it contains; thus, when planning forward, it is profitable to select the smallest (most knowledgeable) of all available belief states, and vice versa. As a result, a conformant planner HSCP is obtained, which is superior to other contemporary planners. The clever choice of data structures is reflected in the low computational and memory requirements of HSCP compared to other planners. However, despite the efficient data structures, explicit belief state representation does not scale up well, and the size of the BDD-based representation is still prohibitive in general. Another drawback is that plans produced by HSCP are not necessarily optimal (i.e., shortest). In [8], Cimatti *et al.* elaborate on [2] by further developing the means of directing the search in the belief space.

Similarly to classical planning, theorem proving has been successfully used for propositional conformant planning. One of the early examples is the planner $\mathcal{C}$-PLAN [6], which uses a SAT-solver to both generate candidate plans and evaluate their success, incrementing the candidate plan length until a successful plan is found.

Another prominent example of a SAT-based conformant planner is Conformant-FF by Hoffman and Brafman [**?**]. Belief states are represented implicitly by storing only the action sequence that leads to them. Together with the given (incomplete) initial state, this representation identifies each belief state, but makes it computationally hard to decide whether the plan is successful. An essential assumption is then made that the goal and the action preconditions be simple conjunctions of propositions. Consequently, only the intersection of the all worlds contained in each belief state is of importance, since they are the only ones guaranteed to be true in the belief state regardless of the uncertainty associated with the initial state (the actions are deterministic). Given a belief state and the corresponding intersection of its worlds, checking whether the goal (or an action precondition) is fulfilled is reduced to checking if the set of goal propositions (resp., action precondition propositions) belongs to that intersection. This is done by testing whether each proposition in question is entailed by a CNF which incorporates the initial knowledge and the semantics of the action sequence. Specifically, the CNF is built by associating with each proposition a numerical "time index". The initial knowledge is indexed with time 0 and added to the CNF. Then, each subsequent action contributes to the CNF a set of effect and frame axioms whose atoms are appropriately indexed. The construction is proved to express the semantics of the action sequence in the sense that an n-step plan is successful in reaching a goal proposition p whenever $p(n)$ is satisfied by every truth assignment which is consistent with the initial knowledge. This testing is preformed by a SAT solver. Thus, memory is traded for time, compared to approaches that use explicit belief state representations. The resulting representation is efficient and the accompanying computation is manageable; however, finding a correct plan still involves a search in the set of all action sequences, which is subject to combinatorial explosion. To circumvent this, the authors resort to using a heuristic function, which is the state transition function with the negative effects of the actions discarded. The result of this work is a competitive conformant planner Conformant-FF.

To, Son, and Pontelli in [43] point out that SAT-based approaches, including Conformant-FF, do not scale up very well when the degree of uncertainty in the initial state is large. To counter this problem, they propose an alternative belief state representation that features small size while also providing an efficient way for determining the satisfaction of a set of literals. They settle on two representations that complement each other: the *prime implicate form* and the *minimal CNF form*. A *prime implicate* of some formula φ is a clause α such that $\varphi \models \alpha$ and there is no other implicate of φ that *subsumes* α (i.e., is a subset of α). A formula φ is in prime implicate form if it is a conjunction of all its prime implicates. One benefit of this normal form is that checking the entailment of a literal by a formula takes linear time in the size of the set of all propositions. Another benefit is that each equivalence class within the set of all CNF formulas has a unique representative in prime implicate form, which is useful for keeping track of repetitive belief states. Moreover, under minor assumptions, the state transition function can be computed in polynomial time in the size of the parent state. This representation is very compact in many cases, but in other cases the exact opposite is true: the number of clauses can be exponential in

the number of propositional symbols. Formal criteria for distinguishing between the two scenarios have not been described; instead, authors rely on an empirical assessment of performance during runtime, and select an alternative representation, *minimal CNF*, if it superior in the test run against the prime implicate form. Minimal CNF is defined to be the result of a simple CNF optimization: it contains neither trivially redundant clauses nor clauses whose resolvent subsumes another clause belonging to the formula. Using this framework, To et al. implement a progression-based planner PIP which dynamically selects one of the two representations and performs heuristic search in the belief state space. In spite of the simplicity of the heuristic, PIP is shown to be competitive and, in instances with highly uncertain initial states, highly superior to all other planners.

Petrick and Bacchus [35, 36] provide a rigorous overhaul of conditional planning by reconciling it with formal logic and situation calculus. In his approach, Petrick brings forward an important distinction between the state of the world and the state of the agent's knowledge about the world. Although this premise is implicit in all knowledge-level approaches, it is especially important in Petrick's approach due to the presence of sensing actions. The approach borrows from classical planning the idea of modeling knowledge as a database and actions as updates to it. However, instead of using a single database, it uses a collection of five databases that store different kinds of knowledge. For example, the database K_f stores ground literals under the open-world assumption, the database K_w stores the planning-time effects of sensing actions, and the database K_x stores restricted disjunctive knowledge in the form of sets of literals such that exactly one of the literals in each set is true. The expressive power of such a setup is clearly superior to STRIPS, but nevertheless very restrictive in comparison to first-order logic.

In further contrast to classical planning and conformant extensions thereof, Petrick provides formal semantics for this representation by specifying a translation of the databases into formulas of first-order modal logic of knowledge — that is, first order logic extended with the modal operator K, with the intended meaning that, given a set of first-order structures $W = \{w_1, w_2, \ldots\}$ representing a belief state as a set of possible worlds, a non-modal formula ϕ is true in some world $w \in W$ iff $w \models \phi$, but $K(\phi)$ is true in w iff $w_i \models \phi$ for all $w_i \in W$.

Thus, a set of databases specifies a belief state, incorporating the possible worlds intuition in a rigorous way. An incomplete yet practically useful inference algorithm is provided for querying knowledge states, which is necessary for checking action preconditions and goals. The actions, likewise, are a generalization of STRIPS actions, but are specified by their effects on the knowledge state, as opposed to the world state. While retaining the familiar notion of database updates with *add* and *delete* lists, Petrick describes the semantics of the knowledge dynamics based on situation calculus (see Section 2.2.1). The resulting planner, PKS, uses forward chaining to find a conditional plan in an incomplete world. PKS features expressivity, including an ability to work with functions, and the high quality of the resulting plans due to its ability to abstract away from certain irrelevant considerations encountered by world-level planners.

A novel approach to propositional conformant planning was proposed by Palacios and Geffner in [33]. At the core of the approach is the observation that, while the approaches based on SAT or BDD have shown significant improvements in the compactness of belief state representations, the domain-independent heuristics they employ are inferior to the state-of-the-art in classical planning and are not

improving as much due to the challenges involved. The solution that [33] proposes is to reformulate a conformant planning problem as a classical one and use off-the-shelf classical planners to do the work.

In the approach, the planning problem is defined in a STRIPS-like manner as a quadruple $P = \langle F, I, O, G \rangle$. However, the initial state I is allowed to be a set of clauses, which are more expressive than Petrick's "one-of" clauses; also, the goals G and the action conditions and effects (O) are allowed to be sets of literals. A state of the world is defined as a propositional truth assignment over the set of propositional symbols F; any such state that satisfies I is a possible initial state. This forms the semantics behind planning: given a truth assignment s over F and the set $I' \subseteq F$ of atoms satisfied by s, the problem $P/s = \langle F, I', O, G \rangle$ is a classical planning problem. Then an action sequence is a conformant plan for P if and only if it is a classical plan for P/s for all truth assignments s that satisfy I.

Similarly to Petrick's approach, the approach of Palacios & Geffner uses the modal operator K to denote *knowing*. In the basic translation K_0, the uncertainty about a fluent L is removed by replacing it with two new fluents: KL, meaning that L is *known* to hold, and $K\neg L$, meaning that $\neg L$ is *known* to hold. The goal situation translates trivially, but only the unit clauses from the initial situation translate into the classical problem. Each conditional action $a : C \mapsto L$ translates into two: a support action $a : KC \mapsto KL$ and a cancellation action $a : \neg K\neg C \mapsto \neg K\neg L$. This basic translation is sound but incomplete. The reason behind this is the failure of K_0 to properly translate disjunctive information about the initial situation. To achieve completeness, a more elaborate translation is proposed with the help of two additional concepts: *tags* and *merges*. A tag $t \in T$ is a conjunction of literals from P whose truth value in the initial situation is unknown. A new class of literals KL/t is introduced with the following semantics: L holds if t was true in the initial situation. A merge m is a non-empty disjunction of tags, and, as such, it is applicable if one of its disjunct tags holds in the initial situation. A merge m yields a new kind of action in the translation $K_{T,M}(P)$, the action $a_m : \bigwedge_{t \in m} KL/t \mapsto KL$, the sole purpose of which is to 'merge' a set of conditional literals into a hard fact, hopefully contributing towards the goal. Here, T is the set of all possible initial situations of the conformant problem P plus the empty tag, and M contains, for each initial and goal literal in P, a single merge m on all possible situations. A particular choice of T and M results in an instance K_{S0} of $K_{T,M}$ which is both sound and complete. K_{S0} is computationally intractable, but it is possible to have a complete translation of polynomial complexity by making assumptions about the initial situation. One useful assumption is to confine the initial situation uncertainty to the prime implicate form, discussed above. The approach introduces an overhead in the form of the translation, which is generally exponential, but polynomial for most domains. The planner T_0 implementing this approach was the best conformant planner in the 2006 International Planning Competition [31].

STRIPS-based approaches to conformant planning have been steadily improving over the years. The improvements tackle such essential problems as compact problem representation and efficient reasoning. In the attempt to increase the expressiveness, the STRIPS framework has been extended from a closed-world database to incorporate incomplete and disjunctive knowledge, which necessitated the introduction of formal semantics. Nevertheless, STRIPS-based planning is inherently propositional and remains fundamentally less expressive than first-order planning based on situation calculus.

2.1.3 Beyond STRIPS

Being one of the earliest action languages, STRIPS was initially defined in [11] as a set of imprecise albeit plausible syntactical rules with little consideration for the underlying semantics. As STRIPS gained prominence, it was observed that minor, meaningful modifications to the canonical examples of STRIPS cause the standard STRIPS problem solver to malfunction. To address this issue, definitive investigations into the semantics of STRIPS were undertaken in [21] and [23].

A notable enhancement over STRIPS was proposed in [34] in the form of the language ADL ("Action Description Language"), which introduced conditional and indirect effects and explored the notions of non-deterministic and concurrent actions. To that end, ADL drops the CWA and lifts many of the syntactical restrictions inherent to STRIPS.

PDDL ("Planning Domain Definition Language"), an expressive language with clear semantics and a prominent successor to both STRIPS and ADL, was introduced in [32] with the intent to foster exchange and reuse of research and provide a unified framework for direct comparison of diverse approaches to planning. Since its introduction, PDDL has indeed become the standard language for planning research and spawned multiple versions which attend to various flavours of planning. The International Conference on Automated Planning and Scheduling uses PDDL for its recurring International Planning Competition (IPC)[31].

The prototype planner developed in this thesis does not utilize any specific planning language, relying instead on plain first-order logic represented by Prolog expressions. For evaluation, however, we borrow some of the planning domains from the conformant track of IPC2006, the last IPC to date to include a conformant track.

2.2 Planning with Situation Calculus

2.2.1 Situation Calculus and Basic Action Theories

Situation calculus is an expressive language $\mathcal{L}_{sc}$ for modeling and reasoning about dynamic worlds. It was first proposed by McCarthy & Hayes in [30] and refined by Reiter in [38]. We consider a simplified, function-free version of the Reiter's axiomatization. That is, we omit generic functions and functional fluents but retain the special-purpose function symbols listed below.

$\mathcal{L}_{sc}$ is based on first order logic with elements of second order logic. It has the standard set of logical symbols and equality. There are three sorts: *action*, *situation*, and *object*, with a countably infinite set of variables for each sort. Additionally, terms can be constructed using the following:

- A countably infinite set $\mathcal{C}$ of constant symbols of sort *object*. These are used to refer to physical objects in the domain.
- A finite set of action symbols. An action symbol uniquely identifies an action, and the arguments of an action term refer to the physical objects operated upon. Thus, the arguments can only be of sort *object*.

- The special constant symbol S_0 which denotes the initial situation. The special functional symbol *do* is used to construct complex terms of sort *situation*. For example, $do(\beta, do(\alpha, S_0))$ is interpreted as the situation that arises after performing the actions α and β in sequence, starting in the initial situation. It is common to write complex situation terms $do(\alpha_k, do(\alpha_{k-1}, \ldots, do(\alpha_1, S_0)\ldots))$ as $do([\alpha_1, \ldots, \alpha_k], S_0)$.

The special predicate symbol $\sqsubset$ is used to denote an ordering relation on situations. Given two situation terms s_1 and s_2, $s_1 \sqsubset s_2$ holds whenever s_2 can be reached from s_1 by applying a non-empty sequence of actions. For example, $do(\alpha, S_0) \sqsubset do(\beta, do(\alpha, S_0))$ should hold, but $do(\alpha, S_0) \sqsubset S_0$ should not. The axiomatization for this is provided in the set Σ described below. The symbol $\sqsubseteq$ is an abbreviation: $s_1 \sqsubseteq s_2$ stands for $s_1 \sqsubset s_2 \vee s_1 = s_2$.

The special binary predicate symbol *Poss* is used to define a relation which holds whenever an action is executable in a given situation, as described below.

Finally, for each $n \geq 0$, there are countably infinite sets of n-ary predicate symbols and $(n+1)$-ary fluent symbols. All arguments of a predicate symbol and all but one arguments of a fluent symbol must be of sort *object*; the last argument of a fluent symbol must be of sort *situation*. These symbols are used to describe the world using a set of relations over a set of terms. The predicate symbols describe static facts. The fluent symbols describe dynamic facts which change depending on the situation term.

Of all well-formed $\mathcal{L}_{sc}$-formulas, the ones that are *uniform* in a certain situation term are of special interest. Given a situation term s, an $\mathcal{L}_{sc}$-formula ϕ is *uniform* in s whenever it does not contain symbols $\{Poss, \sqsubset\}$ in its signature, does not quantify over situations, does not contain equality on situations, and every term of sort situation in ϕ is s.

The language $\mathcal{L}_{sc}$ can be used to define a basic action theory, which, like a STRIPS instance, is a formal description of a planning domain.

Definition 1. A *basic action theory* (BAT) is a collection of axioms $\mathcal{D} = \Sigma \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{ap}$, where

Σ are the four *foundational axioms* for situations:

$$(\forall \alpha, \beta, s_1, s_2) \quad do(\alpha, s_1) = do(\beta, s_2) \rightarrow \alpha = \beta \wedge s_1 = s_2 \tag{2.1}$$

$$(\forall P) \quad P(S_0) \wedge \forall a \forall s[P(s) \rightarrow P(do(a, s))] \rightarrow \forall s\, P(s) \tag{2.2}$$

$$(\forall s) \quad \neg\, s \sqsubset S_0 \tag{2.3}$$

$$(\forall s_1, s_2, \alpha) \quad s_1 \sqsubset do(\alpha, s_2) \leftrightarrow s_1 \sqsubseteq s_2 \tag{2.4}$$

The axioms (2.1, 2.2) postulate unique names for situations and describe the sort *situation* as the smallest set which contains S_0 and is closed under the application of *do*. The induction axiom (2.2) is similar to that of the Peano axiomatization of natural numbers and is the only second-order sentence in the theory. The axioms (2.3, 2.4) describe the ordering on situations and mark S_0 as the least element of this ordering.

$\mathcal{D}_{una}$ are the *unique-name axioms for actions.*

$\mathcal{D}_{S_0}$ are the *initial state axioms* (or the *initial knowledge base*), a set of first-order sentences uniform in S_0. These axioms describe the initial state of the world using predicate and fluent symbols, and include the unique name axioms for constant symbols $\mathcal{C}$.

$\mathcal{D}_{ap}$ are the *action precondition axioms*. For each n-ary action symbol A, an *action precondition axiom* (AP) is an $\mathcal{L}_{sc}$ sentence of the form

$$(\forall x_1, \ldots, x_n, s) \quad Poss(A(x_1, \ldots, x_n), s) \leftrightarrow \Pi_A(x_1, \ldots, x_n, s), \tag{2.5}$$

where $\Pi_A(x_1, \ldots, x_n, s)$ is a formula which is uniform in s and all of whose free variables are among $\{x_1, \ldots, x_n, s\}$.

$\mathcal{D}_{ss}$ are the *successor state axioms*. For each $(n+1)$-ary fluent symbol F, a *successor state axiom* (SSA) is an $\mathcal{L}_{sc}$ sentence of the form

$$(\forall x_1, \ldots, x_n, a, s) \quad F(x_1, \ldots, x_n, do(a, s)) \leftrightarrow \Phi_F(x_1, \ldots, x_n, a, s), \tag{2.6}$$

where $\Phi_F(x_1, \ldots, x_n, a, s)$ is a formula uniform in s, all of whose free variables are among $\{x_1, \ldots, x_n, a, s\}$.

Hereafter, we assume that the successor state axioms have the following more specific syntax, which incorporates the Reiter's solution [38] to the *frame problem* — the problem of specifying the behaviour of the fluents which are unaffected by the execution of an action. This syntax is a de facto standard in the existing approaches [26, 41, 10] to planning in situation calculus. Here and below, $\bar{x}$ is an abbreviation for $x_1, \ldots, x_k$ for some k inferred from context.

$$(\forall \bar{x}, a, s) \quad F(\bar{x}, do(a, s)) \leftrightarrow \gamma_F^+(\bar{x}, a, s) \vee F(\bar{x}, s) \wedge \neg\gamma_F^-(\bar{x}, a, s), \tag{2.7}$$

where $\gamma_F^+(\bar{x}, a, s)$ and $\gamma_F^-(\bar{x}, a, s)$ are first-order formulas uniform in s which express the conditions that need to be satisfied for the fluent to become, respectively, true or false in situation $do(a, s)$. Note that when neither condition is satisfied, the truth value of the fluent at $do(a, s)$ is carried over from the previous situation, s.

Example 2.2.1. Consider as an example a situation calculus axiomatization of the classical Block World domain given in [12]. The object domain consists of identical blocks which can be located either on the table or on top of one another forming vertical stacks, and the goal is to arrange the blocks in a certain desired way. The fluents that represent the properties of the blocks are:

- $clear(x, s)$: holds iff there is no other block on top of the block x in situation s.
- $on(x, y, s)$: holds iff block x is immediately on top of block y in situation s.
- $onTable(x, s)$: holds iff block x is on the table in situation s.

The actions are $move(x, y)$ meaning "move block x on top of block y" and $moveToTable(x)$ meaning "move block x to the table". The preconditions $\mathcal{D}_{ap}$ for the actions are axiomatized as follows:

$$Poss(move(x, y), s) \leftrightarrow clear(x, s) \wedge clear(y, s) \wedge x \neq y,$$
$$Poss(moveToTable(x), s) \leftrightarrow clear(x, s) \wedge \neg onTable(x, s).$$

The world dynamics is axiomatized in $\mathcal{D}_{ss}$ as follows:

$$\begin{aligned}
clear(x, do(a, s)) &\leftrightarrow \exists y \exists z(a = move(y, z) \wedge on(y, x, s)) \\
&\quad \vee \exists y(a = moveToTable(y) \wedge on(y, x, s)) \\
&\quad \vee clear(x, s) \wedge \neg \exists y(a = move(y, x)), \\
on(x, y, do(a, s)) &\leftrightarrow (a = move(x, y)) \\
&\quad \vee on(x, y, s) \wedge \neg(a = moveToTable(x) \vee \exists z(a = move(x, z))), \\
onTable(x, do(a, s)) &\leftrightarrow (a = moveToTable(x)) \\
&\quad \vee onTable(x, s) \wedge \neg \exists y(a = move(x, y)).
\end{aligned}$$

Notice that the universal quantifiers are omitted for brevity. It is evident that the SSA conform nicely to the syntactic form of Equation (2.7).

The fundamental reasoning problem in situation calculus, referred to as the *projection problem* [38], is the task of establishing whether a given action sequence executed in a given action theory results in the desired goal state. More precisely, let $\mathcal{D}$ be a BAT, let $\alpha_1, \ldots, \alpha_n$ be a sequence of ground action terms, let $s' = do([\alpha_1, \ldots, \alpha_n], S_0)$ be the situation that results from executing the action sequence beginning with the initial situation S_0, and let $\phi(s)$ be a first-order sentence uniform in s that represents the desired state. Then the projection problem amounts to determining whether the entailment $\mathcal{D} \models \phi(s')$ holds, which is a theorem proving task. Recall that $\mathcal{D}$ includes the second-order induction axiom (2.2); in second-order logic, theorem proving is well known to be ineffective, in comparison with more situationally-aware approaches called *regression* and *progression*.

Regression is a mechanism of transforming the sentence $\phi(s')$ into an equivalent formula that is uniform in S_0 and, as such, can be reasoned about with no regard for the dynamic components of the BAT, namely Σ, $\mathcal{D}_{ap}$, and $\mathcal{D}_{ss}$. This is achieved by sequentially embedding the information about the dynamics of the world relative to each ground action in $do([\alpha_1, \ldots, \alpha_n], S_0)$ into the regressed formula, starting from α_n and moving backwards. The formula that is to undergo regression needs to be *regressable*, that is, it needs to follow certain syntactic constraints, the discussion of which we omit. Assuming that $\phi(s')$ is regressable, the equivalent regressed formula $\mathcal{R}[\phi(s')]$ is uniform in S_0, and the entailment $\mathcal{D} \models \phi(s')$ holds whenever $\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}[\phi(s')]$ holds. The projection problem is thus reduced to first-order theorem proving, which is not complicated by the presence of distinct situations and thus requires no reasoning about situations.

Another approach to the projection problem is *progression*, proposed in [23]. In contrast to regression which transforms the goal formula, progression sequentially transforms the initial state axioms $\mathcal{D}_{S_0}$

according to each given ground action in the sequence $[\alpha_1, \ldots, \alpha_n]$ and the dynamics associated with it, starting with α_1 and moving forward. Specifically, progression uses the successor state axioms to determine which fluents change their values upon the execution of the next action α_i. Assertions about the new fluent values get added to $\mathcal{D}_{S_0}$, while the outdated knowledge about the same fluents is eliminated from it via *forgetting*, described in detail in Section 2.3.1. Once the BAT has been progressed through the entire action sequence, it is possible, again, to reason about its relationship with $\phi(s')$ in terms of first-order satisfiability uncomplicated by situations, since the new S_0 of the BAT is the same as s' of the goal sentence.

In this thesis, we focus solely on progression-based planning. Progression is superior to regression in that, once computed for a given situation $do([\alpha_1, \ldots, \alpha_n], S_0)$, it performs the projection-to-satisfiability reduction for all possible goal state formulas, whereas the regression needs to be computed for each such formula individually; thus, progression is more suitable for continuous planning. Moreover, the length of the regressed formula may grow exponentially in the length of the action sequence, making it unsuitable for continuous iterative planning. However, progression has not been studied as thoroughly, it is not as easily computed, and the progressed BAT may also grow at an impractical rate unless the BAT is properly constrained [26].

2.2.2 Semantics of Open World Planning

The semantics behind situation calculus planning is based on the following definition, given in [19].

Definition 2. A ground situation term $do([\alpha_1, \ldots, \alpha_n], S_0)$ is a *plan* for a BAT $\mathcal{D}$ and a goal $Goal(s)$ iff

$$\mathcal{D} \models Poss(\alpha_1, S_0) \wedge Poss(\alpha_2, do(\alpha_1, S_0)) \wedge \ldots \wedge Poss(\alpha_n, do([\alpha_1, \ldots, \alpha_{n-1}], S_0)) \\ \wedge Goal(do([\alpha_1, \ldots, \alpha_n], S_0)).$$

A planning algorithm is *sound* if every plan that it generates satisfies the equation above, and *complete* if it generates every such plan that exists.

This definition is very general and can be used as a semantic foundation for classical planning, given a translation from STRIPS to situation calculus. However, since the initial state in a BAT is described by a set of first-order sentences, this definition of planning allows for all kinds of first-order uncertainty in the initial state. This leads to unprecedented expressivity and computational challenges.

Since the right-hand side of the entailment in Definition 2 is a conjunction, the entailment of the entire expression from $\mathcal{D}$ can be established by considering the entailment of each separate conjunct. It is easy to see that each such sub-problem constitutes a projection problem with respect to a different

situation term:

$$\mathcal{D} \models Poss(\alpha_1, S_0) \tag{2.8}$$
$$\mathcal{D} \models Poss(\alpha_2, do(\alpha_1, S_0)) \tag{2.9}$$
$$\ldots$$
$$\mathcal{D} \models Poss(\alpha_n, do([\alpha_1, \ldots, \alpha_{n-1}], S_0)) \tag{2.10}$$
$$\mathcal{D} \models Goal(do([\alpha_1, \ldots, \alpha_n], S_0)). \tag{2.11}$$

Since a plan in situation calculus is a sequence of ground actions, the set of all plans is the infinite set of all such sequences. Obviously, naively generating this set is both impossible and unnecessary, since executable plans should amount to a small fraction of this set. The notion of plan executability is captured by the subformula

$$Poss(\alpha_1, S_0) \wedge \ldots \wedge Poss(\alpha_n, do([\alpha_1, \ldots, \alpha_{n-1}], S_0))$$

of the right-hand side expression from Definition 2, or, equivalently, by the set of entailments (2.8–2.10). A plan is executable if its every constituent ground action is possible to execute with respect to the previous situation, as axiomatized by $\mathcal{D}_{ap}$. To generate the set of all possible actions with respect to a given situation, it is necessary to evaluate the right-hand side of the corresponding action precondition axiom. Formally, the set of all ground actions that are possible in situation s is

$$\{A(\bar{c}) \mid A \text{ is an action name}, \bar{c} \text{ are constants, and } \mathcal{D} \models \Pi_A(\bar{c}, s)\}.$$

Observe that the last set inclusion condition is a yet another instance of the projection problem. Therefore, it can be approached via progression as discussed above. However, there remains the problem of finding all groundings of the free object variables of Π_A. This problem is known as *query answering*, and in most cases it is computationally expensive. A special class of queries — conjunctive queries — has been identified and extensively studied as a tradeoff between expressiveness and computational complexity. These notions are formalized in the seminal work [7] as follows. We denote the object assignment to variables $x_1, \ldots, x_n$ as $\langle a_1, \cdots, a_n \rangle$.

Definition 3. A *query* is a first-order formula $\phi(x_1, \cdots, x_n)$ which contains no free variables other than $x_1, \ldots, x_n$. The free variables are called the *distinguished variables* or *answer variables*. A *boolean query* is a first-order sentence. A *conjunctive query* is a query of the form $\exists \bar{y}\, conj(\bar{x}, \bar{y})$, where $conj(\bar{x}, \bar{y})$ is a conjunction of atoms and equalities over the variables $\bar{x}, \bar{y}$, and possibly constants. An *answer* to the query $\phi(x_1, \cdots, x_n)$ *with respect to an interpretation* $\mathcal{M}$ is the set of tuples

$$\{(a_1, \cdots, a_n) \mid \mathcal{M}, \langle a_1, \cdots, a_n \rangle \models \phi(x_1, \cdots, x_n)\}.$$

The answer to a boolean query is either an empty tuple, read as *true*, or the empty set, read as $false$.

Thus, an answer to a query is defined as a set of domain element tuples which correspond to object

assignments that satisfy the query in a given interpretation. We are, however, interested in the answers with respect to a knowledge base, not an interpretation. To this end, we borrow the notion of *certain answers* from [42].

Definition 4. A *certain answer* to a query $\phi(\bar{x})$ *with respect to a knowledge base* $\mathcal{K}$ is the set of constant tuples $\{\bar{c} \mid \mathcal{K} \models \phi(\bar{c})\}$.

The certain answers have the following semantics: the constant tuple $\bar{c}$ belongs to the answer to the query $\phi(\bar{x})$ wrt a KB $\mathcal{K}$ whenever $\mathcal{K} \models \exists\bar{x}(\bar{x} = \bar{c} \wedge \phi(\bar{x}))$. For conjunctive queries, $\bar{c}$ belongs to the answer to $\exists\bar{y}\, conj(\bar{x}, \bar{y})$ wrt $\mathcal{K}$ whenever $\mathcal{K} \models \exists\bar{x}\exists\bar{y}(\bar{x} = \bar{c} \wedge conj(\bar{x}, \bar{y}))$.

As remarked in [42], Definition 4 implies that query answering wrt a KB can be reduced to testing whether the KB entails every boolean query that is obtained by substituting every conceivable constant tuple in place of the distinguished variables. We expand on this idea in Section 2.4.

2.2.3 Regression-based Planning

Finzi *et al.* [12] develop an open-world planner `wspdf` which uses regression as the underlying mechanism. Their choice of regression over progression is partly due to the fact that, at the time, there did not exist a provably correct algorithm for progressing an incomplete initial database. This approach, although based on situation calculus, addresses exactly the same problem as the propositional approaches described in Section 2.1.2: planning without sensing in an incompletely known world.

`Wspdf` stands for the "world's simplest planner, depth-first". It is indeed a very simple planner which takes as the input a BAT, a plan length bound, a planning goal, and an axiomatization of a domain-specific predicate *badSituation* which is used to heuristically avoid plans known in advance to be bad.

Despite the expressiveness of the situation calculus, `wspdf` is not very far removed from the STRIPS-based approaches. It implements a *domain closure assumption* (DCA) over a finite set of constants — that is, the object domain is finite and usually small. This restriction is strong enough to allow *typed quantifiers* $\forall x : \tau$ and $\exists x : \tau$ in the initial and goal formulas; a *type* $\tau(x)$ is an abbreviation for $(x = c_1 \vee x = c_2 \vee \ldots \vee x = c_k)$, where $\mathcal{C} = \{c_1, \ldots, c_k\}$ is the set of all constants in the language. Thus the quantified formula $Qx : \tau\, \phi(x)$ is merely an abbreviation for a conjunction or a disjunction of formulas $\phi(c_i)$ over the set of all constants $c_i \in \mathcal{C}$.

`Wspdf` implements a regression-based theorem prover. The regression is performed as described in Section 2.2.1, deepening until the goal sentence is uniform in S_0. There is a minor optimization: the regression is performed depth-first on the components of the goal sentence with the hope that a component simplification renders the rest of the sentence irrelevant. The result of regression, a propositional formula, is converted into CNF. The algorithm returns success iff the initial knowledge base entails every conjunct of this CNF. The entailment can be tested using two methods. The first is based on prime implicates, the same approach as that of [43] mentioned earlier, with the same overhead of precomputing prime implicates for faster reasoning thereafter and the same drawback of a possibly very large number of resulting clauses. The second method for testing entailment uses an off-the-shelf SAT-solver, which

eliminates the need for precomputation but makes the reasoning harder. Overall, despite the first-order situation calculus framework, `wspdf` eliminates much of the available expressiveness by using DCA, essentially reducing it to that of propositional logic.

In a notable contrast to all propositional conformant planners which rely on domain-independent heuristic functions, `wspdf` requires additional control knowledge to be supplied. If that knowledge is of high quality, `wspdf`, despite its simplicity, is able to vastly outperform other planners.

2.3 Progression

Progression for situation calculus was introduced by Lin & Reiter in [23], but it was shown to require second order logic to completely characterize a progressed knowledge base. In [26], Liu & Lakemeyer proved that, for a specific class of action theories, progression is always first-order definable and computable. These results have been exploited in a number of approaches and are of fundamental importance to this thesis. We begin with the general definition of progression due to Lin & Reiter and then move on to the details of its practical variant, progression of *local-effect* action theories.

Definition 5 (Progression)**.** Let S_α denote the situation term $do(\alpha, S_0)$. Let $\mathcal{D}$ be a BAT, α a ground action, and $\mathcal{D}_{S_\alpha}$ a set of sentences uniform in S_α.

$\mathcal{D}_{S_\alpha}$ is a *progression* of the initial knowledge base $\mathcal{D}_{S_0}$ with respect to α if, for every interpretation $\mathcal{M}$, $\mathcal{M} \models \mathcal{D}_{S_\alpha}$ iff there exists a model $\mathcal{M}'$ of $\mathcal{D}$ such that:

1. $\mathcal{M}$ and $\mathcal{M}'$ have identical domains for sorts *action* and *object*,
2. $\mathcal{M}$ and $\mathcal{M}'$ interpret all situation-independent predicate and function symbols identically, and
3. For every fluent F and every variable assignment σ, we have $\mathcal{M}, \sigma \models F(\bar{x}, S_\alpha)$ iff $\mathcal{M}', \sigma \models F(\bar{x}, S_\alpha)$.

The key idea of Definition 5 is that the original BAT is a complex theory that mentions distinct situational terms, axiomatizes their relationships, and describes the dynamics of the world, whereas the progression is a situationally simple theory which, nevertheless, entails exactly the same information about the world at a particular situation as the original BAT.

Notice that the word "progression" can refer to both a knowledge base $\mathcal{D}_{S_\alpha}$ (as in the definition above) and the procedure that was used to construct it (as in the discussion in Section 2.2.1).

2.3.1 Local-Effect Progression

In [26], Liu & Lakemeyer showed that limiting the scope of the actions' effects results in significant benefits: the progressed knowledge base no longer needs second-order logic to be defined. The actions are classified as having either *local effects* or *global effects*. Intuitively, a local-effect action is one which affects the properties of only those objects that it explicitly mentions. In Example 2.2.1, the action $move(a, b)$ would affect the status of the block (call it c) on which a resided prior to the action, if there was one. Specifically, the values of the fluent atoms $clear(c, s)$ and $on(a, c, s)$ would change despite the

fact that the action did not mention block c at all. This is an example of a *global-effect* action. A different, local-effect axiomatization is given below.

For the next definition, recall that $\gamma_F^+(\bar{x}, a, s)$ and $\gamma_F^-(\bar{x}, a, s)$ are sub-formulas of a successor state axiom as per Equation (2.7).

Definition 6. An SSA is *local-effect* if both $\gamma_F^+(\bar{x}, a, s)$ and $\gamma_F^-(\bar{x}, a, s)$ are disjunctions of formulas of the form $\exists \bar{z}[a = A(\bar{w}) \wedge \phi_{cc}(\bar{w}, s)]$, where A is an action function symbol, $\bar{w}$ contains $\bar{x}$, $\bar{z}$ are the remaining variables of $\bar{w}$ (if any), and ϕ_{cc} is a formula describing a context condition. A BAT $\mathcal{D}$ is local-effect if every SSA in $\mathcal{D}_{ss}$ is local-effect.

In Example 2.2.1, only the SSA for the fluent $onTable(x, s)$ conforms to this definition. In the SSA for $clear(x, s)$, for instance, $\gamma_F^+(\bar{x}, a, s)$ consists of two disjuncts: $\exists y \exists z(a = move(y, z) \wedge on(y, x))$ and $\exists y(a = moveToTable(y) \wedge on(y, x, s))$, neither one of which is local-effect since neither action mentions x, the fluent's object argument.

Example 2.3.1. Consider an alternate axiomatization of the Blocks World SSA from [26]. It redefines the moving action as a local-effect one by making it ternary: $move(x, y)$ is replaced by $move(x, z, y)$, meaning "move block x from the top of block z to the top of block y". The notion of the table is not used in this version.

$$clear(x, do(a, s)) \leftrightarrow \exists y \exists z(a = move(y, x, z)) \vee clear(x, s) \wedge \neg \exists y \exists z(a = move(y, z, x)),$$
$$on(x, y, do(a, s)) \leftrightarrow \exists z(a = move(x, z, y)) \vee on(x, y, s) \wedge \neg \exists z(a = move(x, y, z)).$$

The restricted syntax of local-effect SSA can be beneficially exploited with the help of the unique name axioms for actions and a logically equivalent transformation. To that end, we introduce the notion of a *transformed SSA*, similarly to [41], and related concepts used in the definition of local-effect progression.

Definition 7. Let $\mathcal{D}$ be a local-effect BAT and α a ground action.

The *transformed SSA* for fluent F wrt α is the result of substituting α into the SSA for F and simplifying the right-hand side using axioms from $\mathcal{D}_{una}$ and replacing every instance of the subformula $\exists z(z = t \wedge \varphi(z))$ with the expression $\varphi(t)$, where φ is arbitrary.

The *argument set* of fluent F with respect to α is the set

$$\Delta_F = \{\bar{t} \mid (\bar{x} = \bar{t}) \text{ appears in the transformed SSA for } F\}.$$

The *characteristic set* of α is the set of atoms

$$\Omega(s) = \{F(\bar{t}, s) \mid F \text{ is a fluent and } \bar{t} \in \Delta_F\}.$$

We use $\mathcal{D}_{ss}[\Omega]$ to denote the instantiation of $\mathcal{D}_{ss}$ with respect to the characteristic set of α at situation $do(\alpha, S_0)$; i.e. the set of all formulas that arise as a result of substituting each member of each fluent's argument set into the corresponding transformed SSA.

Example 2.3.2. The transformed SSA with respect to the ground action $move(c_1, c_2, c_3)$ for the fluent $clear(x, s)$ from Example 2.3.1 is

$$clear(x, do(move(c_1, c_2, c_3), s)) \leftrightarrow (x = c_2) \vee clear(x, s) \wedge \neg(x = c_3).$$

The argument set for $clear(x, s)$ is then $\{c_2, c_3\}$, and its contribution to the characteristic set is $\{clear(c_2, s),\ clear(c_3, s)\}$. Intuitively, the characteristic set contains the atoms subject to change as a result of performing the action in question. If c_3 is "clear" at s, it would cease being "clear" at $do(move(c_1, c_2, c_3), s)$, and the opposite would happen to c_2. The "clear" status of c_1, however, cannot be altered by this action, and so $clear(c_1, s)$ is not in the characteristic set.

To avoid introducing notions used in [26] that are non-essential to this thesis, we present local-effect progression as per [10]. The latter work bases its definition on the notion of *forgetting*, due to Lin & Reiter [22], which we present in the following definitions.

Definition 8. Let $p = P(\bar{t})$ be a ground atom and let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two interpretations. We define $\mathcal{M}_1 \sim_p \mathcal{M}_2$ to hold whenever $\mathcal{M}_1$ and $\mathcal{M}_2$ agree on everything except possibly on the interpretation of p. Specifically, $\mathcal{M}_1$ and $\mathcal{M}_2$ have the same domain and interpret every constant the same. For every predicate symbol Q distinct from P, $(Q)^{\mathcal{M}_1} = (Q)^{\mathcal{M}_2}$. Finally, for every tuple of domain elements $\bar{d}$ that is distinct from $(\bar{t})^{\mathcal{M}_1}$, $\bar{d} \in (P)^{\mathcal{M}_1}$ iff $\bar{d} \in (P)^{\mathcal{M}_2}$.

Definition 9 (Forgetting)**.** Let T be a first-order theory and p a ground atom. A theory T' is a result of *forgetting p in T*, denoted $\texttt{forget}(T, p)$, if, for every interpretation $\mathcal{M}'$, $\mathcal{M}' \models T'$ iff there is a model $\mathcal{M}$ of T such that $\mathcal{M} \sim_p \mathcal{M}'$.

Since, by Proposition 6 from [22], the order of forgetting a sequence of ground atoms is irrelevant to the end result of forgetting, we will reuse the same notation for the result of forgetting a set S of n ground atoms from a theory T as follows: let $\texttt{forget}(T, S)$ denote $\texttt{forget}(\texttt{forget}(\ldots \texttt{forget}(T, p_1) \ldots, p_{n-1}), p_n)$ for an arbitrary enumeration of ground atoms $p_i \in S$.

We are now ready to present the result from [10] concerning the progression of local-effect action theories. The notation $\varphi(\mu/\mu')$ stands for the result of replacing every occurrence of μ in φ with μ'.

Theorem 1. *Let $\mathcal{D}$ be a local-effect BAT and α a ground action. The progression of $\mathcal{D}_{S_0}$ with respect to α, denoted $\texttt{prog}(\mathcal{D}_{S_0}, \alpha)$, is*

$$\texttt{forget}(\mathcal{D}_{S_0} \cup \mathcal{D}_{ss}[\Omega], \Omega(S_0))(S_0/S_\alpha).$$

To illustrate how progression can be computed, we need a more constructive approach to forgetting. The following propositional result, appearing in the form of a definition in [18], illustrates the intuition behind the theorem to follow.

Definition 10. Let T be a propositional formula and p a propositional atom. Then

$$\texttt{forget}(T, p) = T(p/true) \vee T(p/false).$$

In other words, the result of forgetting an atom p from a sentence T is a weaker sentence T' which entails the same set of sentences that do not depend on p. The next theorem, from [22], provides a syntactic result about forgetting a ground atom from a first-order sentence. This is without loss of generality since every first-order theory can be expressed as an equivalent singleton theory by replacing its constituent sentences with their conjunction. The theorem uses the following notation. Given a formula φ and a ground atom $P(\bar{t})$, let $\varphi[P(\bar{t})]$ denote the result of replacing every occurrence of the form $P(\bar{t}')$ in φ by $[\bar{t} = \bar{t}' \wedge P(\bar{t})] \vee [\bar{t} \neq \bar{t}' \wedge P(\bar{t}')]$ (the result is clearly logically equivalent to φ). Let $\varphi^+_{P(\bar{t})}$ denote the result of replacing $P(\bar{t})$ by *true* in $\varphi[P(\bar{t})]$ and let $\varphi^-_{P(\bar{t})}$ denote the result of replacing $P(\bar{t})$ by $false$ in $\varphi[P(\bar{t})]$.

Theorem 2. *Let $T = \{\varphi\}$ be a first-order theory and p a ground atom. Then*

$$\mathit{forget}(T, p) \equiv \varphi^+_p \vee \varphi^-_p.$$

Example 2.3.3. Let $\mathcal{D}$ be a local-effect BAT implementing Blocks World from Example 2.3.1. Let us compute the progression of the initial knowledge base with respect to the ground action $\alpha = move(c_1, c_2, c_3)$ assuming that there are no other fluents. Let S_α denote $do(move(c_1, c_2, c_3), S_0)$. From the previous example we know that $\{clear(c_2, s), clear(c_3, s)\} \subseteq \Omega(s)$. Working out a transformed SSA for $on(x, y, s)$ yields $\Omega(s) = \{clear(c_2, s), clear(c_3, s), on(c_1, c_3, s), on(c_1, c_2, s)\}$. Using these atoms to instantiate the transformed SSA, we obtain the set $\mathcal{D}_{ss}[\Omega]$ of formulas like

$$on(c_1, c_3, S_\alpha) \leftrightarrow c_1 = c_1 \wedge c_3 = c_3 \vee on(c_1, c_3, S_0) \wedge \neg(c_2 = c_3 \wedge c_3 = c_2),$$

which are trivially simplified to produce

$$\begin{aligned}\mathcal{D}_{ss}[\Omega] = \{&clear(c_2, S_\alpha) \leftrightarrow true,\ clear(c_3, S_\alpha) \leftrightarrow false,\\ &on(c_1, c_3, S_\alpha) \leftrightarrow true,\ on(c_1, c_2, S_\alpha) \leftrightarrow false\}.\end{aligned}$$

It remains to use the equation from Theorem 2 to forget, in an arbitrary order, the atoms $\{clear(c_2, S_0), clear(c_3, S_0), on(c_1, c_3, S_0), on(c_1, c_2, S_0)\}$ from the conjunction of all formulas from $\mathcal{D}_{S_0}$ and $\mathcal{D}_{ss}[\Omega]$, and replace the situation term S_0 with S_α everywhere in the result. A progression of $\mathcal{D}_{S_0}$ is thus obtained. The notion of irrelevance, introduced in Section 2.4, can be used to optimize the forgetting step by skipping the computation for the atoms known to be irrelevant to $\mathcal{D}_{S_0} \cup \mathcal{D}_{ss}[\Omega]$.

Note that the formulas in $\mathcal{D}_{ss}[\Omega]$ simplify to such a trivial form only when the corresponding SSA are context-free, as per Definition 6.

2.4 Uncertainty in the Initial KB

As outlined in Section 2.2.2, planning in situation calculus requires an ability to solve the projection problem, which arises when checking the action preconditions and the goal condition. In Section 2.3, we demonstrated that it is possible to avoid reasoning about situationally complex theories by computing the

progression and working with a situationally simple representation of the relevant knowledge. However, establishing the entailment in first-order logic is still a major challenge.

First-order logic is a very powerful language that is capable of describing complex relationships between objects while allowing uncertainty in the description and providing clear and intuitive semantics, but deductive reasoning in an unconstrained first-order theory is undecidable in principle. As argued by Levesque in [20], the only logically correct deductive technique that is feasible on very large knowledge bases is database retrieval under the closed-world assumption. Allowing unknown facts into a propositional knowledge base reduces deductive reasoning to propositional theorem proving, which is much harder than database retrieval, but still manageable. It is reasonable, then, to focus on fragments of first-order logic, trading some of the expressive power for computational benefits.

We focus on the fragment called *proper*$^+$, introduced in [17], for the remainder of this thesis. The reasons for this are twofold. First, Liu & Lakemeyer [26] show that a *proper*$^+$ initial KB is one of the two constraints that result in an *efficiently computable* local-effect progression. Second, the same constraint has been used for achieving efficient (albeit limited) query answering in first-order planning approaches.

2.4.1 *proper* and *proper*$^+$

The *proper*$^+$ normal form is an extension of the earlier *proper* normal form, introduced in [20]. We begin with presenting the framework on which both rely.

Let $\mathcal{L}$ be a first-order language with equality and no function symbols except for a countably infinite set of constants $\mathcal{C} = \{c_1, c_2, \ldots\}$. Let the set $\mathcal{E}$ be the union of axioms of equality (reflexivity, symmetry, transitivity, substitution of equals for equals) and the infinite set of unique name axioms for constants $\{c_i \neq c_j \,|\, i \neq j\}$. Let an *equality well-formed formula* (*ewff*) stand for a quantifier-free formula whose only predicate is equality. Let $\forall\phi$ denote the universal closure of ϕ. Let θ range over substitutions of all variables by constants, and let $\phi\theta$ denote the result of applying θ to ϕ. That is, a substitution is a mapping from variables to constants, and $\phi(x_1, x_2, \ldots)\theta = \phi(\theta(x_1), \theta(x_2), \ldots)$. For convenience, we will sometimes treat substitutions as tuples of constants $(c_1, \ldots, c_n)$ implying that, for $1 \leq i \leq n$, the i-th element of the tuple replaces the i-th free variable in a formula $\phi(x_1, \ldots, x_n)$ to produce $\phi\theta$. Given two sets of formulas S_1 and S_2, let $S_1 \models_{\mathcal{E}} S_2$ denote $\mathcal{E} \cup S_1 \models S_2$, and let $S_1 \equiv_{\mathcal{E}} S_2$ denote $S_1 \models_{\mathcal{E}} S_2$ and $S_2 \models_{\mathcal{E}} S_1$.

Definition 11. The set of formulas S is *proper* if $\mathcal{E} \cup S$ is consistent and S is a finite set of formulas of the form $\forall(e \rightarrow \rho)$ and/or $\forall(e \rightarrow \neg\rho)$, where e is an ewff and ρ is an atom.

Definition 12. A $\forall$-*clause* is a formula of the form $\forall(e \rightarrow d)$, where e is an ewff and d is a disjunction of literals whose arguments are distinct variables. The number of distinct variables in d is the *width* of the $\forall$-clause.

A knowledge base is *proper*$^+$ if it is a finite non-empty set of $\forall$-clauses. The *width* of a *proper*$^+$ KB is the maximum of the widths of the constituent $\forall$-clauses.

Given a *proper*$^+$ knowledge base $\mathcal{K}$, let $\texttt{gnd}(\mathcal{K})$ denote the set $\{d\theta \mid \forall(e \rightarrow d) \in \mathcal{K} \text{ and } \mathcal{E} \models e\theta\}$.

A *proper* knowledge base is a generalization of classical databases and can be seen as a finite representation of a possibly infinite set of ground literals. For example, $\forall(x \neq c_1 \rightarrow \neg P(x))$ denotes a countably infinite set of literals $\{\neg P(c_2), \neg P(c_3), \ldots\}$. Likewise, a *proper*$^+$ knowledge base is a finite representation of a possibly infinite set of ground clauses $\mathtt{gnd}(\mathcal{K})$. Observe that *proper* and *proper*$^+$ knowledge bases should always be interpreted with respect to the axioms $\mathcal{E}$ of equality and unique names for constants. For example, let $\mathcal{K}$ contain a single sentence $\forall x(x \neq c_1 \rightarrow \phi(x))$, where $\phi(x)$ is either a literal or a clause. Then $\mathcal{K} \models_{\mathcal{E}} \phi(c_2)$, but $\mathcal{K} \not\models \phi(c_2)$ since a model $\mathcal{M}$ of $\mathcal{K}$ could map c_1 and c_2 to the same domain element.

Remark. The requirement for d in Definition 12 to have only distinct variables as the arguments is not logically significant, since a $\forall$-clause which has constants or repeating variables as some of the arguments of the disjunction can be trivially transformed into an equivalent formula which agrees with the definition. Henceforth, we will relax this requirement when referring to *proper*$^+$. Note that Definition 5.5 from [26], which appears in this thesis as Definition 20 in Section 2.4.3, is consistent with the relaxed, but not the strict definition of *proper*$^+$.

2.4.2 Evaluation-Based Reasoning

Both *proper* and *proper*$^+$ were designed for logically limited reasoning that uses certain *evaluation procedures* as the reasoning mechanism. The evaluation procedures are based on the assumption that quantification can be understood substitutionally with respect to the countably infinite set of constants of the underlying language. In this section, we outline the benefits and shortcomings of the most prominent evaluation procedures.

For the remainder of this section, let c and c' range over constants, ρ range over atoms, l range over literals, and d range over clauses. Let $\bar{l}$ denote the complement of literal l. Let ϕ_c^x denote the result of replacing every free occurrence of x in ϕ by constant c. Let $H(\Gamma)$ denote the set of constants appearing in the set of formulas Γ and let $H_n^+(\Gamma)$ denote $H(\Gamma) \cup S$, where S is a set of n extra constants not occurring in Γ.

Procedure V

The evaluation procedure V for *proper*, introduced in [20], takes as input a *proper* knowledge base and an $\mathcal{L}$-sentence (a boolean query) and outputs one of three numerical values $\{0, \frac{1}{2}, 1\}$, which correspond to "false", "unknown", and "true".

Definition 13. Let $\mathcal{K}$ be a *proper* KB.

$$V[\mathcal{K}, \rho\theta] = \begin{cases} 1 & \text{if there exists } \forall(e \rightarrow \rho) \in \mathcal{K} \text{ s.t. } V[\mathcal{K}, e\theta] = 1, \\ 0 & \text{if there exists } \forall(e \rightarrow \neg\rho) \in \mathcal{K} \text{ s.t. } V[\mathcal{K}, e\theta] = 1, \\ \frac{1}{2} & \text{otherwise;} \end{cases}$$

$$V[\mathcal{K}, c = c'] = \begin{cases} 1 & \text{if } c \text{ is identical to } c', \\ 0 & \text{otherwise;} \end{cases}$$

$$V[\mathcal{K}, \neg\phi] = 1 - V[\mathcal{K}, \phi];$$

$$V[\mathcal{K}, \phi \wedge \psi] = min\{V[\mathcal{K}, \phi], V[\mathcal{K}, \psi]\};$$

$$V[\mathcal{K}, \forall x(\phi)] = min_{c \in H_1^+(\mathcal{K} \cup \{\phi\})} V[\mathcal{K}, \phi_c^x].$$

V is proved to be sound, but not necessarily complete, for all $\mathcal{L}$-queries. The incompleteness of V can be illustrated as follows: let $\mathcal{K}$ be $\{\forall x(x = c_1 \rightarrow P(x))\}$, and let the query be $\phi = P(c_2) \vee \neg P(c_2)$. Then $V[\mathcal{K}, \phi] = \frac{1}{2}$ ("unknown"), although $\mathcal{K} \models_{\mathcal{E}} \phi$ due to ϕ being a tautology.

In order to achieve completeness, Levesque introduced the normal form $\mathcal{NF}$ [20], later refined in [25]. For $\mathcal{NF}$-queries, V is complete. The following definition relies on the notion of *standard interpretations*, which are FOL interpretations where equality is interpreted as identity and the set of constants is isomorphic with the domain of discourse. This semantics for finite theories is captured by the axioms $\mathcal{E}$.

Definition 14. A set Γ of sentences is *logically separable* iff for every consistent set of ground literals L, if $L \cup \Gamma$ has no standard interpretation, then $L \cup \{\phi\}$ has no standard interpretation from some $\phi \in \Gamma$.

$\mathcal{NF}$ is the least set such that

- $\mathcal{NF}$ contains all ground literals and all ewffs;
- $\mathcal{NF}$ is closed under negation;
- if $\Gamma \subseteq \mathcal{NF}$ such that Γ is logically separable and finite, then $\bigwedge \Gamma \in \mathcal{NF}$;
- if $\Gamma \subseteq \mathcal{NF}$ such that Γ is logically separable and, for some ϕ, $\Gamma = \{\phi_c^x \mid c \in \mathcal{C}\}$, then $\forall x(\phi) \in \mathcal{NF}$.

As both [20] and [25] put it, the queries in $\mathcal{NF}$ are designed to contain no logical puzzles. In particular, $\mathcal{NF}$ includes all non-tautologous ground clauses and their complements, as well as all conjunctions of ground clauses that are closed under resolution. Levesque [20] presents some results that help with deciding whether a formula is in $\mathcal{NF}$. For example, a conjunction of sentences that make up a *proper* knowledge base is in $\mathcal{NF}$. A sentence all of whose literals are conflict-free* is also in $\mathcal{NF}$. Additionally, in [17], Levesque mentions that every negation-free sentence is in $\mathcal{NF}$.

In [25], it is shown that $\mathcal{NF}$ is strictly less expressive than FOL and that the possibility of obtaining compact $\mathcal{NF}$ representations for arbitrary propositional formulas is very unlikely. In exchange for

*Two literals are conflict-free iff either they have the same polarity or they use different predicates or they use different constants at some argument position.

these limitations, V is rather efficient: [27] proves that the combined complexity of V is NP-hard for conjunctive queries, but, at the same time, it can be implemented using database techniques with the efficiency comparable to that of database systems.

An extension of *proper* proposed in [9] allows unknown individuals into the knowledge base and the queries. The authors observe that *proper* knowledge bases have a built-in inifinitary version of the domain closure assumption. To remove this restriction, they extend the language $\mathcal{L}$ with a countably infinite set of constants called *labeled null values*, which are never mentioned by the axioms $\mathcal{E}$ and thus are not unique names. These null values can be used to express the properties of some elements of the domain which are known to exist but whose names are unknown. Inherent in the definition of null values is the ability to express certain kinds of disjunctive information. Those include disjunctions of ground literals that have the same name and polarity and disjunctions of ground literals which have the same arguments (see Examples 3–5 in [9]). The authors prove that V can be extended to handle unknown individuals both in the KB and the queries, while remaining sound and complete for $\mathcal{NF}$, and also that it remains efficient if the number of null values is logarithmic in the size of the KB and the width of the query. The properties of progression of *proper* KBs with unknown individuals have not been studied.

proper itself is not closed under classical progression even for the simplest BATs because classical progression inevitably contains disjunctive information, as illustrated in [24]. In response to this, [24] and [29] replace classical progression with *weak progression*, which is defined as the strongest *proper* KB entailed by the classical progression. They establish that, with the additional constraint of *context-completeness* (that is, the KB contains complete information about the context conditions of the SSA in a given situation), weak progression (1) is also context-complete, (2) coincides with classical progression, (3) is efficiently computable, and (4) query answering with respect to it is tractable for $\mathcal{NF}$ queries. These results also extend to context-free BATs. The restrictions associated with weak progression for *proper* severely limit the ability to express uncertainty and are thus of little relevance to the problem of conformant planning.

Procedure X

The evaluation procedure X for *proper*$^+$, introduced in [17], is an extension of V which is still sound and decidable but not complete even for $\mathcal{NF}$ queries due to the added expressiveness of *proper*$^+$ versus *proper*. In contrast to V, X returns either 1 or 0, corresponding, respectively, to "known to be true" and "not known to be true". Thus, to simulate the expressiveness of V, it is necessary to invoke X twice, to evaluate both the query and its negation, as summarized in the following table.

Table 2.1: Querying whether ϕ holds wrt $\mathcal{K}$

$X[\mathcal{K}, \phi]$	$X[\mathcal{K}, \neg\phi]$	Meaning
0	0	unknown
1	0	yes
0	1	no
1	1	$\mathcal{K}$ is inconsistent

Definition 15. Let S be a set of ground clauses. Define $UP(S)$ to be the closure of S under *unit propagation*: the least set which contains S and, if $\{l\} \cup d$ and $\bar{l}$ are in $UP(S)$, then so is d.

Let $\mathcal{K}$ be a *proper*$^+$ KB. $X[\mathcal{K}, \phi]$ returns 1 if one of the following holds, and 0 otherwise:

1. ϕ is a unit clause such that $\phi \in UP(\texttt{gnd}(\mathcal{K}))$;
2. ϕ is $(c = c')$ such that c is identical to c';
3. ϕ is $\neg(c = c')$ such that c is not identical to c';
4. ϕ is $\neg\neg\psi$ such that $X[\mathcal{K}, \psi] = 1$;
5. ϕ is $(\psi \vee \eta)$, such that there is a $\forall(e \to d) \in \mathcal{K}$ and $\theta \in H_k^+(\mathcal{K} \cup \{\psi, \eta\})$ for which $X[\mathcal{K}, e\theta] = 1$ and, for every literal $l \in d$, $X[\mathcal{K} \cup \{l\theta\}, \psi] = 1$ or $X[\mathcal{K} \cup \{l\theta\}, \eta] = 1$, where k is the number of free variables in d;
6. ϕ is $\neg(\psi \vee \eta)$ such that $X[\mathcal{K}, \neg\psi] = 1$ and $X[\mathcal{K}, \neg\eta] = 1$;
7. ϕ is $\exists x\psi$ and there is a $\forall(e \to d) \in \mathcal{K}$ and $\theta \in H_k^+(\mathcal{K} \cup \{\psi\})$ such that $X[\mathcal{K}, e\theta] = 1$ and, for every literal $l \in d$, there is a constant $c \in H_{k+1}^+(\mathcal{K} \cup \{\psi\})$ such that $X[\mathcal{K} \cup \{l\theta\}, \psi_c^x] = 1$, where k is the number of free variables in d;
8. ϕ is $\neg\exists x\psi$ such that $X[\mathcal{K}, \neg\psi_c^x] = 1$ for all $c \in H_1^+$.

X agrees with V with respect to *proper* knowledge bases. Its incompleteness with respect to *proper*$^+$ stems most prominently from its reliance on unit propagation and the way disjunctive and existential queries are handled: case analysis is performed shallowly, on a single clause from the knowledge base.

Logic of limited belief

A generalization of the evaluation-based query answering wrt *proper*$^+$ knowledge bases is proposed in [28] and [24] in the form of a logic of limited belief called the subjective logic $\mathcal{SL}$. Query answering in $\mathcal{SL}$ uses a procedure similar to X except it parametrizes the depth of case analysis. This development brings with it coherent semantics and tractability under certain reasonable conditions.

$\mathcal{SL}$ is a first-order logic whose atomic formulas are *belief atoms* of the form $\mathbf{B}_k\phi$, where $\mathbf{B}_k$ is a belief operator, $k \geq 0$ is the *depth of belief*, and ϕ is a well-formed formula of the language $\mathcal{L}$. $\mathcal{SL}$ formulas are equalities over terms of $\mathcal{L}$, belief atoms as described above, as well as formulas obtained from belief atoms using standard logical connectives and existential quantification. Consequently, every non-equality predicate must be within the scope of a belief operator, and the belief operators cannot be nested.

The semantics of $\mathcal{SL}$ is defined in terms of *setups* — sets of non-empty ground clauses, which are meant to represent explicit beliefs. The negation and disjunction have the usual meanings, and equality and quantification are understood with respect to the equality axioms $\mathcal{E}$. A belief atom $\mathbf{B}_k\phi$ is satisfied by a setup s (i.e., ϕ is a belief at a level k with respect to a setup s, denoted as $s \models\!\!\approx \mathbf{B}_k\phi$) if one of the following is true:

- ϕ is a clause, $k = 0$, and the unit resolution on the explicit beliefs s contains a subclause of ϕ;
- the subformulas of ϕ that are satisfied by s are sufficient to conclude that ϕ itself is satisfied;
- there is a clause in s that, when used for case analysis, results in the belief $\mathbf{B}_{k-1}\phi$ in all cases.

The result that is of the most interest to the present work is that of the decidability of $\mathcal{SL}$-based reasoning. The evaluation procedure W, introduced in [24] as a sight variant of X and defined below, exploits the fact that it is often sufficient to consider only a finite subset of the set of constants, because the unused constants are created equal and any one of them can serve to represent the behaviour of the rest. In the following definition, $gnd(\mathcal{K})|D$ denotes the subset of $gnd(\mathcal{K})$ which mentions no constants other than those in D.

Definition 16. Let $\mathcal{K}$ be $proper^+$, $k \geq 0$, $\phi \in \mathcal{L}$. Let j be the width of $\mathcal{K}$. Let D be $H_j^+(\mathcal{K} \cup \{\phi\})$. $W[\mathcal{K}, k, \phi]$ returns 1 if one of the following holds, and 0 otherwise:

1. $k = 0$, ϕ is a clause, and there exists $\phi' \in UP(gnd(\mathcal{K})|D)$ such that $\phi' \subseteq \phi$;
2. ϕ is $(c = c')$ and c is identical to c';
3. ϕ is $\neg(c = c')$ such that c is not identical to c';
4. ϕ is $\neg\neg\psi$ such that $W[\mathcal{K}, k, \psi] = 1$;
5. ϕ is $(\psi \vee \eta)$ but not a clause, such that $W[\mathcal{K}, k, \psi] = 1$ or $W[\mathcal{K}, k, \eta] = 1$;
6. ϕ is $\neg(\psi \vee \eta)$ such that $W[\mathcal{K}, k, \neg\psi] = 1$ and $W[\mathcal{K}, k, \neg\eta] = 1$;
7. ϕ is $\exists x\psi$ such that $W[\mathcal{K}, k, \psi^x_c] = 1$ for some $c \in D$;
8. ϕ is $\neg\exists x\psi$ such that $W[\mathcal{K}, k, \neg\psi^x_c] = 1$ for all $c \in D$;
9. $k > 0$, ϕ is a clause, a disjunction, or an existential, and there is a clause $d \in gnd(\mathcal{K})|D$ such that for every literal $l \in d$, $W[\mathcal{K} \cup \{l\}, k-1, \phi] = 1$.

W improves upon X by introducing a semantically sound notion of the level of belief k, which sets the depth of case analysis. It is proved in [24] that, for a $proper^+$ KB $\mathcal{K}$ and $\phi \in \mathcal{L}$, the setup $gnd(\mathcal{K})$ satisfies $\mathbf{B}_k\phi$ iff $W[\mathcal{K}, k, \phi] = 1$, and this kind of reasoning is decidable.

To address query answering in $\mathcal{SL}$, [24] defines answers to queries similarly to Definition 4. Here, $\mathcal{L}^j$ represents the subset of $\mathcal{L}$ where formulas have at most j variables.

Definition 17 ([24])**.** Let $\mathcal{K} \in \mathcal{L}^j$ be $proper^+$, $\phi \in \mathcal{L}^j$, and $k \geq 0$. Define $Ans(\mathcal{K}, \phi, k)$ as $\{\theta \mid gnd(\mathcal{K}) \approx\!\!\!\!\!\!= \mathbf{B}_k\phi\theta\}$.

The set $Ans(\mathcal{K}, \phi, k)$ may well be infinite, but [24] proves that there is a finite representation for it. Let $Ans(\mathcal{K}, \phi, k)|D$ to denote the restriction of $Ans(\mathcal{K}, \phi, k)$ to D. The notation $D \backsimeq H_m^+(\Gamma)$ states that D is the union of $H(\Gamma)$ and m extra constants.

Theorem 3 ([24]). *Let $D \backsimeq H_m^+(\mathcal{K} \cup \{\phi\})$ for some $m \geq j$. $Ans(\mathcal{K}, \phi, k)|D$ is a finite representation for $Ans(\mathcal{K}, \phi, k)$ in the following sense. Let θ be any substitution. Let $*$ be a bijection that is the identity on $H(\mathcal{K} \cup \{\phi\})$ and maps $\theta(x_i)$ into D for $i = 1, \ldots, j$. Then $\theta \in Ans(\mathcal{K}, \phi, k)$ iff $\theta^* \in Ans(\mathcal{K}, \phi, k)|D$.*

Using this result, [24] develops the procedure E which computes a set of answers to a query. E takes four arguments: the knowledge base $\mathcal{K}$, the query ϕ, the belief level k, and a set of constants D. Using database-like methods *division* and *projection*, E returns a relation over D, which is a semantically correct answer to ϕ wrt $\mathcal{K}$ at belief level k if D is properly selected, as stated in the following result.

Theorem 4 ([24]). *Let $\mathcal{K} \in \mathcal{L}^j$ be proper$^+$, $\phi \in \mathcal{L}^j$, and $k \geq 0$. Let $D \backsimeq H_m^+(\mathcal{K} \cup \{\phi\})$ for some $m \geq j(k+2)$. Then $E(\mathcal{K}, \phi, k, D) = Ans(\mathcal{K}, \phi, k)|D$.*

Here, since $m \geq j(k+2) > j$, $Ans(\mathcal{K}, \phi, k)|D$ is a finite representation of $Ans(\mathcal{K}, \phi, k)$ according to Theorem 3. In the subsequent complexity analysis, [24] establishes that E scales exponentially with the number of variables j and the depth of case analysis k. Thus, E provides decidable query answering for *proper*$^+$ knowledge bases and arbitrary queries in a variable-limited first-order logic.

2.4.3 Progression of *proper*$^+$ KB

The properties of progression of *proper*$^+$ knowledge bases have been extensively studied in [26]. In particular, it has been proved that if the context conditions of the successor state axioms are essentially quantifier-free, then *proper*$^+$ is closed under local-effect progression and such progression is efficiently computable. Theories that meet these constraints are called *well-formed* in [10] and are formally defined as follows.

Definition 18. A BAT $\mathcal{D}$ is *well-formed* if all of the following hold:

1. $\mathcal{D}$ is local-effect.
2. For every SSA $F(\bar{x}, do(a, s)) \leftrightarrow \Phi_F(\bar{x}, a, s)$ in $\mathcal{D}_{ss}$ and every action function $A(\bar{x})$, $\Phi_F(\bar{x}, A(\bar{x}), s)$ can be simplified using $\mathcal{D}_{una}$ to a quantifier-free formula.
3. $\mathcal{D}_{S_0}$ is *proper*$^+$.

Above, the second constraint can be understood in terms of transformed SSAs: every transformed SSA that can arise in $\mathcal{D}$ must be quantifier-free. SSAs (and sets thereof) that have this property are referred to in [26] as *essentially quantifier-free*.

In relation to the third constraint, note that $\mathcal{D}_{S_0}$ mentions terms of sort *situation*, which in general are not definable in the underlying language $\mathcal{L}$ of *proper*$^+$ since $\mathcal{L}$ forbids all functional symbols except for a countably infinite set of constants. While $\mathcal{D}_{S_0}$ is uniform in S_0 is thus prevented from mentioning the function symbol *do*, this issue becomes more severe for $\mathcal{D}_{ss}[\Omega]$, discussed in detail below, since it does mention *do*. In either case, there is no obstacle for expressing the respective theories as *proper*$^+$ for the purposes of progression. According to Theorem 1, computing progression involves reasoning about situations only in a very limited sense, so the situation terms can be safely *suppressed* from $\mathcal{D}_{S_0}$ for the

forgetting stage of the computation and reinstated afterwards; the notion of situation-suppressed terms and formulas is formally defined in [37] and notably used in [41] for a similar purpose.

The following results underpin the main findings of [26] and provide the necessary background for the discussion that follows. We omit full proofs in favour of short comments. Associated complexity results are also omitted.

Definition 19. A ground atom p is *irrelevant* to a sentence ϕ if $\texttt{forget}(\phi, p) \equiv \phi$.

Lemma 1 ([26], Proposition 5.3)**.** *Let p be a ground atom and let ϕ_1, ϕ_2, ϕ_3 be sentences such that p is irrelevant to them. Then $\texttt{forget}((\phi_1 \rightarrow p) \wedge (p \rightarrow \phi_2) \wedge \phi_3, p) \equiv (\phi_1 \rightarrow \phi_2) \wedge \phi_3$.*

This is essentially a propositional result and can be proved as such. Using Definition 10 and the fact that forgetting distributes over disjunction, we can transform the left-hand side into the equivalent formula $(\neg\phi_1 \wedge \phi_3) \vee (\neg\phi_1 \wedge \phi_2 \wedge \phi_3) \vee (\phi_2 \wedge \phi_3)$, which trivially transforms to the right-hand side with the help of propositional tautologies.

Lemma 2 ([26], Proposition 5.4)**.** *Let $\phi = \forall(e \rightarrow d)$ be a $\forall$-clause and $P(\bar{c})$ a ground atom. Suppose that for every atom $P(\bar{t})$ appearing in d, $e \wedge (\bar{t} = \bar{c})$ is unsatisfiable. Then $P(\bar{c})$ is irrelevant to ϕ.*

This result is rather intuitive in the light of Definitions 8 and 9. If the predicate symbol P does not appear in d, the claim holds trivially. Otherwise, since the interpretation of $P(\bar{t})$ affects neither the truth value of ϕ nor that of $\texttt{forget}(\phi, P(\bar{c}))$, the two theories have the same set of models.

Definition 20 ([26], Definition 5.5)**.** Let $\mathcal{K}$ be a *proper*$^+$ KB and $P(\bar{c})$ a ground atom. We say that $\mathcal{K}$ *is in normal form wrt* $P(\bar{c})$ if, for every $\forall(e \rightarrow d) \in \mathcal{K}$ and for every $P(\bar{t})$ appearing in d, either $\bar{t}$ is $\bar{c}$ or $e \wedge (\bar{t} = \bar{c})$ is unsatisfiable.

In other words, compared to plain *proper*$^+$, the normal form is syntactically explicit with respect to the semantics of P and $\bar{c}$.

Lemma 3 ([26], Proposition 5.6)**.** *Every proper*$^+$ *theory can be converted into an equivalent one which is in normal form wrt a given ground atom.*

A straight-forward conversion procedure is given in [26]. It is inspired by the syntactical transformation introduced for Theorem 2: replacing every occurrence of an atom $P(\bar{t})$ in a formula by the expression $[\bar{c} = \bar{t} \wedge P(\bar{c})] \vee [\bar{c} \neq \bar{t} \wedge P(\bar{t})]$ preserves the semantics of the formula. Performed on an arbitrary $\forall$-clause with n occurrences of the predicate symbol P, the result of this transformation can be trivially rewritten as a conjunction of 2^n $\forall$-clauses which conform to the definition of the normal form. For example, a $\forall$-clause $\forall(e \rightarrow P(\bar{t}))$ can be equivalently expressed as the conjunction $\forall(e \wedge \bar{t} = \bar{c} \rightarrow P(\bar{c})) \wedge \forall(e \wedge \bar{t} \neq \bar{c} \rightarrow P(\bar{t}))$.

Definition 21 ([26], Definition 5.7)**.** Let $\phi_1 = \forall(e_1 \rightarrow d_1 \vee P(\bar{t}))$ and $\phi_2 = \forall(e_2 \rightarrow d_2 \vee \neg P(\bar{t}))$ be two $\forall$-clauses, where $\bar{t}$ is a vector of constants or a vector of distinct variables. Without loss of generality we assume that ϕ_1 and ϕ_2 do not share variables other than those contained in $\bar{t}$. We call the $\forall$-clause $\forall(e_1 \wedge e_2 \rightarrow d_1 \vee d_2)$ the $\forall$-*resolvent* of ϕ_1 and ϕ_2 wrt $P(\bar{t})$.

Theorem 5 ([26], Theorem 5.8)**.** *Let $\mathcal{K}$ be a proper$^+$ KB and p a ground atom. The result of forgetting p in $\mathcal{K}$ is definable as a proper$^+$ KB.*

By Lemma 3, $\mathcal{K}$ can be converted into normal form wrt p, a theory $NF(\mathcal{K}, p)$. By the definition of the normal form, for each $\forall$-clause $\forall(e \to d) \in NF(\mathcal{K}, p)$ that does not contain p in d, $e \wedge (\bar{c} = \bar{t})$ is unsatisfiable, rendering p irrelevant to it as per Lemma 2.

For any two $\forall$-clauses $\phi_1, \phi_2 \in NF(\mathcal{K}, p)$ which mention p with opposite polarities, by Definition 21, we can compute a $\forall$-resolvent. By Lemma 1, the set of all $\forall$-resolvents of $NF(\mathcal{K}, p)$ together with the set of irrelevant $\forall$-clauses is the result of forgetting p from $\mathcal{K}$. Since the original theory is *proper*$^+$ and each $\forall$-resolvent is a *proper*$^+$ formula, the result is also a *proper*$^+$ theory.

Lemma 4 ([26], Proposition 5.11)**.** *If $\mathcal{D}_{ss}$ is essentially quantifier-free, then $\mathcal{D}_{ss}[\Omega]$ is definable as a proper$^+$ KB.*

Recall that the characteristic set $\Omega(s)$ of a progression with respect to a ground action is the set containing all atoms $F(\bar{c}, s)$ such that the constant tuple $\bar{c}$ is in the argument set Δ_F, for each fluent symbol F. $\mathcal{D}_{ss}[\Omega]$ is the instantiation of the successor-state axioms with respect to the characteristic set $\Omega(do(\alpha, S_0))$. That is, $\mathcal{D}_{ss}[\Omega]$ is a set of formulas of the form $F(\bar{c}, S_\alpha) \leftrightarrow \Phi_F(\bar{c}, \alpha, S_0)$, or, equivalently,

$$(\neg F(\bar{c}, S_\alpha) \vee \Phi(\bar{c}, \alpha, S_0)) \wedge (\neg \Phi(\bar{c}, \alpha, S_0) \vee F(\bar{c}, S_\alpha)). \tag{2.12}$$

Here, $F(\bar{c}, S_\alpha)$ is a ground atom and $\Phi(\bar{c}, \alpha, S_0)$ is the right-hand side of the SSA.

Recall that the SSAs of a well-formed BAT are local-effect and can be simplified using $\mathcal{D}_{una}$ to quantifier-free formulas. That is, for every $(n+1)$-ary fluent symbol F, each $\gamma_F^{\pm}$ in $\Phi(c_1, \ldots, c_n, \alpha, S_0)^\dagger$ is a disjunction of formulas of the form

$$\exists z_1 \cdots \exists z_k (\alpha = A(c_1, \ldots, c_n, z_1, \ldots, z_k) \wedge \phi_{cc}(c_1, \ldots, c_n, z_1, \ldots, z_k, S_0)),$$

which, upon substitution of a ground action $\alpha = A'(b_1, \ldots, b_{n+k})$ yield either a contradiction (when A and A' are different functional symbols) or quantifier-free ground formulas

$$(c_1 = b_1) \wedge \cdots \wedge (c_n = b_n) \wedge \phi_{cc}(c_1, \ldots, c_n, b_{n+1}, \ldots, b_{n+k}, S_0).$$

These can be further simplified using the equality axioms to either a contradiction or boolean combinations of ground atoms.

Let Γ^+, Γ^- be the sets of such transformed disjuncts of γ_F^+ and γ_F^- (respectively) which don't yield a contradiction due to $\mathcal{D}_{una} \cup \mathcal{E}$ when a ground action is substituted. Then $\Phi_F(\bar{c}, \alpha, S_0)$ becomes

$$\bigvee_{\varphi \in \Gamma^+} \varphi(\bar{c}, S_0) \vee F(\bar{c}, S_0) \wedge \neg \bigvee_{\varphi \in \Gamma^-} \varphi(\bar{c}, S_0), \tag{2.13}$$

which is also a boolean combination of ground atoms and, consequently, so is Equation (2.12). Thus,

†see Definition 6

$\mathcal{D}_{ss}[\Omega]$ can be expressed as a finite set of ground clauses, which becomes a subset of *proper*$^+$ when the situation arguments are suppressed as described above. However, following a remark in [41], note that a naive suppression of all situation terms from $\mathcal{D}_{ss}[\Omega]$ leads to a collision of atoms $F(\bar{c}, S_\alpha)$ and $F(\bar{c}, S_0)$, for each fluent symbol F. The solution to this is discussed in relation to the next result.

Theorem 6 ([26], Theorem 5.12)**.** *If $\mathcal{D}$ is a well-formed BAT, then progression of $\mathcal{D}_{S_0}$ with respect to any ground action is definable as a proper$^+$ KB.*

This follows immediately from Theorem 1, Definition 18, Theorem 5, and Lemma 4.

Note that the ground fluent atoms $F(\bar{c}, S_\alpha)$ are irrelevant to forgetting the set $\Omega(S_0)$ from $\mathcal{D}_{S_0} \cup \mathcal{D}_{ss}[\Omega]$ due to the situation term S_α: the foundational axioms Σ of situation calculus mandate that S_0 and $do(\alpha, S_0)$ are never mapped to the same object. Thus, when suppressing the situation term from $\mathcal{D}_{ss}[\Omega]$, a new propositional symbol must be introduced for each fluent symbol F to replace $F(\bar{c}, S_\alpha)$. In the result of forgetting, $F(\bar{c}, S_\alpha)$ should be reinstated, along with the suppressed situation terms.

The Grounding Trick

New results pertaining to progression of well-formed action theories and query answering in $\mathcal{SL}$ were obtained in [10]. In the paper, the authors develop a high-level reasoning mechanism based on situation calculus using the ideas of limited reasoning and finite representations, as outlined in the previous sections. They describe an alternative way to compute the progression of a well-formed BAT, one which is based on the ideas of finite representations and which is claimed by the authors, based on empirical evaluation, to be more efficient than that of [26].

The idea underlying the approach is the so-called "grounding trick", in which the initial *proper*$^+$ KB is converted into a finite propositional representation — a set of ground clauses. Similarly to [26], the authors exploit the merits of well-formed BATs, but on a propositional level. They introduce a propositional variant of progression and demonstrate that the result of converting it back to *proper*$^+$ is equivalent (up to $\mathcal{E}$) to progression of well-formed BAT as per [26].

As before, the width of a *proper*$^+$ KB, denoted j, is taken to be the maximum number of distinct variables in a $\forall$-clause of the KB. The following definitions and results assume that there is a set U of reserved constants $\{u_1, \ldots, u_j\}$ that do not appear in the initial KB and will never appear as arguments in a ground action. All constants from $\mathcal{C}$ which are not in U are referred to as *normal constants*. In this context, the notation $H(\Gamma)$ from Section 2.4.2 denotes the set of all normal constants appearing in Γ. Likewise, $H(\Gamma)^+_n$ denotes the set $H(\Gamma)$ extended with n normal constants which do not appear in Γ.

Let $\texttt{gnd}(\mathcal{K})|D$ denote the set $\{d\theta \mid \forall(e \to d) \in \mathcal{K}, \theta \in D, \models_{\mathcal{E}} e\theta\}$.

Definition 22 (Def.9, [10])**.** Let $\mathcal{K}$ be a *proper*$^+$ KB with width j. Let N be a set of normal constants containing those appearing in $\mathcal{K}$, i.e. $H(\mathcal{K}) \subseteq N$. Define $\texttt{prop}(\mathcal{K}, N)$ to be $\texttt{gnd}(\mathcal{K})|(N \cup U)$.

Note that whereas $\texttt{gnd}(\mathcal{K})$ is, in general, countably infinite, the set $\texttt{gnd}(\mathcal{K})|(N \cup U)$ is strictly finite. Moreover, it is a set of ground clauses, or CNF. Compared to *proper*$^+$, CNF is significantly easier to work with, which is the major benefit of this approach.

The essence of the trick is that the U-constants used for grounding can be taken as representatives for the countably infinite set of normal constants that do not participate in the grounding. Consider a variant of Theorem 3:

Lemma 5. *Let $\mathcal{K}$ be a proper$^+$ KB with width j. Let $N = \mathtt{const}(\mathcal{K})$ and let U be a set of new constants $\{u_1, \ldots, u_j\}$ disjoint from N. Then $\mathtt{gnd}(\mathcal{K})|(N \cup U)$ is a finite representation for $\mathcal{K}$ in the following sense. Let $\forall(e \to d)$ be an arbitrary $\forall$-clause from $\mathcal{K}$ and let θ be an arbitrary substitution compatible with d. Let $*$ be a bijection from $\mathcal{C}$ to $\mathcal{C}$ such that it is an identity on N and maps every member of θ into $(N \cup U)$. Then $d\theta \in \mathtt{gnd}(\mathcal{K})$ iff $d\theta^* \in \mathtt{gnd}(\mathcal{K})|(N \cup U)$.*

Proof. First, observe that $*$, due to being a bijection, maintains unique names for constants and does not affect equality. Thus, $\mathcal{E}$ and $\mathcal{E}^*$ are logically equivalent; in fact, they are identical. By the fact that $*$ is an identity on all normal constants, for a formula ϕ which contains only normal constants and for an arbitrary substitution θ, we trivially have

$$\phi = \phi^*, \quad (\phi\theta)^* = \phi\theta^*. \tag{2.14}$$

Second, let φ and ψ be two first-order sentences. Let $\mathcal{M}$ be an arbitrary interpretation such that $\mathcal{M} \models \mathcal{E}$ and let $\mathcal{M}^*$ be exactly like $\mathcal{M}$ except that $(c^*)^{\mathcal{M}^*} = (c)^{\mathcal{M}}$ and $(c)^{\mathcal{M}^*} = (c^*)^{\mathcal{M}}$. Notice that only the names of the constants are interchanged according to $*$, but their mapping to the domain is retained. Consequently, we have $\mathcal{M} \models \varphi$ whenever $\mathcal{M}^* \models \varphi^*$, and, since the set of models of φ and the set of models of φ^* are isomorphic, if φ is satisfied in all models, then φ^* is satisfied in all models, and vice versa:

$$\models_{\mathcal{E}} \varphi \quad \text{iff} \quad \models_{\mathcal{E}} \varphi^*. \tag{2.15}$$

Assume that, for some $\forall(e \to d) \in \mathcal{K}$ and some substitution θ, $d\theta$ belongs to $\mathtt{gnd}(\mathcal{K})$. By the definition of $\mathtt{gnd}$, we must have $\models_{\mathcal{E}} e\theta$. By equations (2.14, 2.15), $\models_{\mathcal{E}} e\theta$ iff $\models_{\mathcal{E}} e\theta^*$. Also, $\theta^* \in (N \cup U)$ by the definition of $*$. With all set inclusion conditions satisfied, we have $d\theta^* \in \mathtt{gnd}(\mathcal{K})|(N \cup U)$. □

For the next definition, recall that we use θ to range over substitutions of variables by constants and casually treat substitutions as constant tuples. Thus, given a $\forall$-clause $\forall(e \to d)$ where d is a disjunction of literals, $d\theta$ denotes a ground clause obtained from d by replacing all variables with constants from θ.

Definition 23 (Def.10, [10])**.** Let $\mathcal{K}$ be a *proper*$^+$ KB with width j and let $\mathcal{K}_p$ be $\mathtt{prop}(\mathcal{K}, N)$ for some N, $H(\mathcal{K}) \subseteq N$. Let ε be the ewff

$$\bigwedge_{i=1}^{j} \bigwedge_{c \in H(\mathcal{K}_p)} x_i \neq c \wedge \bigwedge_{i \neq k} x_i \neq x_k.$$

Define $\mathtt{FO}(\mathcal{K}_p)$ to be the set of $\forall$-clauses

$$\{\, \forall(\varepsilon \to d\theta(u_1/x_1, \ldots, u_j/x_j)) \mid d\theta \in \mathcal{K}_p \,\}.$$

The set $\texttt{FO}(\mathcal{K}_p)$ is obviously a *proper*$^+$ knowledge base, with each $\forall$-clause obtained in a straightforward way from a ground clause. Observe that all $\forall$-clauses of $\texttt{FO}(\mathcal{K}_p)$ share the same guarding ewff ε. The object assignments under which ε is satisfied are peculiar: they never map distinct variables to the same domain element, and they never map a variable to a domain element which is already mapped to by some constant.

Theorem 7 (Th.6, [10]). *With $\mathcal{K}$, N as before, we have $\texttt{FO}(\texttt{prop}(\mathcal{K}, N)) \equiv_{\mathcal{E}} \mathcal{K}$.*

This result asserts the semantic reversibility of finite grounding. In other words, no information is lost when a *proper*$^+$ KB is grounded as long as a sufficient number of U-constants is used. Clearly, $\texttt{FO}(\texttt{prop}(\mathcal{K}, N))$ does not have to be syntactically identical to $\mathcal{K}$; in fact, it tends to be much more verbose.

The following definition and theorem formalize the idea of using U-constants as representatives for normal constants; `egnd` stands for *extended grounding*.

Definition 24 (Def.11, [10]). Let B be a finite set of normal constants not occurring in $\mathcal{K}$, let $\mathcal{K}_p$ be $\texttt{prop}(\mathcal{K}, N)$ with $H(\mathcal{K}) \subseteq N$, and let c range over normal constants not in N. Define $\texttt{egnd}(\mathcal{K}, B)$ inductively as follows.

1. $\texttt{egnd}(\mathcal{K}_p, \emptyset) = \mathcal{K}_p$
2. $\texttt{egnd}(\mathcal{K}_p, \{c\}) = \mathcal{K}_p \cup \{d\theta(u_k/c) \mid d\theta \in \mathcal{K}_p, 1 \leq k \leq j\}$
3. $\texttt{egnd}(\mathcal{K}_p, \{c\} \cup B) = \texttt{egnd}(\texttt{egnd}(\mathcal{K}_p, \{c\}), B)$

Note that the new clauses are obtained from old clauses by substituting each constant from B in the place of each of the "representatives".

Theorem 8 (Th.7, [10]). *With $\mathcal{K}$, N, B as before, we have $\texttt{egnd}(\texttt{prop}(\mathcal{K}, N), B) \equiv_{\mathcal{E}} \texttt{prop}(\mathcal{K}, N \cup B)$.*

Simply put, extending a grounding of a KB using additional normal constants B is equivalent to grounding that KB using a set of normal constants which contains B.

Next, we define progression of a grounded KB (denoted `pprog`, meaning *propositional progression*).

Definition 25 (Def.12, [10]). Let $\mathcal{D}$ be a well-formed BAT, let $\mathcal{K}_p$ be $\texttt{prop}(\mathcal{D}_{S_0}, N)$ with $H(\mathcal{D}_{S_0}) \subseteq N$, let $\alpha = A(\bar{c})$ be a ground action, and let B be the set of constants appearing in $\bar{c}$ but not in $\mathcal{K}_p$. Define $\texttt{pprog}(\mathcal{K}_p, \alpha)$ as

$$\texttt{forget}(\texttt{egnd}(\mathcal{K}_p, B) \cup \mathcal{D}_{ss}[\Omega], \Omega(S_0))(S_0/S_\alpha).$$

This definition strongly resembles the result regarding first-order progression from Theorem 1. Like the progression of well-formed BATs from [26], it preserves the underlying syntactic form (*proper*$^+$ in [26], CNF here) and is efficiently computable.

Specifically, $\texttt{egnd}(\mathcal{K}_p, B)$ is CNF by definition; $\mathcal{D}_{ss}[\Omega]$ is CNF by Lemma 4; and $\Omega(S_0)$ is a set of ground atoms by definition. According to Definition 10, the result of forgetting a proposition p from a

CNF is a disjunction of two CNFs. It is noted in [10] that this operation can be performed in a straightforward way by computing all resolvents with respect to p and then removing all clauses containing p.

Indeed, let Ψ be set of ground clauses. It can be rewritten as a conjunction $\Psi_{pos} \wedge \Psi_{neg} \wedge \Psi_{irr}$ of pairwise disjoint sets of clauses which, respectively, contain literal p, contain literal $\neg p$, and contain neither p nor $\neg p$. Then

$$\Psi_{pos} = (p \vee C_1) \wedge \cdots \wedge (p \vee C_n) = p \vee (C_1 \wedge \cdots \wedge C_n) = p \vee \Psi'_{pos},$$

where Ψ'_{pos} is the set of clauses of Ψ_{pos} with p removed. Likewise, $\Psi_{neg} = \neg p \wedge \Psi'_{neg}$. Then $\Psi = (p \wedge \Psi'_{pos}) \wedge (\neg p \wedge \Psi'_{neg}) \wedge \Psi_{irr}$, and, by Definition 10,

$$\texttt{forget}(\Psi, p) = \Psi(p/false) \vee \Psi(p/true) = \Psi_{irr} \wedge (\Psi'_{pos} \vee \Psi'_{neg}).$$

The disjunction of two sets of clauses Ψ'_{pos} and Ψ'_{neg} is the set of all clauses $\psi = C_1 \cup C_2$ such that $C_1 \in \Psi'_{pos}$ and $C_2 \in \Psi'_{neg}$. This is exactly the set of resolvents of Ψ with respect to p.

Lemma 6 (Lemma 8, [10])**.** *Let p be a ground atom. Let N be a set of normal constants containing those that appear in p or a proper$^+$ KB $\mathcal{K}$. Then $\texttt{forget}(\mathcal{K}, p) \equiv_{\mathcal{E}} \texttt{FO}(\texttt{forget}(\texttt{prop}(\mathcal{K}, N), p))$.*

This result establishes the connection between forgetting an atom from a *proper*$^+$ KB and its finite propositional representation. Note the close resemblance between forgetting via propositional resolution (above) and forgetting via $\forall$-resolution in the discussion following Theorem 5.

Theorem 9 (Th.9, [10])**.** $\texttt{FO}(\texttt{pprog}(\texttt{prop}(\mathcal{K}, N), \alpha)) \equiv_{\mathcal{E}} \texttt{prog}(\mathcal{K}, \alpha)$

This follows immediately from Theorem 1, Theorem 7, Theorem 8, and the last lemma.

Thus, the progression of a well-formed BAT wrt a ground action can be computed by the means of straight-forward manipulations of CNF formulas. To implement a simple progression-based conformant planner, we also require an ability to query the knowledge base.

Querying a ground KB

In Definition 4, we defined *certain answers* to a query $\phi(\bar{x})$ wrt a knowledge base to be the set of constant tuples $\bar{c}$ such that $\phi(\bar{c})$ is entailed by the KB. Let us refine that definition in the context of *proper*$^+$ theories.

Definition 26. A *certain answer* to a query ϕ *with respect to a proper$^+$ knowledge base* $\mathcal{K}$ is the set of constant tuples $\{\theta \mid \mathcal{K} \models_{\mathcal{E}} \phi\theta\}$. For a boolean query, an empty tuple is an affirmative certain answer, read as *true*.

Henceforth, we will be using the term *answer* to refer to the certain answer from Definition 26.

Lemma 7. *Given a proper$^+$ KB $\mathcal{K}$ and a query ϕ, the set $\{\theta \mid \texttt{gnd}(\mathcal{K}) \models_{\mathcal{E}} \phi\theta\}$, denoted as $Ans(\mathcal{K}, \phi)$, is the certain answer to ϕ with respect to $\mathcal{K}$.*

Proof. By Definition 26, $\{\theta \mid \mathcal{K} \models_{\mathcal{E}} \phi\theta\}$ is the answer to ϕ wrt $\mathcal{K}$. Recall that a *proper*$^+$ KB $\mathcal{K}$ and the corresponding grounding $\texttt{gnd}(\mathcal{K})$ are, respectively, a finite and an infinite representation of the same knowledge, up to $\mathcal{E}$ [20, 17]; that is, $\mathcal{K} \equiv_{\mathcal{E}} \texttt{gnd}(\mathcal{K})$. Hence, $\mathcal{K} \models_{\mathcal{E}} \phi\theta$ iff $\texttt{gnd}(\mathcal{K}) \models_{\mathcal{E}} \phi\theta$. □

Similarly to the notation used in Definition 17, we use $Ans(\mathcal{K}, \phi)|D$ denote the set $\{\theta \mid \theta \in D, \texttt{gnd}(\mathcal{K}) \models_{\mathcal{E}} \phi\theta\}$, which is a restriction of $Ans(\mathcal{K}, \phi)$ to the set of constants D.

Likewise, mirroring Theorem 3, we show that such a restriction can be a finite representation for the answer.

Lemma 8. *Let $\mathcal{K}$ be a proper$^+$ KB with width j and let ϕ be a query with k free variables. Let $N = \texttt{const}(\mathcal{K} \cup \phi)$ and let U be a set of new constants $\{u_1, \ldots, u_n\}$ disjoint from N with $n = max[j, k]$. Then $Ans(\mathcal{K}, \phi)|(N \cup U)$ is a finite representation for $Ans(\mathcal{K}, \phi)$ in the following sense. Let θ be any substitution compatible with ϕ and let $*$ be a bijection from $\mathcal{C}$ to $\mathcal{C}$ such that it is an identity on N and maps every member of θ into $(N \cup U)$. Then $\theta \in Ans(\mathcal{K}, \phi)$ iff $\theta^* \in Ans(\mathcal{K}, \phi)|(N \cup U)$.*

Proof. Let φ and ψ be sentences such that $\varphi \models_{\mathcal{E}} \psi$ and let $\mathcal{M}$ be a model for φ such that $\mathcal{M} \models \mathcal{E}$. Recall the construction of $\mathcal{M}^*$ from $\mathcal{M}$ from the proof of Lemma 5. Then $\mathcal{M} \models_{\mathcal{E}} \psi$ and $\mathcal{M}^* \models_{\mathcal{E}} \psi^*$. Likewise, if $\varphi \not\models_{\mathcal{E}} \psi$ then there must be a model $\mathcal{M}$ of φ which is not a model for ψ, and the corresponding $\mathcal{M}^*$ is a model for φ^* but not for ψ^*. Thus,

$$\varphi \models_{\mathcal{E}} \psi \quad \text{iff} \quad \varphi^* \models_{\mathcal{E}} \psi^*. \tag{2.16}$$

By equation (2.16), $\texttt{gnd}(\mathcal{K}) \models_{\mathcal{E}} \phi\theta$ iff $\texttt{gnd}(\mathcal{K})^* \models_{\mathcal{E}} (\phi\theta)^*$. By equation (2.15), $\models_{\mathcal{E}} e\theta$ iff $\models_{\mathcal{E}} (e\theta)^*$. Observe that the set $\{\theta^* \mid \models_{\mathcal{E}} e\theta\}$ is identical to $\{\theta \mid \models_{\mathcal{E}} (e\theta), \theta \in (N \cup U)\}$. Thus the set $\texttt{gnd}(\mathcal{K})^* = \{d\theta^* \mid \forall(e \rightarrow d) \in \mathcal{K}, \models_{\mathcal{E}} e\theta\}$ can be equivalently expressed as $\{d\theta \mid \forall(e \rightarrow d) \in \mathcal{K}, \models_{\mathcal{E}} e\theta, \theta \in (N \cup U)\}$, which is exactly the set $\texttt{gnd}(\mathcal{K})|(N \cup U)$, whose logical consequences are a subset of those of $\texttt{gnd}(\mathcal{K})$. Therefore, $\texttt{gnd}(\mathcal{K}) \models_{\mathcal{E}} \phi\theta$ iff $\texttt{gnd}(\mathcal{K}) \models_{\mathcal{E}} \phi\theta^*$. □

2.5 Discussion

In this chapter, we formulated and proved a set of results, namely Lemmas 5, 7, and 8. Lemmas 5 is an explicit re-formulation of a hidden result from [10]. Lemma 7 is a minor yet original contribution. Lemma 8 is a new result inspired by a similar development from [24].

These results form the basis for the algorithm developed in the next chapter in the following sense. Note a peculiarity in the proof of Lemma 8: the finite representation $Ans(\mathcal{K}, \phi)|(N \cup U)$ for query ϕ can be obtained by deciding which tuples θ, generated over the finite set $(N \cup U)$, satisfy the entailment $\texttt{gnd}(\mathcal{K})|(N \cup U) \models_{\mathcal{E}} \phi\theta$. That is, the finite representation for the answer to a query can be obtained by repeatedly testing whether a finite CNF entails a ground query. If the query is additionally constrained to a quantifier-free formula, this testing reduces to simple propositional entailment and can be performed by an off-the-shelf SAT solver. This, together with the ability to express uncertain knowledge using *proper*$^+$ and the ability to efficiently compute progression using the grounding trick, provides sufficient machinery for a sound and complete conformant planner described in the next chapter.

Chapter 3

A Conformant Planner

In this chapter, we develop a sound and complete conformant planning algorithm and describe the design of a simple progression-based conformant planner. We limit our attention to well-formed basic action theories and quantifier-free queries.

3.1 Basic considerations

In the formulation of Geffner *et al.* [13], planning is a model-based autonomous behaviour, where the model is a variation of the *basic state model* — a generic STRIPS-like description of a state space. The introduction of PDDL and subsequent advances in directing the search in the PDDL state space owe to the wide acceptance of this approach by the planning community. Unfortunately, this classical planning approach restricts the ability of even the most elaborate algorithms to work with interesting domains, one of the limiting factors being its inherent domain closure assumption. On the other hand, dealing with an uncertain number of unnamed objects is a task which is encountered and solved by humans on a daily basis and which should not be overlooked simply because of its incompatibility with classical planning.

By using a first-order language to describe world states, we gain an ability to express structured knowledge about the application domain, including properties of and relationships between objects, which escapes from the classical planning setting. Furthermore, in the presence of quantification, domain objects do not need to be explicitly named or constrained to a finite set.

By engineering a specific fragment of situation calculus, we can efficiently compute progression of an incomplete initial theory. This paves the way for designing new planning algorithms which are capable of handling open domains and which operate on a precise specification of what a plan is. To provide said specification, we formulate the planning problem as a logical entailment. In doing so, we are not forced to use resolution or other deductive techniques to compute plans. Instead, we are free to design custom planning algorithms that search the tree of situations for a solution, possibly using heuristics to cut useless branches. The semantics of planning defined via an entailment allows us to prove soundness

and completeness of such algorithms.

3.1.1 The search space

Recall that a basic action theory $\mathcal{D}$ contains the initial state of the world in its subset $\mathcal{D}_{S_0}$, and the world dynamics in the form of action precondition axioms $\mathcal{D}_{ap}$ and the successor state axioms $\mathcal{D}_{ss}$. The goal conditions are formalized in the form of a first-order formula, which should be situation-dependent — otherwise, its truth value cannot be affected by any number of actions.

In Definition 2, we implicitly defined a planning problem as a pair $\langle \mathcal{D}, Goal(s) \rangle$, where $\mathcal{D}$ is a basic action theory and $Goal(s)$ is a first-order formula that is uniform in the situation term s and contains no free variables other than s. In the most general terms, solving a planning problem means deciding the entailment

$$\mathcal{D} \models \exists s \, (S_0 \sqsubseteq s) \wedge Goal(s), \tag{3.1}$$

that is, binding the situational variable to an object that corresponds to a sequence of actions that satisfies the goal condition. We define a solution to a planning problem to be a finite sequence of ground actions (a *plan*) $\alpha_1, \ldots, \alpha_n$ for which $\mathcal{D}$ entails $Goal(do([\alpha_1, \ldots, \alpha_n], S_0))$. We refer to the set of all plans for a given planning problem as the *search space*.

Consider the search space for an arbitrary planning problem. The search space contains plans of all lengths, of which there are as many as there are natural numbers. On its own, every plan is finite, but no planning algorithm can exhaustively search through an infinite search space. Therefore, every practical planner must impose an upper bound on the length of a plan. We can formally reflect that by introducing into the right-hand side of equation (3.1) the function $length(s)$ which maps every situation term $do([\alpha_1, \ldots, \alpha_n], S_0)$ to the number n of ground actions it involves. The following entailment is the decision problem for a practical planner:

$$\mathcal{D} \models \exists s \, (S_0 \sqsubseteq s \wedge Goal(s) \wedge length(s) \leq N), \text{ for some } N \geq 0. \tag{3.2}$$

In this new context, the search space is smaller, but still infinite: there is an infinite supply of constants, and thus of ground actions. To see the problem clearly, let us equivalently express the right-hand side of equation (3.2) by replacing quantification over situations with quantification over actions using the foundational axioms of situation calculus:

$$Goal(S_0) \vee \exists \alpha_1 \, Goal(do([\alpha_1], S_0)) \vee \ldots \vee \exists \alpha_1 \ldots \exists \alpha_N \, Goal(do([\alpha_1, \ldots, \alpha_N], S_0)). \tag{3.3}$$

Since equation (3.3) is a disjunction, it is satisfied iff at least one of the individual disjuncts is satisfied. To avoid long formulas, we will use for illustration the subformula $\exists \alpha_1 \, Goal(do([\alpha_1], S_0))$ representing plans with length 1.

Recall that the set $\mathcal{A}$ of action symbols in a BAT is finite. Additionally, recall that for every action name $A \in \mathcal{A}$ there is a precondition axiom of the form $\forall (Poss(A(\bar{x}, s)) \leftrightarrow \Pi_A(\bar{x}, s))$, which serves to

indicate whether a particular action is legal to perform. This information, implicit in quantification over actions, must be explicitly embedded into the statement of the planning problem when reducing quantification over actions to that over object variables. For plans of length 1, the corresponding subformula can be equivalently rewritten as

$$\bigvee_{A\in\mathcal{A}} \exists \bar{x}\, [Poss(A(\bar{x}), S_0) \wedge Goal(do(A(\bar{x}), S_0))]\,, \tag{3.4}$$

and similarly for arbitrary length plans. Thus, each subformula of Equation (3.3) responsible for a set of plans of some fixed length can be further broken down into sub-problems, each of which corresponds not only to plans of some fixed length k, but also to a particular sequence of action names.

Due to the infinite object domain, the quantification over objects cannot be rid of by the means of a finite disjunction. In the underlying language of well-formed BATs, we have a bijection between the object domain and the countably infinite set of constants $\mathcal{C}$. Using the constants, we can equivalently express equation (3.3) as an infinite disjunction. Albeit infinite, the result would be completely free of quantifiers. For instance, for plans of length k with $1 \leq k \leq N$, the corresponding subformula is

$$\bigvee_{\substack{A_1,\ldots,A_k\in\mathcal{A}\\ \bar{c}_1,\ldots,\bar{c}_k\in\mathcal{C}}} \begin{array}{l} Poss(A_1(\bar{c}_1), S_0) \wedge \ldots \wedge Poss(A_k(\bar{c}_k), do([A_1(\bar{c}_1), \ldots, A_{k-1}(\bar{c}_{k-1})], S_0)) \wedge \\ \wedge\, Goal(do([A_1(\bar{c}_1), \ldots, A_k(\bar{c}_k)], S_0)). \end{array} \tag{3.5}$$

An optimization is readily available. For any constant tuple $\bar{c}$, $Poss(A(\bar{c}, s))$ holds if and only if $\Pi_A(\bar{c}, s)$ does. For each occurrence of the predicate $Poss$ in a formula such as (3.4), the set of all such tuples can be obtained by posing $\Pi_A(\bar{x}, s)$ as a query with respect to the BAT. For example, for the case of single-action plans, the set of ground action arguments that make $Poss(A(\bar{x}), S_0)$ true is $\{\bar{c} \mid \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \Pi_A(\bar{c}, S_0)\}$. Observe that those disjuncts of Equation (3.5) that involve constant tuples $\bar{c}_1 \notin \{\bar{c} \mid \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \Pi_A(\bar{c}, S_0)\}$ are trivially false and can be dropped, while those that involve tuples from the answer set can be simplified by removing the trivially true occurrence of $Poss$. In the case of plans of length 1, equation (3.4) can be equivalently rewritten as

$$\bigvee_{A\in\mathcal{A}} \bigvee_{\mathcal{D}_{S_0}\cup\mathcal{D}_{una}\models\Pi_A(\bar{c},S_0)} Goal(do(A(\bar{c}), S_0)), \tag{3.6}$$

and similarly for plans of arbitrary length, with a caveat: for plan lengths $k > 1$, query answering involves solving a projection problem, which will be discussed below.

Equations (3.5, 3.6) suggest the iterative deepening search strategy for the planner, outlined in Algorithm 1. Although this algorithm is not guaranteed to terminate, it is sound and complete for plans with length up to and including N.

3.1.2 Query answering using SAT and the projection problem

To arrive at a decidable algorithm, we employ the idea of finite representations developed in Section 2.4.3. Using the "grounding trick", we can convert the $proper^+$ KB $\mathcal{K}$ representing initial knowledge of a well-formed BAT into a finite set of ground clauses $\mathcal{K}_p$ and compute a finite representation of the

Algorithm 1: Iterative deepening search for a plan

Data: a planning problem $\langle \mathcal{D}, Goal(s) \rangle$
Data: plan length bound $N \geq 0$
Result: a solution to the planning problem or the empty set
foreach $k \in (0, \ldots, N)$ **do**
  foreach $(A_1, \ldots, A_k) \in \mathcal{A}^k$ **do**
    foreach $\bar{c}_1 \in \{\bar{c} \mid \mathcal{D} \models \Pi_{A_1}(\bar{c}, S_0)\}$ **do**
      $\cdots$
        foreach $\bar{c}_k \in \{\bar{c} \mid \mathcal{D} \models \Pi_{A_k}(\bar{c}, do([A_1(\bar{c}_1), \ldots, A_{k-1}(\bar{c}_{k-1}))], S_0))\}$ **do**
          if $\mathcal{D} \models Goal(do([A_1(\bar{c}_1), \ldots, A_k(\bar{c}_k)], S_0)$ **then**
            return $[A_1(\bar{c}_1), \ldots, A_k(\bar{c}_k)]$
return $\emptyset$

answers to quantifier-free queries with respect to it. Finite answer sets can be used to turn infinite disjunctions such as equation (3.6) into finite ones, resulting in a finite search space for the planner.

Algorithm 2: $QA(\mathcal{K}, \phi(\bar{x}))$

Data: a $proper^+$ KB $\mathcal{K}$
Data: a quantifier-free query $\phi(\bar{x})$
Result: the set Ans of constant tuples
$N = \texttt{const}(\mathcal{K}) \cup \texttt{const}(\phi)$
$U = \{u_1, \ldots, u_{max[j,k]}\}$
$\mathcal{K}_p = \texttt{gnd}(\mathcal{K})|(N \cup U)$
$Ans = \emptyset$
foreach $\theta \in (N \cup U)$ **do**
  if $\mathcal{K}_p \wedge \neg\phi\theta$ *is UNSAT* **then**
    $Ans = Ans \cup \theta$
return Ans

The output of Algorithm 2 is a set of constant tuples which may include any of the reserved U-constants. Such tuples cannot be used directly as answers to the query: they need to be interpreted according to the context. Specifically, an answer tuple $(t_1, \ldots, u_i, \ldots, t_k)$ represents an infinite set of tuples $\{(t_1, \ldots, c, \ldots, t_k) \mid c \in \mathcal{C} \setminus (\texttt{const}(\mathcal{K}) \cup \texttt{const}(\phi))\}$, according to Lemma 8. Algorithm 3, a slight version of Algorithm 2, computes such context-dependent sets of tuples; specifically, it replaces U-constants in the answers with fresh normal constants, which can later be used to extend the knowledge base. Note that Algorithm 3 takes an additional input, a set of normal constants. This is done for convenience in defining Algorithm 4 later on.

Note that Algorithms 2 and 3 can only be applied when the query and the knowledge base are uniform in the same situation term. Whenever this condition is violated, the output of either algorithm is undefined. Thus, to answer a query $\phi(\bar{x}, s_2)$ wrt a KB $\mathcal{K}(s_1)$ such that s_1, s_2 are ground situation terms and $s_1 \sqsubset s_2$, we need to progress $\mathcal{K}(s_1)$ through the action sequence that separates s_1 from s_2. This can be done using either of the two approaches to progression discussed in Section 2.4.3. In our implementation, we focus on propositional progression from Definition 25 to minimize overhead, since

Algorithm 3: $QAC(\mathcal{K}, \phi(\bar{x}), C)$

Data: a $proper^+$ KB $\mathcal{K}$
Data: a quantifier-free query $\phi(\bar{x})$
Data: a set of normal constants C
Result: the set Ans of context-dependent normal constant tuples
$N = \texttt{const}(\mathcal{K}) \cup \texttt{const}(\phi) \cup C$
$U = \{u_1, \ldots, u_{max[j,k]}\}$
$\mathcal{K}_p = \texttt{gnd}(\mathcal{K})|(N \cup U)$
$Ans = \emptyset$
foreach $\theta \in (N \cup U)$ **do**
  if $\mathcal{K}_p \wedge \neg\phi\theta$ *is UNSAT* **then**
    if θ *contains* U*-constants* **then**
      replace each U-constant in θ with a new normal constant
    $Ans = Ans \cup \theta$
return Ans

the knowledge base needs to be grounded for query answering anyway.

In doing so, we can altogether dispense with the $proper^+$ representation of the knowledge and maintain only the grounded version of the knowledge base.

3.1.3 Planning algorithm

We summarize the preceding developments in the following essential result.

Algorithm 4: Planning algorithm

Data: a planning problem $\langle \mathcal{D}, Goal(s) \rangle$
Data: plan length bound $N \geq 0$
Result: a solution to the planning problem or the empty set
foreach $k \in (0, \ldots, N)$ **do**
  foreach $(A_1, \ldots, A_k) \in \mathcal{A}^k$ **do**
    foreach $\bar{c}_1 \in QAC(\mathcal{D}, \Pi_{A_1}(\bar{x}, S_0), \emptyset)$ **do**
      $\ldots$
        foreach $\bar{c}_k \in QAC(\mathcal{D}, \Pi_{A_k}(\bar{x}, do([A_1(\bar{c}_1), \ldots, A_{k-1}(\bar{c}_{k-1}))]), \bar{c}_1 \cup \ldots \cup \bar{c}_{k-1})$ **do**
          if $\mathcal{D} \models Goal(do([A_1(\bar{c}_1), \ldots, A_k(\bar{c}_k)], S_0)$ **then**
            return $[A_1(\bar{c}_1), \ldots, A_k(\bar{c}_k)]$
return $\emptyset$

Theorem 10. *Algorithm 4 terminates and implements a sound and complete conformant planner with bounded plan length.*

For proof, refer to Sections 3.1.1 and 3.1.2.

3.2 Design

Algorithm 4 was implemented in ECLiPSe Prolog [39] and, for the experimental evaluation below, uses the state-of-the-art SAT solver Glucose [1].

3.2.1 Planning domain and instance specification

To represent logical formulas in our implementation, we use a quantifier-free subset $\mathcal{L}'$ of the underlying language $\mathcal{L}$ of *proper*$^+$ described in Section 2.4.1 with its syntax adjusted as follows. In the expressions, we use Prolog operators $\{\texttt{neg}, \texttt{v}, \texttt{\^{}}, \texttt{->}, \texttt{:}\}$ which respectively correspond to logical symbols $\{\neg, \vee, \wedge, \rightarrow, =\}$ with their usual meanings. Variable symbols are represented by 0-ary Prolog terms of the form $\texttt{x}N$, where N is a positive integer. Similarly, constant symbols are represented by Prolog terms of the form $\texttt{c}N$ or $\texttt{u}N$, depending on whether it is a normal constant or a U-constant. Prolog term equality is used to test for variable or constant name equality. Fluents and static predicates are represented by Prolog terms of arbitrary arity whose arguments are variables or constants as described above. Fluents do not have an explicit situational argument; a predicate is treated as a fluent whenever it is declared as such. To represent tautology and contradiction, we use Prolog terms `tRUE` and `fALSE`.*

For convenience, we use Prolog lists of Prolog terms to represent ground clauses. Sets of such lists can be further stored in Prolog lists; those are assumed to represent conjunctions of clauses, i.e. CNF.

At the beginning of the domain specification, we require that all fluent names are declared in a Prolog list using the Prolog predicate `fluents/1`:

`fluents([`fluent_name$_1$`,...,`fluent_name$_n$`]).`

where each fluent_name$_i$ is a 0-ary Prolog term.

Action precondition axioms are defined using the Prolog predicate `ap/2`:

`ap(`action_name(Args)`,` expression(Args)`).`

where action_name(Args) is a Prolog term whose name is the unique action name and whose arguments Args are Prolog variables; expression(Args) is an $\mathcal{L}'$-formula whose variables are Args.

Successor state axioms are defined in their transformed form (see Definition 7) wrt each action name using the Prolog predicate `ssa/4`:

`ssa(`fluent_name(_,...)`,` action_name(Args)`,`Gamma$^+$`,`Gamma$^-$`).`

where

- fluent_name(_,...) is a Prolog term whose name is one of the fluent names, whose arity corresponds to the fluent's arity, and whose arguments are anonymous Prolog variables;

*The first letter is in lower case due to Prolog syntax.

- action_name(Args) is a Prolog term whose name is one of the action names and whose arguments are Prolog variables;

- Gamma$^+$ and Gamma$^-$ are lists representing disjunctions γ^+ and γ^- from Definition 6. Each member of either disjunction is the result of eliminating the action names using $\mathcal{D}_{una}$ from the corresponding subformula of the original successor state axiom. Specifically, each such member is a Prolog list of length 2 of the form `[[Args_1],expression(Args_2)]`, where all members of Args_1 and Args_2 are Prolog variables from the arguments Args of the action name, Args_1 are the action arguments which are assigned to the fluent's arguments (the argument list must follow the fluent's argument arity and order), and expression(Args_2) is an $\mathcal{L}'$-formula representing the context condition.

Initial state of a problem instance is described using the Prolog predicate `fclause/1` (meaning $\forall$-clause) whose argument must be an $\mathcal{L}'$-formula of the form ewff -> clause, where ewff never mentions predicates of $\mathcal{L}'$ but mentions equality (`:`), and clause is a disjunction of $\mathcal{L}'$-literals whose arguments are variables as described above. All variables are assumed to be universally quantified. The variable names in every single $\forall$-clause must form a continuously numbered sequence of names starting from `x1`.

Goal state is described using the Prolog predicate `goal/1` whose argument is an $\mathcal{L}'$-sentence.

3.2.2 Search logic

The main rule of the program is `solve(Plan, Max)`, where `Plan` is the return variable and `Max` is the upper bound on the length of the plan. The rule implements a basic iterative deepening depth-first search in the space of states represented by Prolog terms of the form `state(KB, Const)`, where `KB` is a set of ground clauses represented by Prolog lists and `Const` is the list of all constants that appear in `KB`.

```
solve(Plan, Max) :- loadUnsat, max_length(Plan, Max),
    reachable(State, Plan), goal_state(State).
```

The rule for `loadUnsat` loads the Glucose SAT solver binding as an external predicate `unsat/1`. `max_length(Plan, Max)` unifies `Plan` with an uninitialized list whose length starts at zero and is incremented on each backtracking call up to and including the value of `Max`. `reachable(State, Plan)` is a recursive rule that returns a state `State` obtained by generating an executable sequence of ground actions that fills the list `Plan` and computing the progression of the initial state knowledge with respect to it. `goal_state(State)` succeeds iff `State` entails the formula describing the goal conditions. Backtracking from a failing goal to `reachable` iterates over all plans of current size, guaranteeing an exhaustive search.

The rule for `reachable` generates a state as follows. For a zero-length plan, it delegates to the rule

`initial_state/1`:

```
initial_state(state(PropKB, Const)) :-
    getKB(FOL_KB), propKB(FOL_KB, PropKB, Const).
```

Here, `getKB/1` merely gathers all initial state ∀-clauses in a list. `propKB/3` parses the list, extracts from it a list `C` of all normal constant symbols and a list `V` of all variable symbols, infers the width j of the KB from `V`, generates a list `U` of exactly j continuously numbered U-constants, and extracts all normal constants from the goal formula, appending them to `C`. Finally, it invokes the rule `prop/3` on each ∀-clause to generate a finite CNF representation $\texttt{gnd}(\texttt{FOL\_KB})|(\texttt{C} \cup \texttt{U})$ of the KB as per Lemma 5, which is stored in the return variable `PropKB`.

For non-empty plans, `reachable` recursively builds up on the grounded initial state using the rule `legal_move/3`:

```
legal_move(state(KB2, Const2), [A|History], state(KB1, Const1)) :-
    poss(A, state(KB1, Const1)),
    not useless(A, History, KB1),
    progress(A, state(KB1, Const1), state(KB2, Const2)).
```

Here, `poss/2` browses through action precondition axioms declared using the predicate `ap/2` described above, unifies the variable `A` with the axiom's action argument thus selecting an action name, instantiates its arguments with constants from `Const1` and invokes the predicate `entails/2` to establish using the SAT solver whether `KB1` logically entails the instantiated precondition formula for the action. Backtracking ensures an exhaustive search over all precondition axioms and thus action names, as well as all substitutions of constants, including U-constants, as action arguments. When `poss` selects an action with one or more U-constants as action arguments and confirms that the action is possible to execute, it introduces new normal constant names (one for each distinct U-constant) and substitutes them in place of the U-constants in the action term. The grounding of the KB is consequently extended with respect to the new constants when the progression is computed.

The rule `useless/3` defines domain-specific declarative heuristics. Taking the current knowledge base and the history of actions as inputs allows for meaningful reasoning about the usefulness of the newly selected ground action, i.e. deciding whether the execution of the action will bring the agent closer to the goal. If a selected action is deemed possible and useful, `legal_move` computes the progression of

`state(KB1, Const1)` with respect to the action using the rule `progress/3` according to Definition 25:

```
progress(GAction, state(PropKB, Const1), state(ProgressedKB, Const2)) :-
    align(state(PropKB, Const1), GAction, state(ExtPropKB, Const2)),
    charSet(GAction, Omega),
    dssOmega(Omega, GAction, DssSet),
    union(ExtPropKB, DssSet, KB_newinfo),
    forgetMultiple(KB_newinfo, Omega, KB_forget),
    unrename(KB_forget, KB_restored),
    cleanup(KB_restored, KB_raw_clean),
    fixGrounding(KB_raw_clean, Const2, ProgressedKB).
```

Here, `align/3` checks whether the ground action introduces new constants and, if so, extends the grounding of `PropKB` onto those constants according to Definition 24. It then invokes the rule `charSet/2` to compute the characteristic set `Omega` of the ground action according to Definition 7. Next, the rule `dssOmega/3` is used to compute $\mathcal{D}_{ss}[\Omega]$ — the instantiation of all transformed SSAs declared using `ssa/4` with respect to each ground atom in the characteristic set. Each thus instantiated formula is simplified using the rule `simplify/2` and converted into a list-based CNF using the rule `clausalForm/2`.[†]

Next, this new knowledge is merged with the extended grounded KB to yield `KB_newinfo`, and the outdated knowledge represented by `Omega` is forgotten from it using simple propositional resolution implemented in the rule `forgetMultiple/3` according to the discussion on page 32. Recall from the discussion following Theorem 6 that suppressing the situational argument from fluents while forgetting old knowledge from a raw progression leads to a conflict between the atoms representing the same fluent at different situations. To circumvent this, `dssOmega` wraps all conflicting atoms in $\mathcal{D}_{ss}[\Omega]$ in a Prolog term `wrap/1`; the rule `unrename/2` in `progress` simply removes the wrapping since the forgetting resolves the conflict. The optimizing rule `cleanup/2` discards contradictory clauses from the resulting KB, as well as redundant literals from the non-contradictory clauses.

The final and crucial step of the progression computation is the rule `fixGrounding/3`. Observe that the output of the previous steps, `KB_raw_clean`, is not equivalent to the grounding of a first-order progression of `PropKB`: its grounding is incomplete because it was based on a KB of a possibly smaller width. The rule `fixGrounding` implements a regrounding procedure according to Theorem 9, ensuring that the result is indeed a propositional representation of the first-order progression.

3.3 Experiments

We tested the planner on three different domains which have previously appeared in conformant planning literature. The domains were selected in order to illustrate different aspects of the abilities of our planner.

[†]Both `simplify/2` and `clausalForm/2` were borrowed from the prime implicate compiler for the planner *wspdf*

The domain "Cube" is essentially propositional and relates the performance of our algorithm to classical conformant planning. The domain "Adder" illustrates the absence of the domain closure assumption in our approach; additionally, it poses the challenge of a very large search space. The domain "Blocks World" is the most balanced of the three in the sense that it incorporates a large search space, a need for unnamed objects, and multiple fluents that result in a fast-growing progression.

All experiments in this thesis were performed on a dual-core Intel Core i3-380M CPU at 2.53GHz with 4GB RAM. The run times were obtained by averaging the CPU times over three runs; the standard deviation was insignificant.

3.3.1 Cube

The Cube domain appeared in the conformant track of IPC2006 and was used in the experimental evaluation of the planner T_0 in [33]. We use a slightly modified version to accommodate the differences in the planning formalisms. The objective of the agent is to successfully navigate inside a discrete 3-dimensional space. The geometry of the cube is described using the predicates $xabove(x, y)$, $yabove(x, y)$, $zabove(x, y)$, which describe the spatial relationships between the positions for each of the axes. Since the position constants act as natural numbers with respect to which the axes are defined, a correct world description must specify the same order of the positions on all axes. For example, the following is a correct partial description of the geometry of a cube with a side of length 3.

$$xabove(c_2, c_1), xabove(c_3, c_2),$$
$$yabove(c_2, c_1), yabove(c_3, c_2),$$
$$zabove(c_2, c_1), zabove(c_3, c_2)$$

Since our formalism does not implement CWA, the boundaries of the cube may be defined by sentences like $\forall(\neg xabove(c1, x))$, meaning that there is no position below $c1$, or they may be left unknown. Note that it is impossible within our formalism to describe a truly unbounded cube: this would require sentences like $\forall x \exists y(xabove(x, y))$, which invariably require existential quantification. We discuss this issue in the conclusion of this chapter.

The position of the agent is described by three fluents $xpos(x, s)$, $ypos(x, s)$, $zpos(x, s)$, where the object argument in each case is a position. The uncertainty about initial position of the agent can be described as follows:

$$xpos(c_1, S_0) \vee xpos(c_2, S_0) \vee xpos(c_3, S_0),$$
$$ypos(c_1, S_0) \vee ypos(c_2, S_0),$$
$$zpos(c_3, S_0)$$

Here, the exact position of the agent is unknown, but it is known to be within a $3 \times 2 \times 1$ cuboid near the origin. In the terminology of Section 2.1.2, this description corresponds to a belief state. To measure the problem's degree of incompleteness ("DoI" in the tables below), we use the number of possible worlds in

the initial belief state for named positions only (i.e., not including the properties of the infinitely many unnamed positions). Provided that we axiomatically enforce that the agent cannot be at two distinct positions at once, the degree of incompleteness in the previous example is $3 \times 2 \times 1 = 6$. The statement "an agent cannot be at more than one x-axis position at once" can be captured by the formula

$$\forall(x \neq y \rightarrow \neg xpos(x) \vee \neg xpos(y));$$

alternatively, it can be enforced by the successor state and precondition axioms, as we do below.

The agent is able to attempt to move one unit at a time along any axis using the actions $xmove(x, y)$, $ymove(x, y)$, $zmove(x, y)$. The precondition of each action requires the arguments to be adjacent positions on the respective axis, but does not take into account the position of the agent; thus, execution of the action may or may not have an effect. The dynamics of the world is axiomatized for the x-axis as follows, and similarly for y- and z-axes:

$$\begin{aligned} Poss(xmove(x, y), s) \;&\leftrightarrow\; xabove(x, y) \vee xabove(y, x), \\ xpos(x, do(a, s)) \;&\leftrightarrow\; \exists z(a = xmove(z, x) \wedge xpos(z, s)) \\ &\quad \vee xpos(x, s) \wedge \neg\exists z(a = xmove(x, z)). \end{aligned}$$

In comparison to the original global-effect PDDL description of Cube, the present version is forced to list all objects that may be affected by the execution of an action as the arguments of that action. As discussed in Section 2.3.1, this is mandated by our choice of the mechanism for progressing knowledge bases. Unfortunately, this dramatically increases the search space. While computing each step of a plan, the planner must consider not the mere six actions as in the original formulation, but at least $3n^2$ ground actions, where n is the known size of the cube. For example, there is a choice of 27 different ground actions for the first step of the instance `cube3-1.pl` (see below.) To alleviate this burden, we introduce a set of domain-specific declarative heuristics in the spirit of [12]. The heuristic rules are implemented by the Prolog predicate `useless` mentioned in Section 3.2.2. For Cube, the heuristics can be summarized as follows:

- Do not perform the same action twice;

- Do not revert the effects of an already performed action;

- Do not move along an axis if already at correct position on it;

- Explore each axis separately and in order.

The experimental data below confirms the effectiveness of these heuristic rules. Their Prolog implementation can be found in Appendix 1. It is important to note that some of the rules rely on a specific goal formula structure. In other words, we define heuristics using both the `useless` predicate and the goal formula. This is a peculiarity of our implementation and applies to other domains as well.

Table 3.1: Cube test runs. Run times were limited to 300 seconds of CPU time.

Instance	Cube size	DoI	Plan length	CPU time (s)	
				blind	heuristic
cube2-1.pl	2	8	3	0.10	0.06
cube3-1.pl	3	4	4	14.51	0.68
cube3-2.pl	3	3	5	235.15	1.46
cube3-3.pl	3	18	5	271.56	1.62
cube3-4.pl	3	8	6	-	1.82
cube3-5.pl	3	27	6	-	2.97
cube4-1.pl	4	64	9	-	95.73
cube5-1.pl	5	5	5	-	127.74
cube5-2.pl	5	125	12	-	-

Experimental data

We ran the planner on several instances of Cube, which differ in the size of the cube and that of the initial belief state. The run times are summarized in Table 3.1.

The instances differ the number of named positions (Cube size) and the degree of uncertainty in the initial situation (as described above). An inferred but more explicit metric is the length of the plan that satisfies the goal. In the table, we provide the length of the shortest plan.

It is evident from the data that the brute-force search falls from the computational cliff almost immediately. The heuristic search is superior in comparison but in the long run still suffers from the exponential explosion of the search space.

3.3.2 Adder

The Adder domain is a version of a domain of the same name which appeared in the conformant track of IPC2006 and, like Cube, was used in [33]. The objective is to generate a logical circuit which computes a goal boolean function. The domain objects represent bits which may take one of the two boolean values and some of which are marked as immutable. In the original formulation, due to DCA, instances are constrained to utilize a finite set of bits, which results in circuits that reuse the mutable bits over and over as intermediate nodes of the circuit. Due to the expressiveness of our formalism, our version of the domain allows to drop this unnatural requirement.

The fluents of the domain are $constant(x, s)$ and $high(x, s)$. The former reflects the mutability of a bit, i.e., whether the bit is yet undriven (and thus can be used as the output of a logical gate) or driven (and thus can only act as an input). The fluent $high(x, s)$ reflects the boolean value of a particular bit: $high(c_1, s)$ means that the bit c_1 is high, and $\neg high(c_2, s)$ means that the bit c_2 is low. The initial state, in general, is a possibly incomplete truth assignment on bits along with their incompletely known

mutability properties:

$$constant(c_1, S_0), constant(c_2, S_0), \neg constant(c_4, S_0),$$
$$high(c_1, S_0), \neg high(c_3, S_0), high(c_4, S_0).$$

From a practical point of view, however, there is little room for uncertainly in the initial state of Adder. To describe the desired behaviour of a logical function, we introduce its k inputs as immutable bits (say, $c_1, \ldots, c_k$) and its m outputs as mutable bits (say, $c_{k+1}, \ldots, c_{k+m}$). We then use these objects in a goal sentence of the form

$$\bigwedge_{1 \leq i \leq m} [\, high(c_{k+i}) \leftrightarrow f_i(c_1, \ldots, c_k) \,],$$

where each $f_i(c_1, \ldots, c_k)$ is a boolean combination of atoms $high(c)$ for $c \in \{c_1, \ldots, c_k\}$. Specifying truth values for any of the input or output bits in the initial state would merely eliminate the respective bits from their respective roles, which defeats the purpose of introducing them. Thus, it is an inherent property of the Adder domain to have small initial states and very large goal formulas.

Note that, in comparison to Cube, Adder allows us to harness the relative expressiveness of our formalism. Every non-trivial logic circuit requires intermediate bits to connect one gate's outputs to another's inputs, and since determining the minimally sufficient number of such bits is a part of solving the planning problem itself, it is impossible to anticipate and hard-code the intermediate bits into the initial state description. To circumvent this, as mentioned above, the authors of the original domain allow for the reuse of non-constant bits in the role of intermediate bits. In our formulation, it suffices to assert that there is an infinite supply of non-constant bits:

$$\forall(\textstyle\bigwedge_{c \in \mathtt{const}(\mathcal{K})} x \neq c \rightarrow \neg constant(x)).$$

Given this initial state axiom, we can now properly describe the immutability of bits using the fluent $constant(x, s)$ and not worry about running out of intermediate bits before a large enough circuit can be built.

The domain's actions $and(x, y, z), or(x, y, z), xor(x, y, z), not(x, z)$ represent boolean operations with their usual meanings such that, for each, the last argument is the output bit, and the rest are input bits. Executing an action is to be understood as the addition of the corresponding logical gate to the circuit. Every action binds its output bit's truth value to the values of its input bits and marks the output bit as immutable, which cannot be reversed by subsequent actions. The complete axiomatization of the

domain is as follows.

$$
\begin{aligned}
&Poss(and(x,y,z),s) \leftrightarrow \neg constant(z) \wedge x \neq y \wedge y \neq z \wedge x \neq z,\\
&Poss(or(x,y,z),s) \leftrightarrow \neg constant(z) \wedge x \neq y \wedge y \neq z \wedge x \neq z,\\
&Poss(xor(x,y,z),s) \leftrightarrow \neg constant(z) \wedge x \neq y \wedge y \neq z \wedge x \neq z,\\
&Poss(not(x,z),s) \leftrightarrow \neg constant(z) \wedge x \neq z,
\end{aligned}
$$

$$
\begin{aligned}
high(x,do(a,s)) \leftrightarrow\ & \exists z_1 \exists z_2 (a = and(z_1,z_2,x) \wedge high(z_1,s) \wedge high(z_2,s)) \vee\\
& \exists z_1 \exists z_2 (a = or(z_1,z_2,x) \wedge (high(z_1,s) \vee high(z_2,s))) \vee\\
& \exists z_1 \exists z_2 (a = xor(z_1,z_2,x) \wedge\\
& \qquad (high(z_1,s) \wedge \neg high(z_2,s) \vee \neg high(z_1,s) \wedge high(z_2,s))) \vee\\
& \exists z_1 (a = not(z_1,x) \wedge \neg high(z_1,s)) \vee\\
& high(x,s) \wedge\\
& \neg(\exists z_1 \exists z_2 (a = and(z_1,z_2,x) \wedge \neg(high(z_1,s) \wedge high(z_2,s))) \vee\\
& \quad \exists z_1 \exists z_2 (a = or(z_1,z_2,x) \wedge \neg high(z_1,s) \wedge \neg high(z_2,s)) \vee\\
& \quad \exists z_1 \exists z_2 (a = xor(z_1,z_2,x) \wedge\\
& \qquad (high(z_1,s) \wedge high(z_2,s) \vee \neg high(z_1,s) \wedge \neg high(z_2,s))) \vee\\
& \quad \exists z_1 (a = not(z_1,x) \wedge high(z_1,s))),
\end{aligned}
$$

$$
\begin{aligned}
constant(x,do(a,s)) \leftrightarrow\ & \exists z_1 \exists z_2 (a = and(z_1,z_2,x)) \vee \exists z_1 \exists z_2 (a = or(z_1,z_2,x)) \vee\\
& \exists z_1 \exists z_2 (a = xor(z_1,z_2,x)) \vee \exists z_1 (a = not(z_1,x)) \vee constant(x,s).
\end{aligned}
$$

Experimental data

It is a consequence of the Adder domain's small initial states and very large goal formulas that, in test runs on Adder instances, a greater portion of the available computational power (compared to Cube) is spent on deciding propositional entailments, as opposed to exploring the search space.

Moreover, compared to Cube, the search space in Adder is immense owing to the number and arity of the action symbols together with very liberal preconditions. For example, the instance `adder2.pl` starts with 5 named bits; thus, the first action of the plan is selected out of 684 ground actions, 130 of which are possible, and, as the plan grows and new intermediate bits are introduced, this number grows further for each subsequent step of the plan. Even disregarding the introduction of new bits, the synthesis of a 5-gate circuit in this instance involves a search among 37 billion different paths. It would be possible to cut down on the search tree by a large factor by reformulating the domain in terms of a single action $nand(x,y,z)$ or a single action $nor(x,y,z)$. However, this domain was intentionally designed to require a very large search. We left it as is to allow for a direct comparison to the original Adder.

The comparison makes it obvious that in cases such as this, our naive implementation is inadequate.

Table 3.2: Adder test runs. Run times were limited to 1000 seconds of CPU time.

Instance	Number of Bits In + Out = Tot.	Plan length	CPU time (s) blind	heuristic
adder0.pl	2 + 1 = 3	1	0.05	0.04
adder1.pl	2 + 2 = 4	2	3.33	2.51
adder2.pl	4 + 1 = 5	3	-	797.09

This is due to the fact that the heuristic rules are rather trivial. We leave implementing smarter heuristics to future work. Table 3.2 presents the experimental data on three instances of Adder. A brief description of each follows.

`adder0.pl` implements a 1-bit adder which outputs the least significant digit of the sum. The minimal circuit that computes this consists of a single XOR gate. This instance does not require intermediate bits, although they are still considered in the search.

`adder1.pl` implements a complete "Half-adder", i.e. a 1-bit adder that computes both the sum and the carry of the two inputs; it thus has two outputs. The minimal circuit consists of two gates and also does not require intermediate bits.

`adder2.pl` partially implements a 2-bit adder. It takes two 2-bit numbers and outputs the second bit of the sum. The minimal circuit consists of three gates which are connected using intermediate bits. Although the computation is very slow, this instance finally showcases the expressiveness of our formalism: the first plan that is found is `and(c1, c3, c7), xor(c7, c2, c8), xor(c8, c4, c6)`, where the bits `c7` and `c8` are new names, absent from both the initial KB and the goal formula.

As with Cube, we introduced declarative heuristic rules to speed up the search, but it did not make as significant a difference. The heuristic rules amount to two points: first, do not add a gate if a gate of the same kind has already been created for the same inputs, and, second, introduce new intermediate bits only as outputs, and never as inputs, of new gates.

3.3.3 Blocks World

We use a rendition of the classic Blocks World domain based on the one which appeared in [26] and was used in Example 2.3.1 above. It is a conformant, local-effect version of Reiter's axiomatization from [12]. As usual, the world consists of an arrangement of blocks sitting on a table. The blocks can be moved around with the objective of achieving a desired arrangement. There are no static predicates. The fluents are $clear(x, s)$, $on(x, y, s)$, and $ontable(x, s)$; refer to Example 2.2.1 for a detailed explanation.

The actions are $move(x, y, z)$, $movetotable(x, y)$, and $movefromtable(x, z)$. As in Example 2.3.1, to localize the effects, the actions mention all objects that they affect. In all actions, the first argument is the block being moved. In $move(x, y, z)$, the second argument is the block underneath the one being moved, and the last one is the destination block. In $movetotable(x, y)$, the second argument is, likewise, the block just underneath the one being moved, and the third argument is absent since the destination

of the motion is the table. In $movefromtable(x, z)$, the second argument is the destination, and the intermediate argument is absent since the motion starts from the table.

As before, with a local-effect axiomatization comes the penalty of a very large search space. This is exacerbated by the fact that, in order to make the domain compatible with conformant planning, we made the preconditions more liberal by moving some of the predicate conditions to the context conditions of the successor state axioms. Hence, many more actions are possible, but they have no effect unless their context conditions are satisfied.

$$Poss(move(x, y, z), s) \leftrightarrow x \neq y \wedge y \neq z \wedge x \neq z \wedge clear(x, s) \wedge \neg ontable(x, s),$$
$$Poss(movetotable(x, y), s) \leftrightarrow x \neq y \wedge clear(x, s) \wedge \neg ontable(x, s),$$
$$Poss(movefromtable(x, z), s) \leftrightarrow x \neq z \wedge clear(x, s) \wedge ontable(x, s).$$

$$\begin{aligned} clear(x, do(a, s)) \leftrightarrow\ & \exists z_1 \exists z_2 (a = move(z_1, x, z_2) \wedge on(z_1, x, s) \wedge clear(z_2, s)) \\ & \vee \exists z_1 (a = movetotable(z_1, x) \wedge on(z_1, x, s)) \\ & \vee clear(x, s) \\ & \wedge \neg \exists z_1 \exists z_2 (a = move(z_1, z_2, x) \wedge on(z_1, z_2, s) \wedge clear(x, s)) \\ & \wedge \neg \exists z_1 (a = movefromtable(z_1, x) \wedge clear(x, s)), \end{aligned}$$

$$\begin{aligned} on(x, y, do(a, s)) \leftrightarrow\ & \exists z_1 (a = move(x, z_1, y) \wedge on(x, z_1, s) \wedge clear(y, s)) \\ & \vee (a = movefromtable(x, y) \wedge clear(y, s)) \\ & \vee on(x, y, s) \\ & \wedge \neg \exists z_1 (a = move(x, y, z_1) \wedge on(x, y, s) \wedge clear(z_1, s)) \\ & \wedge \neg \exists z_1 (a = movetotable(x, y) \wedge on(x, y, s)), \end{aligned}$$

$$\begin{aligned} ontable(x, do(a, s)) \leftrightarrow\ & (a = movetotable(x, y) \wedge on(x, y, s)) \\ & \vee ontable(x, s) \wedge \neg (a = movefromtable(x, y) \wedge clear(y, s)). \end{aligned}$$

The rather unnatural decision to move some of the preconditions to context effects is motivated by the fact that otherwise it would be impossible to solve a planning problem with disjunctive knowledge about the position of a cube, e.g. like in the instance that appears in [12], where the block `a` is known to be either on block `b` or on block `e`. In a straightforward local-effect axiomatization, we would not be able to move block `a` at all, since neither the precondition for $move(a, b, e)$ nor that for $move(a, e, b)$ would be satisfied, as both require an unambiguous knowledge about $on(a, b, s)$ or $on(a, e, s)$, respectively. In a global effect axiomatization, this problem does not arise.

Surprisingly, in one aspect, the language of our axiomatization turned out to be less expressive then the one used by Finzi *et al.* [12]. Whereas the authors of [12] freely use both existential and universal

quantifiers in the initial state, which is made possible by the closed domain, we are limited to only universal quantifiers. Consequently, we cannot properly express the connection between the meanings of $clear(x, s)$ and $on(x, y, s)$ (and, similarly, between $ontable(x, s)$ and $on(x, y, s)$). In particular, we are able to express one half of the exclusive disjunction as a $\forall$-clause $\forall(\neg clear(x, s) \vee \neg on(y, x, s))$, but the other half requires existential quantification: $\forall(\exists on(y, x, s) \vee clear(x, s))$. Therefore, despite the trick with context conditions, we are still unable to correctly express the instance considered in [12].

Experimental data

The experimental data appears in Table 3.3. A brief description of the instance follows.

`bw0.pl` is a stack of two blocks; the top block needs to be moved to the table. With no uncertainty in the initial state, the problem is easily solved by the planner. `bw1.pl` is the same but the blocks need to be swapped; the problem is easily solved.

`bw2.pl` states that all (unnamed) blocks are clear and not on top of anything but the table. The goal is to have a block which is not on table. `bw3.pl` is the same but an additional block must be on top of the first one. Both are easily solved.

`bw4.pl` contains disjunctive information: block `c3` is either on block `c1` or on block `c2`. The missing connection between $on(y, x, s)$ and $clear(x, s)$ is expressed by the axiom $\forall(on(c_3, x, s) \vee clear(x, s))$. Note that this does not properly replace the meaning mentioned above; in fact, this axiom asserts that `c3` is on top of *every* block which is not clear. Thus, we cannot introduce any unclear blocks into the instance without violating the basic rules of Blocks World. The goal to have `c3` on `c1` is successfully solved by a single step plan.

`bw5.pl` is the same as `bw4.pl` except it requires an additional step to to put `c3` on the table. The algorithm fails to find a plan due to a stack overflow.

All attempts to solve problems that require more than two steps of progression were fruitless. The reason behind this was found to be the very fast growth of the progression. It was also observed that the progressed knowledge bases were of low quality; i.e. very redundant. This is in agreement with our naive implementation of forgetting using propositional resolution. A mere removal of contradictory and tautological clauses yielded a 15-fold improvement in memory usage, producing the results in Table 3.3.

We did not implement a heuristics for Blocks World for the above reason.

3.4 Discussion

Theorem 10 summarizes the theoretical findings of this thesis. The implementation and its experimental evaluation demonstrate practical use of those findings.

One of the issues that makes our results difficult to compare to those of other approaches is the fact that our implementation is an unoptimized proof of concept. With this caveat, we were able to successfully demonstrate that our approach works well for simple closed-domain instances such as Cube, and does not incur significant costs on open-domain instances such as Adder and Blocks World unless the precondition axioms are made to be extremely liberal. We conjecture that the problem with very

Table 3.3: Blocks World test runs. Memory overflow was the limiting factor.

Instance	Number of Blocks	Plan length	CPU time (s)
bw0.pl	2	1	0.03
bw1.pl	2	2	2.29
bw2.pl	0	1	0.02
bw3.pl	0	2	0.62
bw4.pl	3	1	10.83
bw5.pl	3	2	-

large search spaces can be subdued by a slight modification to the rule `legal_move/3` in conjunction with powerful declarative heuristics, similar to those we used in Cube and Finzi *et al.* used for Blocks World in [12].

The inefficient use of memory by progression stems from the selected forgetting mechanism. As remarked above, forgetting via propositional resolution results in low-quality, redundant knowledge, which becomes a serious obstacle when the domain contains many fluents with many arguments, as Blocks World does. To overcome this issue, a better forgetting mechanism is needed. Alternatively, knowledge base recompilation into a more efficient representation, e.g. prime implicates, on every step of progression would improve memory usage at the cost of CPU time.

An illustration of the lack of expressive power of our approach is the inherent inability of $proper^+$ knowledge bases to express the existence of unnamed objects except by using universal quantification. That is, we can assert that "all objects which are not c_1 are on table", but we cannot assert that "there is at least one object on table which is not c_1". This substantially limits our ability to describe relationships between unnamed objects and their properties, e.g. the relationship between *clear* and *on*. In practice, this limitation can be in part avoided by simply giving names to unnamed objects which are desired to possess unique properties. However, as we demonstrated with Blocks World above, there are essential rules in some domains that simply cannot be expressed. This issue calls for extending $proper^+$ as a part of future work.

Another seeming limitation of our approach is the lack of quantifiers in queries. This is not quite so. Observe that we do allow a hint of existential quantification in goal formulas. Recall that, in instance goals, it is possible to use normal constants that do not appear in the initial KB. For example, consider a Blocks World KB $\{\forall(clear(x, S_0)), \forall(x = c_1 \rightarrow ontable(x, S_0)\}$ and a goal $on(c_2, c_1, s)$. Since there is no information in the KB about c_2 specifically, the goal formula is essentially the same as $\exists x(x, c_1, s)$. In such cases, our planner effectively solves the problem of existence of an unnamed object with the required properties and assigns to it the name suggested by the goal formula. We conjecture that the ability for unbounded quantification in queries is inherent in our current formalism and can be introduced into the prototype with little effort.

Chapter 4

Conclusion

4.1 Contributions

In this thesis, we reconsidered the problem of planning with incomplete knowledge in the context of situation calculus semantics and initial states formulated in an expressive fragment of first-order logic. We used the notion of finite representations for query answers and $proper^+$ knowledge bases to arrive at a decidable planning algorithm. We employed a recently discovered approach to local-effect progression of $proper^+$ knowledge bases and an off-the-shelf SAT solver to implement a prototype planner.

Specifically, in Chapter 2, we gave an overview of classical planning, including its formal statement, limitations, and a brief account of the existing approaches to it. Next, we gave an overview of the existing conformant extensions of classical planning, their relative successes and limitations, including the ubiquitous limited expressiveness and difficulties with finding good computational heuristics. We remarked on the utility of modern SAT solvers for classical and extended classical planning. Then, we presented a brief history of reasoning about action and outlined the trend for increasing expressiveness and striving for clear semantics.

Later in Chapter 2, we introduced the language of situation calculus and the notion of basic action theories, including the underlying motivation, syntax, semantics, and the associated essential reasoning problems. We described two efficient approaches to the projection problem—regression and progression—and motivated our choice for using the latter. Then, we thoroughly described the semantics of planning in the context of situation calculus and remarked on the fact that it generalizes the approaches based on STRIPS. We then described the basic considerations for solving thus defined planning problems and pointed out the omnipresence of the projection problem. Continuing with situation calculus, we gave an account of an existing planning mechanism implementing regression; we pointed out that it sacrifices most of the expressiveness, and observed yet another example of a beneficial use of a SAT solver for planning. Next, we formally defined progression of general first-order action theories and for action theories with local effects specifically. Delving into the latter, we described how progression can be computed and gave examples to demonstrate major steps of such computation.

While addressing the problem of computational feasibility, we overviewed the fragment of first-order logic corresponding to the *proper*$^+$ normal form and presented the associated recent results which motivate our choice of *proper*$^+$ as the underlying language for our planning algorithm. Specifically, *proper*$^+$ features reasonable expressivity while providing significant computational benefits for local-effect progression and query answering. We traced the development of several evaluation procedures for *proper*$^+$ knowledge bases, introducing the underlying theoretical results, and concluded with a discussion of one of the most recent advances in the area, the logic of limited belief.

We continued with a detailed exposition of an existing method for computing the progression in well-formed basic action theories and provided proof outlines for the most important underlying results. Finally, we presented recent findings regarding the progression of well-formed BATs using finite propositional representations of *proper*$^+$ knowledge bases and connected these developments with previous results. In Lemma 5, we reformulated and proved a hidden result from [10] and used it in conjunction with a minor original result (Lemma 7) and an original yet inspired by existing work Lemma 8 to compile a set of techniques which pave the way for developing a sound and complete algorithm for conformant planning.

In Chapter 3, we developed, from general principles, a methodology for finding a solution to a planning problem defined as a logical entailment of a goal formula from a basic action theory. By sequentially transforming the goal formula with the help of the foundational axioms of situation calculus, we were able to break the entailment problem into a set of smaller problems. Then, by using our results about finite representations of query answers, we showed that it is possible to bypass a search for a plan in an infinite search space while still being able to find all correct plans. We used these results to formulate an algorithm that implements a sound and complete conformant planner.

We then described the design of our simple prototype planner based on the proposed algorithm and provided an experimental evaluation of its performance. Unsurprisingly, the prototype did not scale up well, but it nevertheless confirmed our findings. We discovered a significant deficiency in the expressiveness of the *proper*$^+$ normal form which renders our planner unable to work with a complete definition of Blocks World.

4.2 Future Work

The prototype successfully demonstrated the validity of the theoretical findings. However, as evidenced by the experimental evaluation, the prototype requires major optimization in order to be practical. Also, we discovered certain limitations of the available expressiveness, the resolution of which would be highly beneficial. For future work, we propose to implement:

- Efficient representation of knowledge with recompilation on each step of progression. Good candidates would possibly be prime implicates or minimal CNF from [43]. Since the present implementation features polynomial growth of progression, this would make a considerable practical difference, allowing the planner to find longer plans.

- An improved integration of declarative heuristics with the mechanism for action grounding. As discussed in relation to the Adder domain, a naive traversal of the search space results in spectacular failure on certain domains. A thoughtful implementation of heuristics such as the one found in [12] would facilitate the search immeasurably.

- An extension of queries to allow some form of existential and universal quantification. This would noticeably increase the power of goal formulas and precondition axioms.

- An extension to the expressiveness of $proper^+$ knowledge bases, possibly with the help of Skolem constants as was done in [9] for *proper*. Such a development would require a thorough study of the associated properties of progression.

Appendix 1

Domains and Instances

1.1 Cube

1.1.1 Domain

```
fluents([xpos, ypos, zpos]).

ap(xmove(X,Y), xabove(X,Y) v xabove(Y,X)).
ap(ymove(X,Y), xabove(X,Y) v xabove(Y,X)).
ap(zmove(X,Y), xabove(X,Y) v xabove(Y,X)).

ssa(xpos(_), xmove(C1,C2),
[[[C2], xpos(C1) ]],
[[[C1], xpos(C1) ]]).

ssa(xpos(_), ymove(C1,C2),
[],
[]).

ssa(xpos(_), zmove(C1,C2),
[],
[]).

ssa(ypos(_), ymove(C1,C2),
[[[C2], ypos(C1) ]],
[[[C1], ypos(C1) ]]).

ssa(ypos(_), xmove(C1,C2),
```

```
[],
[]).

ssa(ypos(_), zmove(C1,C2),
[],
[]).

ssa(zpos(_), zmove(C1,C2),
[[[C2], zpos(C1) ]],
[[[C1], zpos(C1) ]]).

ssa(zpos(_), xmove(C1,C2),
[],
[]).

ssa(zpos(_), ymove(C1,C2),
[],
[]).

useless(A, History, _) :- member(A,History).

useless(xmove(X,Y), History, _) :- member(xmove(Y,X),History).
useless(ymove(X,Y), History, _) :- member(ymove(Y,X),History).
useless(zmove(X,Y), History, _) :- member(zmove(Y,X),History).

useless(xmove(A,B), History, state(KB, _, _)) :- goal(X ^ _ ^ _), entails(KB,X).
useless(ymove(A,B), History, state(KB, _, _)) :- goal(_ ^ Y ^ _), entails(KB,Y).
useless(zmove(A,B), History, state(KB, _, _)) :- goal(_ ^ _ ^ Z), entails(KB,Z).

useless(ymove(_,_), _, state(KB, _, _)) :- goal(X ^ _ ^ _), not entails(KB,X).
useless(zmove(_,_), _, state(KB, _, _)) :- goal(X ^ _ ^ _), not entails(KB,X).
useless(zmove(_,_), _, state(KB, _, _)) :- goal(_ ^ Y ^ _), not entails(KB,Y).
```

1.1.2 Instances

cube2-1.pl

```
% geometry
fclause(x1:c2 ^ x2:c1 -> xabove(x1,x2)).
fclause(x1:c2 ^ x2:c1 -> yabove(x1,x2)).
fclause(x1:c2 ^ x2:c1 -> zabove(x1,x2)).
```

```
% initial position
fclause(x1:c1 ^ x2:c2 -> xpos(x1) v xpos(x2)).
fclause(x1:c1 ^ x2:c2 -> ypos(x1) v ypos(x2)).
fclause(x1:c1 ^ x2:c2 -> zpos(x1) v zpos(x2)).

% goal condition
goal(xpos(c1) ^ ypos(c1) ^ zpos(c1)).
negatedListGoal(_) :- fail.
```

cube3-1.pl

```
% geometry
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> xabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> yabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> zabove(x1,x2) ).

% initial position
fclause(x1:c1 ^ x2:c2 -> xpos(x1) v xpos(x2)).
fclause(x1:c2 ^ x2:c3 -> ypos(x1) v ypos(x2)).
fclause(x1:c2 -> zpos(x1)).

% goal condition
goal(xpos(c1) ^ ypos(c1) ^ zpos(c1)).
negatedListGoal(_) :- fail.
```

cube3-2.pl

```
% geometry
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> xabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> yabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> zabove(x1,x2) ).

% initial position
fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> xpos(x1) v xpos(x2) v xpos(x3)).
fclause(x1:c2 -> ypos(x1)).
fclause(x1:c3 -> zpos(x1)).

% goal condition
goal(xpos(c1) ^ ypos(c1) ^ zpos(c1)).
negatedListGoal(_) :- fail.
```

cube3-3.pl

```
% geometry
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> xabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> yabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> zabove(x1,x2) ).

% initial position
fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> xpos(x1) v xpos(x2) v xpos(x3)).
fclause(x1:c1 ^ x2:c2 -> ypos(x1) v ypos(x2)).
fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> zpos(x1) v zpos(x2) v zpos(x3)).

% goal condition
goal(xpos(c1) ^ ypos(c1) ^ zpos(c1)).
negatedListGoal(_) :- fail.
```

cube3-4.pl

```
% geometry
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> xabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> yabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> zabove(x1,x2) ).

% initial position
fclause(x1:c1 ^ x2:c3 -> xpos(x1) v xpos(x2)).
fclause(x1:c1 ^ x2:c3 -> ypos(x1) v ypos(x2)).
fclause(x1:c2 ^ x2:c3 -> zpos(x1) v zpos(x2)).

% goal condition
goal(xpos(c1) ^ ypos(c1) ^ zpos(c1)).
negatedListGoal(_) :- fail.
```

cube3-5.pl

```
% geometry
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> xabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> yabove(x1,x2) ).
fclause((x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> zabove(x1,x2) ).

% initial position
fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> xpos(x1) v xpos(x2) v xpos(x3)).
```

```
fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> ypos(x1) v ypos(x2) v ypos(x3)).
fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> zpos(x1) v zpos(x2) v zpos(x3)).

% goal condition
goal(xpos(c1) ^ ypos(c1) ^ zpos(c1)).
negatedListGoal(_) :- fail.
```

cube4-1.pl

```
% geometry
fclause((x1:c4 ^ x2:c3)v(x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> xabove(x1,x2) ).
fclause((x1:c4 ^ x2:c3)v(x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> yabove(x1,x2) ).
fclause((x1:c4 ^ x2:c3)v(x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> zabove(x1,x2) ).

% initial position
fclause(x1:c1 ^ x2:c2 ^ x3:c3 ^ x4:c4 -> xpos(x1) v xpos(x2) v xpos(x3) v xpos(x4)).
fclause(x1:c1 ^ x2:c2 ^ x3:c3 ^ x4:c4 -> ypos(x1) v ypos(x2) v ypos(x3) v ypos(x4)).
fclause(x1:c1 ^ x2:c2 ^ x3:c3 ^ x4:c4 -> zpos(x1) v zpos(x2) v zpos(x3) v zpos(x4)).

% goal condition
goal(xpos(c1) ^ ypos(c1) ^ zpos(c1)).
negatedListGoal(_) :- fail.
```

cube5-1.pl

```
% geometry
fclause((x1:c5 ^ x2:c4)v(x1:c4 ^ x2:c3)v(x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> xabove(x1,x2)).
fclause((x1:c5 ^ x2:c4)v(x1:c4 ^ x2:c3)v(x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> yabove(x1,x2)).
fclause((x1:c5 ^ x2:c4)v(x1:c4 ^ x2:c3)v(x1:c3 ^ x2:c2)v(x1:c2 ^ x2:c1) -> zabove(x1,x2)).

% initial position
fclause(x1:c1 -> xpos(x1)).
fclause(x1:c1 ^ x2:c2 ^ x3:c3 ^ x4:c4 ^ x5:c5
            -> ypos(x1) v ypos(x2) v ypos(x3) v ypos(x4) v ypos(x5)).
fclause(x1:c2 -> zpos(x1)).

% goal condition
goal(xpos(c1) ^ ypos(c1) ^ zpos(c1)).
negatedListGoal(_) :- fail.
```

1.2 Adder

1.2.1 Domain

```
% Adder

fluents([high, constant]).

% Action precondition axioms
ap( and_(X,Y,Z), neg constant(Z) ^ neg X:Y ^ neg X:Z ^ neg Y:Z ).
ap( or_(X,Y,Z), neg constant(Z) ^ neg X:Y ^ neg X:Z ^ neg Y:Z ).
ap( xor_(X,Y,Z), neg constant(Z) ^ neg X:Y ^ neg X:Z ^ neg Y:Z ).
ap( not_(X,Z), neg constant(Z) ^ neg X:Z ).

% Transformed successor state axioms
ssa( high(_), and_(X,Y,Z) ,
    [[[Z], high(X) ^ high(Y) ]],
    [[[Z], neg high(X) v neg high(Y) ]]).

ssa( high(_), or_(X,Y,Z) ,
    [[[Z], high(X) v high(Y) ]],
    [[[Z], neg high(X) ^ neg high(Y) ]]).

ssa( high(_), xor_(X,Y,Z) ,
    [[[Z], high(X) ^ neg high(Y) v neg high(X) ^ high(Y) ]],
    [[[Z], high(X) ^ high(Y) v neg high(X) ^ neg high(Y) ]]).

ssa( high(_), not_(X,Z) ,
    [[[Z], neg high(X) ]],
    [[[Z], high(X) ]]).

ssa( constant(_), and_(X,Y,Z),
    [[[Z], tRUE]],
    []).

ssa( constant(_), or_(X,Y,Z),
    [[[Z], tRUE]],
    []).

ssa( constant(_), xor_(X,Y,Z),
```

```
    [[[Z], tRUE]],
    []).

ssa( constant(_), not_(X,Z),
    [[[Z], tRUE]],
    []).

% Do not introduce redundant gates
useless(and_(X,Y,_), History, _) :- member(and_(X,Y,_), History).
useless(and_(X,Y,_), History, _) :- member(and_(Y,X,_), History).
useless(or_(X,Y,_), History, _) :- member(or_(X,Y,_), History).
useless(or_(X,Y,_), History, _) :- member(or_(Y,X,_), History).
useless(xor_(X,Y,_), History, _) :- member(xor_(X,Y,_), History).
useless(xor_(X,Y,_), History, _) :- member(xor_(Y,X,_), History).
useless(not_(X,_), History, _) :- member(not_(X,_), History).

% Do not introduce new (unconnected) inputs
useless(and_(X,_,_), History, state(_, Const, _)) :- not member(X, Const).
useless(and_(_,X,_), History, state(_, Const, _)) :- not member(X, Const).
useless(or_(X,_,_), History, state(_, Const, _)) :- not member(X, Const).
useless(or_(_,X,_), History, state(_, Const, _)) :- not member(X, Const).
useless(xor_(X,_,_), History, state(_, Const, _)) :- not member(X, Const).
useless(xor_(_,X,_), History, state(_, Const, _)) :- not member(X, Const).
useless(not_(X,_), History, state(_, Const, _)) :- not member(X, Const).
```

1.2.2 Instances

adder0.pl

```
%       c2  c1  <- constant
%     + c4  c3  <- constant
% ------------
%   c7  c6  c5  <- neg constant

% 1-bit adder outputting the least significant digit of the sum

fclause(neg x1:c1 ^ neg x1:c3 -> neg constant(x1)).
fclause(x1:c1 -> constant(x1)).
fclause(x1:c3 -> constant(x1)).

goal( (neg high(c5) v (neg high(c1) ^ high(c3)) v (high(c1) ^ neg high(c3)))
```

```
    ^ (high(c5) v (high(c1) v neg high(c3)) ^ (neg high(c1) v high(c3))) ).

negatedListGoal([[high(c5), high(c1), high(c3)],
                 [high(c5), neg high(c1), neg high(c3)],
                 [neg high(c5), high(c1), neg high(c3)],
                 [neg high(c5), neg high(c1), high(c3)]]).
```

adder1.pl

```
% 1-bit adder outputting the entire sum ("Half-adder")

fclause(neg x1:c1 ^ neg x1:c3 -> neg constant(x1)).
fclause(x1:c1 -> constant(x1)).
fclause(x1:c3 -> constant(x1)).


goal( (neg high(c5) v (neg high(c1) ^ high(c3)) v (high(c1) ^ neg high(c3)))
    ^ (high(c5) v (high(c1) v neg high(c3)) ^ (neg high(c1) v high(c3)))
    ^ (neg high(c6) v (high(c1) ^ high(c3))) ^ (neg (high(c1) ^ high(c3)) v high(c6))).

negatedListGoal([[high(c1), high(c3), high(c5), high(c6)],
                 [high(c1), neg high(c3), neg high(c5), high(c6)],
                 [neg high(c1), high(c3), neg high(c5), high(c6)],
                 [neg high(c1), neg high(c3), high(c5), neg high(c6)]]).
```

adder2.pl

```
% 2-bit adder, outputs second digit of the sum

fclause(neg x1:c1 ^ neg x1:c2 ^ neg x1:c3 ^ neg x1:c4 -> neg constant(x1)).
fclause(x1:c1 -> constant(x1)).
fclause(x1:c2 -> constant(x1)).
fclause(x1:c3 -> constant(x1)).
fclause(x1:c4 -> constant(x1)).

goal( (neg high(c6) v (neg high(c1) v neg high(c3)) ^ neg high(c2) ^ high(c4)
    v (neg high(c1) v neg high(c3)) ^ high(c2) ^ neg high(c4) v high(c1) ^ high(c3)
    ^ neg high(c2) ^ neg high(c4) v high(c1) ^ high(c3) ^ high(c2) ^ high(c4))
    ^ (high(c6) v (((high(c1) ^ high(c3)) v high(c2) v neg high(c4)) ^ ((high(c1)
    ^ high(c3)) v neg high(c2) v high(c4)) ^ neg (high(c1) ^ high(c3) ^ neg high(c2)
    ^ neg high(c4)) ^ neg (high(c1) ^ high(c3) ^ high(c2) ^ high(c4))))).
```

```
negatedListGoal([[high(c6),  high(c1),  high(c2),  high(c4)],
                 [high(c6),  high(c1),  neg high(c2), neg high(c4)],
                 [high(c6),  high(c3),  high(c2),  high(c4)],
                 [high(c6),  high(c3),  neg high(c2), neg high(c4)],
                 [high(c6),  neg high(c1), neg high(c3), high(c2),  neg high(c4)],
                 [high(c6),  neg high(c1), neg high(c3), neg high(c2), high(c4)],
                 [neg high(c6), high(c1),  high(c2),  neg high(c4)],
                 [neg high(c6), high(c1),  neg high(c2), high(c4)],
                 [neg high(c6), high(c3),  high(c2),  neg high(c4)],
                 [neg high(c6), high(c3),  neg high(c2), high(c4)],
                 [neg high(c6), neg high(c1), neg high(c3), high(c2),  high(c4)],
                 [neg high(c6), neg high(c1), neg high(c3), neg high(c2), neg high(c4)]]).
```

1.3 Blocks World

1.3.1 Domain

```
fluents([clear, on, ontable]).

ap( move(X,Y,Z),        neg X:Z ^ neg X:Y ^ neg Y:Z ^ clear(X) ^ neg ontable(X) ).
ap( movetotable(X,Y),   neg X:Y ^ clear(X) ^ neg ontable(X) ).
ap( movefromtable(X,Z), neg X:Z ^ clear(X) ^ ontable(X) ).

ssa( clear(_), move(B1,B2,B3),
    [[[B2], clear(B3) ^ on(B1,B2)]],
    [[[B3], clear(B3) ^ on(B1,B2)]]).

ssa( clear(_), movetotable(B1,B2),
    [[[B2], on(B1,B2)]],
    []).

ssa( clear(_), movefromtable(B1,B2),
    [],
    [[[B2], clear(B2)]]).

ssa( on(_,_), move(B1,B2,B3),
    [[[B1,B3], clear(B3) ^ on(B1,B2)]],
    [[[B1,B2], clear(B3) ^ on(B1,B2)]]).
```

```
ssa( on(_,_), movetotable(B1,B2),
    [],
    [[[B1,B2], on(B1,B2)]]).

ssa( on(_,_), movefromtable(B1,B2),
    [[[B1,B2], clear(B2)]],
    []).

ssa( ontable(_), move(B1,B2,B3),
    [],
    []).

ssa( ontable(_), movetotable(B1,B2),
    [[[B1], on(B1,B2)]],
    []).

ssa( ontable(_), movefromtable(B1,B2),
    [],
    [[[B1], clear(B2)]]).
```

1.3.2 Instances

bw0.pl

```
% c2 is on c1, need to move it to table

fclause(x1:c1 -> ontable(x1)).
fclause(x1:c2 -> neg ontable(x1)).
fclause(x1:c1 -> neg clear(x1)).
fclause(x1:c2 -> clear(x1)).
fclause(x1:c1 ^ x2:c2 -> on(x2,x1)).
fclause(x1:c1 ^ x2:c2 -> neg on(x1,x2)).

goal(ontable(c2)).
negatedListGoal(_) :- fail.
useless(_,_,_) :- fail.
```

bw1.pl

```
% c2 is on c1, need to swap them
```

```
fclause(x1:c1 -> ontable(x1)).
fclause(x1:c2 -> neg ontable(x1)).
fclause(x1:c1 -> neg clear(x1)).
fclause(x1:c2 -> clear(x1)).
fclause(x1:c1 ^ x2:c2 -> on(x2,x1)).
fclause(x1:c1 ^ x2:c2 -> neg on(x1,x2)).

goal(on(c1,c2)).
negatedListGoal(_) :- fail.
useless(_,_,_) :- fail.
```

bw2.pl

```
% All blocks are on table, need to move one off it.

fclause(tRUE -> ontable(x1)).
fclause(tRUE -> clear(x1)).
fclause(tRUE -> neg on(x1,x2)).

goal(neg ontable(c1)).
negatedListGoal(_) :- fail.
useless(_,_,_) :- fail.
```

bw3.pl

```
% All blocks are on table, need to move one off it
% and put another one on it.

fclause(tRUE -> ontable(x1)).
fclause(tRUE -> clear(x1)).
fclause(tRUE -> neg on(x1,x2)).

goal(neg ontable(c1) ^ on(c2,c1)).
negatedListGoal(_) :- fail.
useless(_,_,_) :- fail.
```

bw4.pl

```
% c3 is either on c1 or c2; need it to be on c1.

fclause(x1:c1 v x1:c2 -> ontable(x1)).
```

```
fclause(x1:c3 -> neg ontable(x1)).

fclause(x1:c1 ^ x2:c2 -> neg clear(x1) v neg clear(x2)).
fclause(x1:c1 ^ x2:c2 -> clear(x1) v clear(x2)).
fclause(x1:c3 -> clear(x1)).

fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> on(x3,x1) v on(x3,x2)).
fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> neg on(x3,x1) v neg on(x3,x2)).
fclause(x1:c3 -> neg on(x2,x1)).
fclause(neg x1:c3 ^ x2:c1 -> neg on(x1,x2)).
fclause(neg x1:c3 ^ x2:c2 -> neg on(x1,x2)).

fclause(tRUE -> neg on(x1,x2) v neg clear(x2)).
fclause(x1:c3 -> on(x1,x2) v clear(x2)).

goal(on(c3, c1)).
negatedListGoal(_) :- fail.
useless(_,_,_) :- fail.
```

bw5.pl

```
% c3 is either on c1 or c2; need it to put it on table.

fclause(x1:c1 v x1:c2 -> ontable(x1)).
fclause(x1:c3 -> neg ontable(x1)).

fclause(x1:c1 ^ x2:c2 -> neg clear(x1) v neg clear(x2)).
fclause(x1:c1 ^ x2:c2 -> clear(x1) v clear(x2)).
fclause(x1:c3 -> clear(x1)).

fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> on(x3,x1) v on(x3,x2)).
fclause(x1:c1 ^ x2:c2 ^ x3:c3 -> neg on(x3,x1) v neg on(x3,x2)).
fclause(x1:c3 -> neg on(x2,x1)).
fclause(neg x1:c3 ^ x2:c1 -> neg on(x1,x2)).
fclause(neg x1:c3 ^ x2:c2 -> neg on(x1,x2)).

fclause(tRUE -> neg on(x1,x2) v neg clear(x2)).
fclause(x1:c3 -> on(x1,x2) v clear(x2)).

goal(ontable(c3)).
```

```
negatedListGoal(_) :- fail.
useless(_,_,_) :- fail.
```

References

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI*, volume 9, pages 399–404, 2009.

[2] Piergiorgio Bertoli, Alessandro Cimatti, and Marco Roveri. Heuristic search + symbolic model checking = efficient conformant planning. In *IJCAI*, volume 1, pages 467–472. Citeseer, 2001.

[3] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997.

[4] Blai Bonet and Hector Geffner. Planning with incomplete information as heuristic search in belief space. In *AIPS-2000*, pages 52–61. AAAI Press, 2000.

[5] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1):165–204, 1994.

[6] Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147(1):85–117, 2003.

[7] Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90. ACM, 1977.

[8] Alessandro Cimatti, Marco Roveri, and Piergiorgio Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1):127–206, 2004.

[9] Giuseppe De Giacomo, Yves Lespérance, and Hector J Levesque. Efficient reasoning in proper knowledge bases with unknown individuals. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence*, volume 2, pages 827–832. AAAI Press, 2011.

[10] Yi Fan, Minghui Cai, Naiqi Li, and Yongmei Liu. A first-order interpreter for knowledge-based Golog with sensing based on exact progression and limited reasoning. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[11] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3):189–208, 1972.

[12] Alberto Finzi, Fiora Pirri, and Raymond Reiter. Open world planning in the situation calculus. In *AAAI/IAAI*, pages 754–760, 2000.

[13] Hector Geffner and Blai Bonet. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(1):1–141, 2013.

[14] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[15] Nathanael Hyafil and Fahiem Bacchus. Conformant probabilistic planning via CSPs. In *ICAPS*, volume 98, pages 205–214, 2003.

[16] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[17] Gerhard Lakemeyer and Hector J Levesque. Evaluation-based reasoning with disjunctive information in first-order knowledge bases. In *Principles of Knowledge Representation and Reasoning*, pages 73–81. Citeseer, 2002.

[18] Jérôme Lang, Paolo Liberatore, and Pierre Marquis. Propositional independence. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

[19] Hector J Levesque. What is planning in the presence of sensing? In *Proceedings of the National Conference on Artificial Intelligence*, volume 2, pages 1139–1146, 1996.

[20] Hector J Levesque. A completeness result for reasoning with incomplete first-order knowledge bases. In *Principles of KRR*, pages 14–23. Morgan Kaufmann Publishers, 1998.

[21] Vladimir Lifschitz. On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, 1987.

[22] Fangzhen Lin and Ray Reiter. Forget it! In *Working Notes of AAAI Fall Symposium on Relevance*, pages 154–159, 1994.

[23] Fangzhen Lin and Ray Reiter. How to progress a database. *Artificial Intelligence*, 92(1):131–167, 1997.

[24] Yongmei Liu. *Tractable reasoning in incomplete first-order knowledge bases.* PhD thesis, Toronto, Canada, 2006. AAINR15760.

[25] Yongmei Liu and Gerhard Lakemeyer. On the expressiveness of levesque's normal form. *Journal of Artificial Intelligence Research*, 31:259–272, 2008.

[26] Yongmei Liu and Gerhard Lakemeyer. On first-order definability and computability of progression for local-effect actions and beyond. In *IJCAI*, pages 860–866, 2009.

[27] Yongmei Liu and Hector J Levesque. A tractability result for reasoning with incomplete first-order knowledge bases. In *IJCAI*, pages 83–88, 2003.

[28] Yongmei Liu and Hector J Levesque. Tractable reasoning in first-order knowledge bases with disjunctive information. In *IJCAI*, volume 20, page 639, 2005.

[29] Yongmei Liu and Hector J Levesque. Tractable reasoning with incomplete first-order knowledge in dynamic systems with context-dependent actions. In *IJCAI*, volume 5, pages 522–527. Citeseer, 2005.

[30] John McCarthy and Patrick J Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–504. University Press, 1969.

[31] Drew McDermott. The 1998 AI planning systems competition. *AI magazine*, 21(2):35, 2000.

[32] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL — the planning domain definition language. Technical report, available at http://www.cs.yale.edu/∼dvm, 1998.

[33] Hector Palacios and Hector Geffner. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research*, 35(2):623, 2009.

[34] Edwin P D Pednault. Formulating multi-agent dynamic-world problems in the classical planning framework. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning About Actions and Plans: Proceedings of the 1986 Workshop*, pages 47–82, San Mateo, CA, 1987. Morgan Kaufmann Publishers.

[35] Ronald Peter Andrew Petrick. *A Knowledge-level approach for effective acting, sensing, and planning*. University of Toronto Doctoral dissertation, Toronto, Canada, 2006.

[36] Ronald Peter Andrew Petrick and Fahiem Bacchus. A knowledge-based approach to planning with incomplete information and sensing. In *AIPS*, pages 212–222, 2002.

[37] Fiora Pirri and Ray Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM (JACM)*, 46(3):325–361, 1999.

[38] Raymond Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*, volume 16. MIT press Cambridge, 2001.

[39] Joachim Schimpf and Kish Shen. Eclipse — from LP to CLP. *Theory and Practice of Logic Programming, Special Issue on Prolog Systems 1-2*, 12:127–156, 2011.

[40] David E Smith and Daniel S Weld. Conformant Graphplan. In *AAAI/IAAI*, pages 889–896, 1998.

[41] Mikhail Soutchanski and Wael Yehia. Towards an expressive practical logical action theory. *Turing-100*, 10:307–325, 2012.

[42] Sergio Tessaris. *Questions and answers: reasoning and querying in Description Logic.* University of Manchester Doctoral dissertation, 2001.

[43] Son Thanh To, Tran Cao Son, and Enrico Pontelli. On the use of prime implicates in conformant planning. In *AAAI*, 2010.