Formalizing Unsolvability Certificates for Automated Planning in Lean 4

MSc Thesis *(Afstudeerscriptie)*

written by

Amos Nicodemus

under the supervision of **Dr Gregor Behnke** and **Dr Malvin Gattinger**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
January 9th, 2026	Dr Benno van den Berg (chair)
	Prof Dr Ulle Endriss
	Dr Ronald de Haan
	Dr Gregor Behnke (supervisor)
	Dr Malvin Gattinger (supervisor)

Institute for Logic, Language and Computation

Abstract

The objective of classical planning is to find a sequence of actions which leads from an initial state to a goal. Several solvers for automatically solving these planning problems exist, which either return a plan (such a sequence of actions) or state that no plan exists. In the latter case, we would want to verify that in fact there does not exist a plan, which can be done using certificates of unsolvability. Recently, two certificate systems have been developed, namely inductive certificates [19, 12] and a rule based proof system [18, 12]. The simplest version of an inductive certificate is a set of states that is closed under taking actions and contains the initial state but no goal state. The rule based proof system consists of rules to reason about sets of dead states, where a state is dead if no plan traverses the state.

Eriksson has implemented a validator based on this proof system in C++ [14]. Like any other software, this validator can have bugs, hence it would be useful to have a formally verified validator. The goal of the thesis is to implement a validator in Lean 4 for this proof system.

Contents

Chapter 1

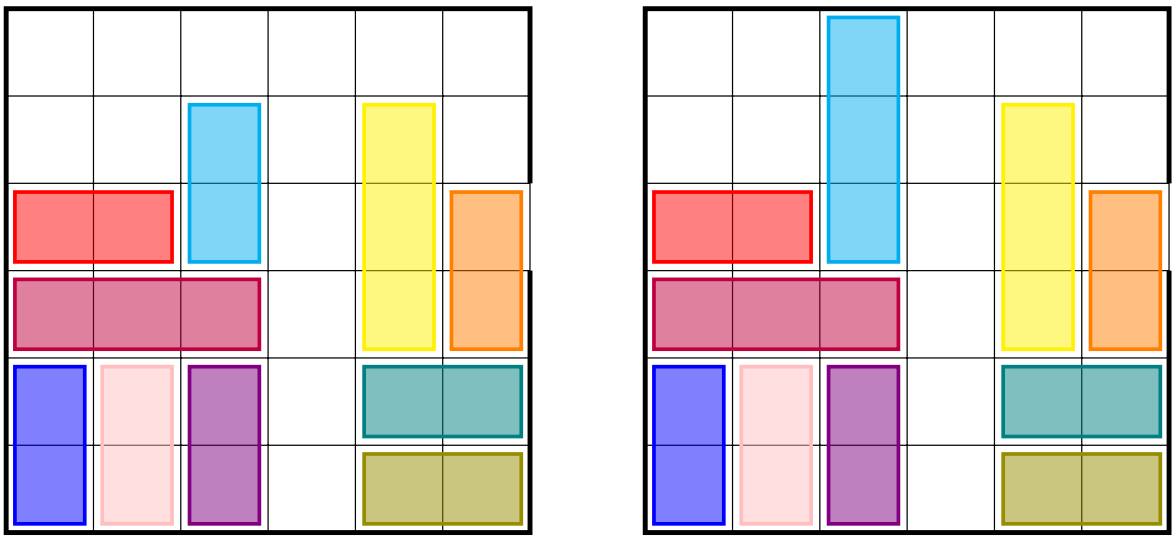
Introduction

In classical planning the objective is to find a sequence of actions which leads from an initial state to a goal. Like many problems in modern computer science, state-of-the-art algorithms to find such a sequence of actions can be very complex. As any other software, solvers (i.e. implementations of these algorithms) can, and often will, contain bugs. The most direct way to avoid any bugs, is to implement the solvers in a formal language like Lean 4 or Isabelle/HOL and formally prove that they are correct. Unfortunately, this approach has some disadvantages: it is time-consuming, state-of-the-art solvers are difficult to formalize, a formalized solver will inevitably be less efficient due to the overhead of formalization, and formalized implementations are less flexible since modifications also require to modify the proof.

A more successful approach is to let solvers output a certificate which can be used by an independent validator to verify that the output is correct [3]. While this approach does not ensure that the solvers don't contain any bugs, it can ensure that the output of the solvers is correct for specific instances, and it can help to discover bugs. Of course the validator programs themselves are not immune to bugs, but the validators are often less complex and therefore easier to formally verify. Eriksson, Röger and Helmert developed a proof system for certifying unsolvability of classical planning problems [18, 12]. The goal of this thesis is to implement a validator for these certificates in Lean 4 and to formally prove its correctness.

Lean is an interactive proof assistant and functional programming language based on dependent type theory [26]. As a proof assistant, Lean 4 provides an interactive environment with powerful tactics. It features *Mathlib*, one of the fastest growing libraries for mathematics covering a wide range of topics [4]. At the same time, Lean 4 is a feature-rich functional programming language which can be compiled into efficient C code. This combination is essential for implementing a formally verified program and makes Lean a well-suited language for this thesis.

In this introductory chapter we first discuss the relevant basic notions in automated planning in Section 1.1. Next will have a look at certificates and certificate systems for automated planning in particular in Section 1.2 and Section 1.3. For the latter, Sections 1.3.1 to 1.3.3 focus on the background needed for thesis, and Sections 1.3.4 to 1.3.6 discuss some related work. Finally, in Section 1.4 we will give an overview of the basic syntax of Lean 4.



(a) Solvable configuration

(b) Unsolvable configuration

Figure 1.1: Two starting configurations for rush hour. In (b) the square in front of the red car will always be blocked by the cyan car, preventing the red car from moving to the exit. The solvable configuration is taken from [31].

1.1 Automated Planning

Informally, a planning problem consists of an initial state, a collection of actions which can be used to alter the current state and a goal that we want to achieve.

Example 1 (Rush Hour)**.** As an example, consider the game of Rush Hour[1]. The starting point of the game is a 6×6 grid with one exit and several cars of length two and trucks of length three placed on the grid, as shown in Figure 1.1. The cars and trucks can move forwards or backwards (but not sideways) if the space in front of or to the back of the car is unoccupied. The goal is to find a sequence of moves which gets the red car out of the grid.

Formally, planning problems are usually described in the STRIPS formalism [20]. The definitions used here are based on those of Eriksson's PhD thesis [12].

Definition 2 (state)**.** Given a set of propositional variables V, a *state* is a subset $s \subseteq V$. We use "*state set*" to denote sets of states.

Informally, a state s contains all the variables that are true, whereas the variables in $V \setminus s$ are false.

Definition 3 (goal state)**.** Given a set of propositional variables V and a *goal description* $G \subseteq V$, a state $s \subseteq V$ is a *goal state* iff $G \subseteq s$.

Definition 4 (action)**.** An *action* is a quadruple $a = \langle \text{pre}(a), \text{add}(a), \text{del}(a), \text{cost}(a) \rangle$, where $\text{pre}(a)$, $\text{add}(a)$ and $\text{del}(a)$ are subsets of a set of propositional variables V and $\text{cost}(a) \in \mathbb{N}_0$. Here $\text{pre}(a)$ is the *precondition* of a, $\text{add}(a)$ are the *adding effects* of a and $\text{del}(a)$ are the *deleting effects* of a. The *cost* of a $\text{cost}(a)$ is not relevant for unsolvability, and therefore we will often omit it.

[1] Rush Hour is manufactured by ThinkFun [9].

Definition 5 (STRIPS planning task)**.** A *STRIPS planning task* is a quadruple $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ where

- V^Π is a finite set of propositional variables,
- A^Π is a finite set of actions for the set V^Π,
- $I^\Pi \subseteq V^\Pi$ is the *initial state*
- $G^\Pi \subseteq V^\Pi$ is the goal description.

The set of all states (that is, the power set of V^Π) is denoted by S^Π and the set of all goal states is denoted by S^Π_G. We write $\|\Pi\|$ for the size of the description of Π.

Example 6 (Rush Hour continued)**.** A possible way to model Rush Hour as a STRIPS planning task is to use the propositional variables

- $\mathsf{at}(C, X, Y)$ for every car/truck C and position (X, Y) (with $1 \leq X \leq 6$ and $1 \leq Y \leq 6$) indicating whether C is at (X, Y),
- $\mathsf{free}(X, Y)$ for every position (X, Y) indicating where (X, Y) is unoccupied, and
- solved indicating whether the red car has left the maze.

For every car C and every position (X, Y) with $3 \leq X \leq 6$ and $1 \leq Y \leq 6$ we have an action $\mathsf{move\_right}(C, X, Y)$ with

- preconditions $\mathsf{at}(C, X-2, Y)$, $\mathsf{at}(C, X-1, Y)$ and $\mathsf{free}(X, Y)$,
- adding effects $\mathsf{at}(C, X, Y)$ and $\mathsf{free}(X-2, Y)$, and
- deleting effects $\mathsf{at}(C, X-2, Y)$ and $\mathsf{free}(X, Y)$.

Similarly, we define actions $\mathsf{move\_left}$, $\mathsf{move\_up}$ and $\mathsf{move\_down}$ for moving the cars in the other directions, and we do the same for all trucks. Finally, we have an action exit with

- preconditions $\mathsf{at}(\mathsf{red\_car}, 5, 4)$ and $\mathsf{at}(\mathsf{red\_car}, 6, 4)$,
- adding effects solved, $\mathsf{free}(5, 4)$ and $\mathsf{free}(6, 4)$, and
- deleting effects $\mathsf{at}(\mathsf{red\_car}, 5, 4)$ and $\mathsf{at}(\mathsf{red\_car}, 6, 4)$.

The initial state contains $\mathsf{at}(C, X, Y)$ iff position (X, Y) is occupied by car/truck C in the initial configuration, it contains $\mathsf{free}(X, Y)$ iff no car/truck occupies position (X, Y) in the initial configuration, and it does not contain solved. The goal is $\{\mathsf{solved}\}$.

This is of course not the only way of modelling Rush Hour, as one could for example drastically reduce the number of variables and actions by specializing the description for a specific initial configuration.

Definition 7 (action application)**.** An action a is *applicable* in a state s iff $\text{pre}(a) \subseteq s$. Assuming that a is applicable in s, applying a to s results in the successor state $s[a] \coloneqq (s \setminus \text{del}(a)) \cup \text{add}(a)$.

A sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ is applicable in s iff a_i is applicable in $s[a_1] \cdots [a_{i-1}]$ for all $1 \leq i \leq n$. Instead of $s[a_1] \cdots [a_n]$ we will usually write $s[\pi]$.

A state s' is called *reachable from* a state s iff there exists a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ such that π is applicable in s and $s' = s[\pi]$. If s' is reachable from I^Π, then we also say that s' is *reachable*.

Definition 8 (plan)**.** Given a planning task Π and a state s, a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$ is a *plan for* s iff π is applicable in s and $s[\pi]$ is a goal state. A plan for I^Π is also called a *plan for* Π or a *plan* (if Π is clear from the context).

Definition 9 (solvable and unsolvable)**.** Let Π be planning task. A state $s \in S^\Pi$ is *solvable* iff a plan for s exists, otherwise s is *unsolvable*. A planning task Π is *solvable* if a plan for Π exists, otherwise Π is *unsolvable*.

Definition 10 (progression)**.** Let Π be a planning task and let $S \subseteq S^\Pi$ be a state set. For an action $a \in A^\Pi$, the *progression* of S with a is defined to be the set $S[a] \coloneqq \{s[a] \mid s \in S, a \text{ is applicable in } s\}$. For a set of actions $A \subseteq A^\Pi$, the progression of S with A is the set $S[A] \coloneqq \bigcup_{a \in A} S[a]$. We also say that $S[A^\Pi]$ is the progression of S.

Definition 11 (regression)**.** Let Π be a planning task and let $S \subseteq S^\Pi$ be a state set. For an action $a \in A^\Pi$, the *regression* of S with a is defined to be the set $[a]S \coloneqq \left\{s' \in S^\Pi \;\middle|\; a \text{ is applicable in } s', s'[a] \in S\right\}$. For a set of actions A, the regression of S with A is the set $[A]S \coloneqq \bigcup_{a \in A} [a]S$. We also say that $[A^\Pi]S$ is the regression of S.

1.2 Certificates

Combinatorial solvers like automated planners can make various claims $P(\mathcal{I}, \vec{x})$ about a problem instance $\mathcal{I}$. For example, the claim $\mathsf{solution}(\mathcal{I}, S)$ states that S is a solution for $\mathcal{I}$, $\mathsf{unsolvable}(\mathcal{I})$ that $\mathcal{I}$ has no solution, and $\mathsf{optimal}(\mathcal{I}, S, c)$ that the solution S with cost c is an optimal solution for $\mathcal{I}$. As motivated in the introduction of this chapter, certificate systems are a useful way to validate that these statements are correct (according to the semantics of interest). On an abstract level, a certificate system consists of three components:

1. a theoretical framework,
2. extensions in the solvers themselves, which enable them to generate the certificates, and
3. validators that can validate the certificates generated by the solvers.

For the theoretical framework we are interested in different properties, which are listed below. While this list is based on certificates of unsolvability and optimality for automated planning [19, 28, 10], they are also relevant for other combinatorial problems.

Definition 12 (certificate criteria)**.** Let $P(\mathcal{I}, \vec{x})$ be a statement about an instance $\mathcal{I}$ for some combinatorial problem, possibly using parameters $\vec{x}$. Then a certificate system for P should have the following properties:

- *Soundness*: If a certificate for the statement $P(\mathcal{I}, \vec{x})$ exists, then $P(\mathcal{I}, \vec{x})$ holds.
- *Completeness*: If $P(\mathcal{I}, \vec{x})$ is true, then a certificate for $P(\mathcal{I}, \vec{x})$ exists.
- *Efficient generation*: The generation of the certificates should incur at most a polynomial overhead for the solver.
- *Efficient verification*: The time complexity of validating a certificate should be polynomial in its size.
- *Generality*: The generation of certificates can be efficiently implemented for a wide range of algorithms, rather than being limited to a specific implementation or a specific method for solving planning problems.

For the purpose of thesis we will focus on implementing a validator (third component) and proving the soundness and completeness properties of the theoretical framework. Although we aim to make this validator efficient, we will not focus on verifying that the certificates are efficiently verifiable, nor will we discuss their generality or how the certificates can be efficiently generated.

1.3 Certificates for Automated Planning

There are three main verification problems related to classical planning, namely verifying that a plan is correct (which implies that the planning problem is solvable), verifying that a planning problem is unsolvable and verifying that a plan is optimal. For verifying plans, the plan itself serves as the obvious certificate [1]. At the time of writing, two formalisms for certifying unsolvability have been proposed, namely inductive certificates [19, 12] and a proof system [18, 12], which will be discussed in Section 1.3.2 and Section 1.3.3. Since both systems work with large sets of states, we first have a look at different formalisms to represent these sets in Section 1.3.1. Section 1.3.4 discusses three certificate systems for optimality and Section 1.3.5 discusses an alternative approach to certification by reducing planning problems to model checking. Finally, in Section 1.3.6 we have a look at the implementations of all these certificate systems.

1.3.1 Representing Sets of States

For every formalism $\mathbf{R}$, we assume a set of propositional variables V, which is ordered by a strict total order $\prec$. Each $\mathbf{R}$-formula (i.e. a formula represented in the formalism $\mathbf{R}$) is associated with a set of variables $\text{vars}(\varphi)$, which contains at least the variables occurring in φ (and possibly also other variables). We write $\|\varphi\|$ for the *size of* (the representation of) φ. In what follows a *literal* is a variable (*positive literal*) or its negation (*negative literal*). A *clause* is a disjunction of literals, a *cube* is a conjunction of literals, a *CNF formula* is a conjunction of clauses and a *DNF formula* is a disjunction of cubes. We are interested in the following four formalisms to represent sets of states, which are discussed in more detail in [12, Section 3.2].

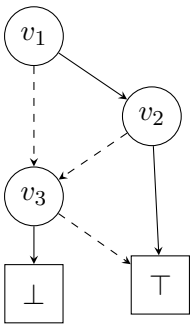


Figure 1.2: A BDD formula representing the formula $(v_1 \wedge v_2) \vee \neg v_3$ with variable ordering $v_1 \prec v_2 \prec v_3$. The dashed arrows correspond with the assignment false, and the solid arrows with the assignment true.

- **BDDs**: A Reduced Ordered Binary Decision Diagram [6] (ROBDD, or BDD for short) uses a rooted, directed, acyclic graph to represent a formula φ. An example is shown in Figure 1.2. The graph has two terminal nodes $\top$ and $\bot$. Every other node n is associated with a variable $v_n \in \text{vars}(\varphi)$, and has two outgoing edges, corresponding to the assignment of "true" and "false" to the variable v_n. ROBDDs are *reduced* in the sense that all redundant nodes (i.e. nodes where both outgoing edges go to the same node) are removed, and all isomorphic subgraphs are merged. They are ordered in the sense that for every edge (n, n') we have $v_n \prec v_{n'}$. To decide whether an assignment $\mathcal{I}$ is a model of φ we traverse the BDD starting at the root by choosing for each node n the outgoing edge corresponding to $\mathcal{I}(v_n)$ until we reach one of the terminal nodes. If this node is $\top$, then $\mathcal{I}$ is a model of φ, and if it is $\bot$ then $\mathcal{I}$ is not a model of φ.
- **Horn formulas**: Horn formulas are CNF-formulas where every clause contains at most one positive literal (there is no limit on the number of negative literals). An example of a Horn formula is $(\neg v_1 \vee \neg v_3 \vee \neg v_4) \wedge (\neg v_1 \vee v_2)$.
- **2CNF formulas**: A 2CNF formula is a CNF-formula where every clause contains at most two literals. The formula $(v_1 \vee \neg v_3) \wedge (v_1 \vee v_2) \wedge \neg v_4$ is an example of a 2CNF formula.
- **MODS (Explicit Enumeration)**: A MOD formula φ is a DNF formula where every cube contains all variables of $\text{vars}(\varphi)$ either positively or negatively (but not both, so each cube is consistent). All cubes of φ are required to be disjoint, i.e. for all cubes c_i and c_j it holds that $c_i \wedge c_j \equiv \bot$. A MODS formula can be seen as an enumeration of partial models over the variables $\text{vars}(\varphi)$. Note that the first paper [19] did not use this formalism yet and that [18] described the less general *explicit sets*, where $\text{vars}(\varphi)$ is always equal to the set of variables of the planning task. An example of a MODS-formula with variables $\{v_1, v_3, v_4\}$ is $(v_1 \wedge v_3 \wedge \neg v_4) \vee (\neg v_1 \wedge \neg v_3 \wedge \neg v_4)$.

For each of the two certificate systems, the authors describe which operations are needed to be able to efficiently verify the certificates, and for each of the formalisms they discuss which operations are supported efficiently. We are interested in operations listed by Definition 13. This list is based on Section 3.2.1 in [12], but we replaced the operations validity (**VA**) and consistency (**CO**) by $\top$**C** and $\bot$**C** since this will lead to fewer cases during validation, and we removed model enumeration (**ME**) and model count (**CT**) because we will not use these operations.

Definition 13 (operations on formulas)**.**

- *Model testing* (**MO**): Given an **R**-formula φ and an assignment $\mathcal{I}$, decide whether $\mathcal{I} \models \varphi$ holds. The assignment $\mathcal{I}$ must assign a value to all variables of φ, i.e. $\text{vars}(\varphi) \subseteq \text{dom}(\mathcal{I})$.
- *Clausal entailment* (**CE**): Given an **R**-formula φ and a clause γ, decide whether $\varphi \models \gamma$ holds.
- *Implicant* (**IM**): Given an **R**-formula φ and a cube δ, decide whether $\delta \models \varphi$ holds.
- *Sentential entailment* (**SE**): Given two **R**-formula φ and ψ, decide whether $\varphi \models \psi$ holds.
- *Bounded conjunction* ($\wedge$**BC**): Given two **R**-formulas φ and ψ, construct an **R**-formula representing $\varphi \wedge \psi$.
- *General conjunction* ($\wedge$**C**): Given **R**-formulas $\varphi_1, \ldots \varphi_n$, construct an **R**-formula representing $\varphi_1 \wedge \cdots \wedge \varphi_n$.
- *Bounded disjunction* ($\vee$**BC**): Given two **R**-formulas φ and ψ, construct an **R**-formula representing $\varphi \vee \psi$.
- *General disjunction* ($\vee$**C**): Given **R**-formulas $\varphi_1, \ldots \varphi_n$, construct an **R**-formula representing $\varphi_1 \vee \cdots \vee \varphi_n$.
- *Negation* ($\neg$**C**): Given an **R**-formula φ, construct an **R**-formula representing $\neg\varphi$.
- *Conjunction of literals* (**CL**): Given a cube δ, construct an **R**-formula representing δ.
- *Bot* ($\bot$**C**): Construct an **R**-formula representing $\bot$.
- *Top* ($\top$**C**): Construct an **R**-formula representing $\top$.
- *Renaming* (**RN**): Given an **R**-formula φ and an injective variable renaming $r : \text{vars}(\varphi) \to V'$, construct the formula $\varphi[r]$, where $\varphi[r]$ is the formula we obtain replacing each variable v in φ by $r(v)$.
- *Renaming consistent with order* (**RN**$_\prec$): Given an **R**-formula φ and a monotone injective variable renaming $r : \text{vars}(\varphi) \to V'$, construct the formula $\varphi[r]$.
- *Transform to CNF* (**toCNF**): Given an **R**-formula φ, construct an equivalent CNF formula.
- *Transform to DNF* (**toDNF**): Given an **R**-formula φ, construct an equivalent DNF formula.

Table 1.1 shows which operations each formalism supports *efficiently*, meaning that the operation can be performed in time polynomial in the size of the given formulas. See [12] for a discussion of each operation for every formalism. For $\bot$**C** and $\top$**C** note that BDD can represent $\top$ and $\bot$ by the graph containing only the terminal node $\bot$ or $\top$. Horn and 2CNF can represent $\bot$ and $\top$ by the conjunction of one empty clause and the empty conjunction respectively, and MODS can represent them by the empty disjunction and the disjunction containing one empty cube respectively.

	BDD	Horn	2CNF	MODS
MO	yes	yes	yes	yes
CE	yes	yes	yes	yes
IM	yes	yes	yes	yes
SE	yes	yes	yes	yes
$\wedge$**BC**	yes	yes	yes	yes
$\wedge$**C**	no	yes	yes	no†
$\vee$**BC**	yes	no	no	no†
$\vee$**C**	no	no	no	no†
$\neg$**C**	yes	no	no	no
CL	yes	yes	yes	yes
$\bot$**C**	yes	yes	yes	yes
$\top$**C**	yes	yes	yes	yes
RN	no	yes	yes	yes
RN$_\prec$	yes	yes	yes	yes
toCNF	no	yes	yes	yes
toDNF	no	no	no	yes

Table 1.1: Overview of operations that are efficiently supported for each of the formalisms. Entry "no†" means that the operations are efficiently supported if all involved formulas have the same set of variables (but not in the general case). Adapted from Table 3.1 in [12].

1.3.2 Inductive Certificates

Intuitively an *inductive set* is a set of states, which once entered cannot be left again.

Definition 14 (inductive set)**.** Let $\Pi = \langle V^\Pi, A^\Pi, I^\Pi, G^\Pi \rangle$ be a STRIPS planning task, then a set of states $S \subseteq S^\Pi$ is *inductive in* Π iff $S[A^\Pi] \subseteq S$. If Π is clear from the context we just say that S is *inductive.*

The notion of an inductive set can be used to express that a planning problem is unsolvable. Indeed, if there is an inductive set which contains the initial state, but no goal state, then it is impossible to leave this inductive set and therefore one can never reach a goal state.

Definition 15 (inductive certificate)**.** Given a STRIPS planning task Π, an *inductive certificate for* a state $s \in S^\Pi$ is a set $S \subseteq S^\Pi$ such that

- $s \in S$,
- $S \cap S^\Pi_G = \emptyset$,
- S is inductive in Π.

An inductive certificate for the initial state I^Π is also called an *inductive certificate* for Π. If a formalism $\mathbf{R}$ is used to represent the set S, then the certificate is also called an *inductive* $\mathbf{R}$*-certificate.*

For any state s, the set of all states reachable from s is inductive. Furthermore, it is the smallest inductive set containing s. Note that s is solvable iff there exists a reachable goal state, hence this immediately implies soundness and completeness for inductive certificates, as defined in Definition 12.

Theorem 16 (soundness and completeness)**.** *Let* Π *be a STRIPS planning task. There exists an inductive certificate for a state* $s \in S^\Pi$ *iff* s *is unsolvable. There exists an inductive certificate for* Π *iff* Π *is unsolvable.*

Proof. See Theorem 4.3 in [12]. □

Theorem 17. *If* $\mathbf{R}$ *efficiently supports the operations* $\boldsymbol{MO}$, $\boldsymbol{CE}$, $\boldsymbol{SE}$, $\wedge\boldsymbol{BC}$, $\boldsymbol{CL}$ *and* $\boldsymbol{RN}_\prec$ *from Definition 13, then inductive* $\mathbf{R}$*-certificates* φ *for* Π *can be verified in polynomial time in* $\|\varphi\|$ *and* $\|\Pi\|$*.*

Proof. See Theorem 4.4 in [12]. □

In particular this implies that inductive certificates using BDD's, Horn formulas, 2CNF formulas and MODS are all efficiently verifiable. While the two results above are nice theoretical properties, it can be computationally challenging to construct and verify inductive certificates. To make constructing certificates easier Eriksson, Röger and Helmert also introduce disjunctive and conjunctive certificates [19, 12]. These certificates consist of a family of state sets $\mathcal{F}$ such that respectively $\bigcup_{S\in\mathcal{F}} S$ and $\bigcap_{S\in\mathcal{F}} S$ are inductive certificates. They then introduce r-disjunctive and r-conjunctive certificates as a special form of disjunctive and conjunctive certificates. Here $r \in \mathbb{N}_0$ is a parameter which ensures that the certificates can be verified in polynomial time when r is fixed.

Eriksson, Röger and Helmert describe how inductive certificates, r-disjunctive or r-conjunctive certificates can be generated in explicit and symbolic blind search, in forward heuristic search with various heuristics (delete relaxation, critical path, pattern database, linear merge and shrink and landmarks) and the Trapper algorithm [19]. Additionally, the clause-learning algorithm by Steinmetz and Hoffmann [32] is able to generate 1-disjunctive certificates [12]. Since different heuristics sometimes require different formalisms to represent the state sets, it is not always possible to combine information from multiple heuristics into one certificate, compromising the generality criterium of Definition 12. The proof system described in the next section aims to overcome this shortcoming.

1.3.3 Proof System

The basic idea behind the proof system has been introduced in [18], but it has been significantly reworked in Eriksson's PhD thesis [12], therefore we will use the latter. The motivation behind the proof system is to give the prover more flexibility and to make composite reasoning possible. This is achieved by introducing a natural deduction style proof system with *basic statements* and *derivation steps*. The basic statements allow us to establish basic knowledge about the specific planning problem which can easily be verified. The derivation steps operate on a purely syntactic level, which makes them easy to verify, and allow us to derive knowledge from previously derived knowledge. The key concept used in the proof system are *dead states*, which are states that are not part of any plan.

Definition 18 (dead state and dead state set)**.** A state s is *dead* if no plan traverses s, i.e. there is no plan $\pi = \langle a_1, \ldots, a_n \rangle$ such that$s = I[a_1] \cdots [a_i]$ for some $0 \leq i \leq n$.[2] A set of states is dead iff all its elements are dead.

[2]The definition in [12] uses $1 \leq i \leq n$, but this does not match the informal definition, and it would allow the initial state do be dead even though a plan exists (in contradiction with Theorem 5.1 in [12]).

Equivalently, dead states can be defined as states that are not reachable (i.e. there is no path from I to the state) or not solvable (i.e. there is no path from the state to any goal state), or both. Note that the initial state of a planning problem Π is dead iff Π is unsolvable. Since deciding whether a planning problem is (un)solvable is PSPACE-complete [7], the problem of deciding whether a state is dead is also PSPACE-complete.

However, in many cases it is easier to decide whether states are dead, and the proof system allows to incrementally prove that sets of states are dead.

Definition 19 (proof system: types)**.** The proof system uses the following types:

State set variables:	$X_{\mathbf{R}}$	$::= \{I^\Pi\} \mid S^\Pi_G \mid \emptyset \mid \varphi_{\mathbf{R}}$
State set literals:	$L_{\mathbf{R}}$	$::= X_{\mathbf{R}} \mid \overline{X_{\mathbf{R}}}$
Action set expressions:	A	$::= A^\Pi \mid a \mid (A \cup A)$
State set expressions:	$S_{\mathbf{R}}$	$::= X_{\mathbf{R}} \mid \overline{S_{\mathbf{R}}} \mid (S_{\mathbf{R}} \cup S_{\mathbf{R}}) \mid (S_{\mathbf{R}} \cap S_{\mathbf{R}}) \mid S_{\mathbf{R}}[A] \mid [A]S_{\mathbf{R}}$
Set expressions:	E	$::= S_{\mathbf{R}} \mid A$

Here $\varphi_{\mathbf{R}}$ is a set explicitly given by an $\mathbf{R}$-formula, and a is any action in A^Π. We will also write S instead of $S_{\mathbf{R}}$ if the formalism is irrelevant, and we write E, E' or E'' instead of $Z : E$ to denote an object of type E.

In [12] it is not clear whether $X_{\mathbf{R}}$ refers to $X_{\mathbf{R}}$ or $\varphi_{\mathbf{R}}$, hence we modify the definition slightly to remove this ambiguity.[3]

All basic statements are of the form $S \subseteq S'$ or $A \subseteq A'$, but since these statements need to be verified by the validator, only some specific instances of the statement $S \subseteq S'$ are allowed, ensuring that this can be done efficiently without losing too much generality. Remember from Table 1.1 that not all formalisms efficiently support $\wedge\mathbf{C}$, and none of the formalisms efficiently supports $\vee\mathbf{C}$. For this reason a parameter r is used below to bound the size of unions and intersections, which ensures that the basic statements can still be efficiently verified.

Definition 20 (proof system: basic statements)**.** The following basic steps can be used as premises in the proof system:

B1 $\bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ with $|\mathcal{L}| + |\mathcal{L}'| \leq r$

B2 $(\bigcap_{X_{\mathbf{R}} \in \mathcal{X}} X_{\mathbf{R}})[A] \cap \bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ with $|\mathcal{X}| + |\mathcal{L}| + |\mathcal{L}'| \leq r$

B3 $[A](\bigcap_{X_{\mathbf{R}} \in \mathcal{X}} X_{\mathbf{R}}) \cap \bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}} \subseteq \bigcup_{L'_{\mathbf{R}} \in \mathcal{L}'} L'_{\mathbf{R}}$ with $|\mathcal{X}| + |\mathcal{L}| + |\mathcal{L}'| \leq r$

B4 $L_{\mathbf{R}} \subseteq L'_{\mathbf{R}'}$

B5 $A \subseteq A'$

For the derivation steps we follow the conventions used by [12] and write $S \sqsubseteq S'$ instead of $S \subseteq S'$ to stress that the rules operate on a purely syntactic level. For the same reason we also write $S \sqcup S'$ instead of $S \cup S'$ and $S \sqcap S'$ instead of $S \cap S'$. Note the for the syntax of the proof system, $S \sqsubseteq S'$ and

[3] Additionally, [12] only defines the negation of state set variables (not of state set expressions), even though the syntactic rules use $\overline{S}$ and not $\overline{X}$. While it does not seem to be useful in the current proof system to work with the negation of composite sets, there is nothing that forbids it either, hence we add it to the definition of $S_{\mathbf{R}}$ (instead of replacing $\overline{S}$ by $\overline{X}$).

$S \subseteq S'$ are still considered to be the same expression, so the basic statements can be used as premises for the derivation steps.

Definition 21 gives an overview of the derivation steps of the proof system. It would lead us too far to discuss all of them, but let us briefly motivate **CI** and **CG** (for a motivation of the rules related to dead states, see [12, Section 5.2.1]). If the initial state is dead, then there is no plan going through the initial state, but since the initial state is the first state of any plan for Π, this means that no plan for Π exists, thus Π is unsolvable. Similarly, if all goal states are dead, then there is no plan containing any goal state, so since the last state of every plan for Π has to be a goal state, there are no plans for Π, making Π unsolvable.

Definition 21 (proof system: derivation steps)**.** The following rules can be used to derive that states are dead:

Empty set **D**ead $\dfrac{}{\emptyset \text{ dead}}$ **ED**

Union **D**ead $\dfrac{S \text{ dead} \qquad S' \text{ dead}}{S \sqcup S' \text{ dead}}$ **UD**

Subset **D**ead $\dfrac{S' \text{ dead} \qquad S \sqsubseteq S'}{S \text{ dead}}$ **SD**

Progression **G**oal $\dfrac{S[A^\Pi] \sqsubseteq S \sqcup S' \qquad S' \text{ dead} \qquad S \sqcap S_G^\Pi \text{ dead}}{S \text{ dead}}$ **PG**

Progression **I**nitial $\dfrac{S[A^\Pi] \sqsubseteq S \sqcup S' \qquad S' \text{ dead} \qquad \{I^\Pi\} \sqsubseteq S}{\overline{S} \text{ dead}}$ **PI**

Regression **G**oal $\dfrac{[A^\Pi]S \sqsubseteq S \sqcup S' \qquad S' \text{ dead} \qquad \overline{S} \sqcap S_G^\Pi \text{ dead}}{\overline{S} \text{ dead}}$ **RG**

Regression **I**nitial $\dfrac{[A^\Pi]S \sqsubseteq S \sqcup S' \qquad S' \text{ dead} \qquad \{I^\Pi\} \sqsubseteq \overline{S}}{S \text{ dead}}$ **RI**

The following rules allow us to derive that a task is unsolvable:

Conclusion **I**nitial $\dfrac{\{I^\Pi\} \text{ dead}}{\Pi \text{ unsolvable}}$ **CI**

Conclusion **G**oal $\dfrac{S_G^\Pi \text{ dead}}{\Pi \text{ unsolvable}}$ **CG**

The rules below follow from basic set theory and allow us to reason about state and action set expressions:

Union **R**ight $\dfrac{}{E \sqsubseteq (E \sqcup E')}$ **UR**

Union **L**eft $\dfrac{}{E \sqsubseteq (E' \sqcup E)}$ **UL**

Intersection **R**ight $\dfrac{}{(E \sqcap E') \sqsubseteq E}$ **IR**

Intersection **L**eft $\dfrac{}{(E' \sqcap E) \sqsubseteq E}$ **IL**

DIstributivity $\dfrac{}{((E \sqcup E') \sqcap E'') \sqsubseteq ((E \sqcap E'') \sqcup (E' \sqcap E''))}$ **DI**

Subset **U**nion $\dfrac{E \sqsubseteq E'' \qquad E' \sqsubseteq E''}{(E \sqcup E') \sqsubseteq E''}$ **SU**

Subset **I**ntersection $\dfrac{E \sqsubseteq E' \qquad E \sqsubseteq E''}{E \sqsubseteq (E' \sqcap E'')}$ **SI**

Subset **T**ransitivity $\dfrac{E \sqsubseteq E' \qquad E' \sqsubseteq E''}{E \sqsubseteq E''}$ **ST**

The last group of rules allows us to reason about progressions and regressions:

Action **T**ransitivity $\dfrac{S[A] \sqsubseteq S' \qquad A' \sqsubseteq A}{S[A'] \sqsubseteq S'}$ **AT**

Action **U**nion $\dfrac{S[A] \sqsubseteq S' \qquad S[A'] \sqsubseteq S'}{S[A \sqcup A'] \sqsubseteq S'}$ **AU**

Progression **T**ransitivity $\dfrac{S[A] \sqsubseteq S'' \qquad S' \sqsubseteq S}{S'[A] \sqsubseteq S''}$ **PT**

Progression **U**nion $\dfrac{S[A] \sqsubseteq S'' \qquad S'[A] \sqsubseteq S''}{(S \sqcup S')[A] \sqsubseteq S''}$ **PU**

Progression to **R**egression $\dfrac{S[A] \sqsubseteq S'}{[A]\overline{S'} \sqsubseteq \overline{S}}$ **PR**

Regression to **P**rogression $\dfrac{[A]\overline{S'} \sqsubseteq \overline{S}}{S[A] \sqsubseteq S'}$ **RP**

The inference rules in Definition 21 are relatively easy to verify, since we only need to check that the premises and the conclusion match syntactically. On the contrary, for the basic statements from Definition 20 we need to verify that they hold semantically. For **B5** this is easy, because A and A' are either the set of all actions, or the number of actions they represent is linear in the size of the representation. Verification of the statements **B1-B4** is more complicated, and we refer to Eriksson's PhD thesis [12] for the full explication.

Theorem 22. *The statement* **B1** *can be validated in time polynomial in the total size of the involved formulas if* **R** *efficiently supports one of the following:*

- ***SE***, $\wedge$***BC***, $\vee$***BC***, $\bot$***C*** *and* $\top$***C***
- ***toCNF***, ***CE***, $\wedge$***BC*** *and* $\top$***C***
- ***toDNF***, ***IM***, $\vee$***BC*** *and* $\bot$***C***

Proof. This follows from Theorem 5.5 in [12]. We replace **VA** by **IM** (with the empty cube) or **SE** and $\top$**C**, and **CO** by **CE** (with the empty clause) or **SE** and $\bot$**C**. Note that if the formalism supports **toCNF** and **CE**, or **toDNF** and **IM**, then **SE** is not needed, as is argued in the proof of Theorem 5.5. After this, for each of the three options above there is a supported option in each of the cells in the table of Theorem 5.5. □

For BDDs we can use the first case of Theorem 22 to verify **B1**, and for Horn, 2CNF and MODS the second case.

Theorem 23. *The statements* **B2** *and* **B3** *can be validated in time polynomial in the total size of the involved formulas if* $\mathbf{R}$ *efficiently supports one of the following:*

- ***SE***, $\wedge$***BC***, $\vee$***BC***, ***CL***, $\bot$***C*** *and* $\boldsymbol{RN}_{\prec}$
- ***toCNF***, ***CE***, $\wedge$***BC***, ***CL***, *and* $\boldsymbol{RN}_{\prec}$

Proof. This follows from Theorem 5.6 in [12]. Again, for each of the two options one can find a supported option in each of the cells in the table of Theorem 5.5, when replacing **CO** by **CE** or **SE** and $\bot$**C**. Assuming that $\mathcal{X}$ is non-empty, one can argue that the left-hand-side of (5.3) and (5.4) is always non-empty, so $\top$**C** (or **VA**) is not needed. □

For BDDs the first case of Theorem 23 is used to verify **B2** and **B3**, whereas for Horn, 2CNF and MODS the second one is used.

Theorem 24. *If* $\mathbf{R} = \mathbf{R}'$*, then* **B4** *can be verified efficiently using* ***SE****. Otherwise, if* $\mathbf{R} \neq \mathbf{R}'$ *the statement* **B4** *with can be validated in time polynomial in* $\|L_{\mathbf{R}}\|$ *and* $\|L'_{\mathbf{R}'}\|$ *for the case in the first column if* $\mathbf{R}$ *and* $\mathbf{R}'$ *support one of the options in the second column efficiently.*

	$\mathbf{R}$	$\mathbf{R}'$
$\varphi_{\mathbf{R}} \subseteq \psi_{\mathbf{R}'}$	***toDNF***	***IM***
	CE	***toCNF***
$\overline{\varphi_{\mathbf{R}}} \subseteq \psi_{\mathbf{R}'}$	***toCNF***	***IM***
	IM	***toCNF***
$\varphi_{\mathbf{R}} \subseteq \overline{\psi_{\mathbf{R}'}}$	***toDNF***	***CE***
	CE	***toDNF***
$\overline{\varphi_{\mathbf{R}}} \subseteq \overline{\psi_{\mathbf{R}'}}$	***IM***	***toDNF***
	toCNF	***CE***

If $\mathbf{R}$ *and/or* $\mathbf{R}'$ *supports* $\neg$***C*** *and* $\overline{\varphi_{\mathbf{R}}}$ *and/or* $\overline{\psi_{\mathbf{R}'}}$ *occurs, then the case can also be reduced to* $\varphi_{\mathbf{R}} \subseteq \psi_{\mathbf{R}'}$*.*

Proof. This follows from Theorem 5.7 in [12]. In Theorem 5.7 each of the cases has additional options using the operations **ME** and **MO**, which can only used when one of the formalisms is MODS (at least when it comes to the formalisms we are interested in). However, all cases with MODS can also be done efficiently using **toDNF** or **toCNF**, so for simplicity we use those. Furthermore, this avoids the complications discussed in the proof of Theorem 5.7 when $\varphi_{\mathbf{R}}$ and $\psi_{\mathbf{R}'}$ have different sets of variables, so in these cases it is likely more efficient to use the options listed here. □

If $\mathbf{R}$ or $\mathbf{R}'$ is MODS, then we can always efficiently validate **B4** by using **toDNF** or **toCNF** for MODS and **IM** or **CE** for the other formalism. If $\mathbf{R}$ and $\mathbf{R}'$ are BDD, Horn or 2CNF (with $\mathbf{R} \neq \mathbf{R}'$), then $\varphi_{\mathbf{R}} \subseteq \overline{\psi_{\mathbf{R}'}}$ (third case) cannot be validated efficiently, as this would require that $\mathbf{R}$ or $\mathbf{R}'$ efficiently supports **toDNF** or $\mathbf{R}'$ efficiently supports $\neg$**C** and **toCNF**. Similarly, $\varphi_{\mathbf{R}} \subseteq \psi_{\mathbf{R}'}$ (first case) cannot be efficiently validated if $\mathbf{R}$ is Horn or 2CNF and $\mathbf{R}'$ is BDD, and $\overline{\varphi_{\mathbf{R}}} \subseteq \overline{\psi_{\mathbf{R}'}}$ (fourth case) cannot be efficiently validated if $\mathbf{R}$ is BDD and $\mathbf{R}'$ is Horn or 2CNF. Other cases are efficiently supported by using **toCNF** for Horn or 2CNF.

Inductive certificates can be translated into proof trees in the proof system.

Theorem 25. *Given an inductive* **R***-certificate for* Π*, it is possible to construct derivation in the proof system in linear time. This proof can be verified using the operations* $\boldsymbol{MO}$*,* $\boldsymbol{CE}$*,* $\boldsymbol{SE}$*,* $\wedge\boldsymbol{BC}$*,* $\boldsymbol{CL}$*,* $\boldsymbol{RN}_{\prec}$ *from Definition 13.*

Proof. See Theorem 5.8 in [12]. □

There are similar reductions for r-disjunctive and r-conjunctive certificates; albeit with different operations and a higher time complexity. Completeness of the inductive certificates (Theorem 16) together with this reduction implies that the proof system is also complete. Soundness of the proof system follows from the correctness of all individual rules.

Theorem 26 (soundness and completeness)**.** *Let* Π *be a STRIPS planning task. There exists a proof of unsolvability of* Π *in the proof system iff* Π *is unsolvable.*

Proof. See Theorem 5.9 in [12]. □

All planning algorithms that can generate inductive certificates can also generate proofs in the proof system, since we can use the reduction to translate the inductive certificates to proofs. For several of these formalisms Eriksson also discusses more efficient methods to generate the certificates [12]. Additionally, Eriksson, Röger and Helmert argue that the proof system allows combining information from multiple heuristics in heuristic search, and that both the algorithm described by Steinmetz and Hoffmann [32] and the algorithm described by Alcázar and Torralba [2] can generate proofs in the proof system [18]. Eriksson and Helmert [15] have extended the proof system with rules that integrate UNSAT certificates, and they showed that the Property Directed Reachability algorithm [33] can generate certificates in this extended proof system. They also showed that a variant of the algorithm that does not make SAT-calls can generate certificates in the original proof system.

1.3.4 Certificates of Optimality

At the time of writing three certificates for optimality have been introduced: compilation to unsolvability [28], an adaptation of the proof system discussed in Section 1.3.3 [28] and an approach using pseudo-Boolean proofs [10].

In the first approach [28, "Compilation to Unsolvability"] the planning task Π is compiled into a modified planning task Π^x which also keeps track of the cost, and ensures the cost of plans is less than x, where $x \in \mathbb{N}$. To verify that a plan with cost x is optimal, it is then sufficient to verify that the plan is valid and that Π^x is unsolvable. While this approach is sound, complete and efficiently verifiable, it is not always possible to efficiently generate the certificates. The reason for this is that, while the size of Π^x is linear in x, exploring the state-space of Π^x can incur an exponential overhead in x.

The second approach [28, "Optimality Proof System"] replaces the rules about dead states in Definition 21 by rules to reason about lower bounds on the cost of plans though states. This approach is also sound and complete, but according to Mugdan, Christen and Eriksson the other three criteria of Definition 12 don't have clear-cut answers.

The third approach [10] uses an invariant φ to make an overapproximation of the reachable state-cost pairs (i.e. a pair of a state and a cost to reach that state from the initial state). Here, a certificate to show that the optimal solution has cost at least B consists of

- an invariant φ over state-cost pairs (s, c),
- a proof that the state-cost pair $(I^\Pi, 0)$ satisfies φ,
- a proof that for every goal state s and cost $c < B$, (s, c) does not satisfy φ, and
- a proof that if (s, c) satisfies φ and a is an action applicable in s with $c + \operatorname{cost} a < B$, then $(s[a], c + \operatorname{cost}(a))$ also satisfies φ.

The invariant is represented by a circuit where the gates evaluate pseudo-Boolean constraints. This approach has been shown to be sound, complete and efficiently verifiable. Certificates can be efficiently generated when using A^* with $h^{\max}$ or pattern base-heuristics. The authors expect that the certificates can be efficiently generated for other approaches as well.

1.3.5 Reduction to Model Checking

Recently, Wang and Abdulaziz formally verified an encoding from temporal planning problems (which includes the STRIPS formalism we use) to network of timed automata in Isabelle/HOL [36]. This encoding reduces the planning problem to checking whether there is a model for this network of timed automata, which can be solved using a certified model checker. There is an existing formally verified validator for these certificates that can be used to verify the correctness of the output of the solver. This approach gives similar correctness guarantees as the formally verified validator we implement. However, it cannot be used to certify existing planning algorithms, since they cannot produce these certificates.

1.3.6 Implementations

Several plan validators have been implemented, most notably VAL in C++ [25], INVAL in Lisp [23] and a formally verified validator in Isabelle/HOL by Abdulaziz and Lammich [1]. All these validators read the planning description in PDDL format [21, 30].

There are C++ implementations of validators for the inductive certificates [13], the proof system for unsolvability certificates [14, 13], the optimality certificates using the compilation to unsolvability certificates [27] and the proof system for optimality certificates [14, 27]. For all these certificate systems Fast-Downward [24] has been extended to generate the certificates. Additionally, for the unsolvability proof system there are also extensions for SymPA [34, 22] and an implementation with the h^C-based clause-learning algorithm [32, 13], which are both based on Fast-Downward. Furthermore, an implementation of Property Directed Reachability (without SAT-calls) has been extended to output proofs [17]. For verifying these proofs, the C++ implementation of the proof system for unsolvability certificates has been extended to support dual-Horn formulas for representing state-sets [16]. All implementations use Fast-Downward to parse the planning task in PDDL format, ground the problem and output a description of the problem in STRIPS, which is then used as input problem for the validator.

This is reasonable for a first implementation of the certificates, since the theoretical framework has been developed in STRIPS. It is however not acceptable for a fully formalized version, because this assumes that the grounding by Fast-Downward and conversion to STRIPS are correctly implemented in Fast-Downward. Therefore, to have a fully formalized version, one would have to implement a formalized grounder which grounds the PDDL description and verifies that it is equivalent to the STRIPS version. The plan validator in Isabelle/HOL has such a grounder, but it might be hard to integrate this grounder in Lean.

At the time of writing, there is no implementation for the optimality certificates based on pseudo-Boolean proof logging. An advantage of this certificate system is that the main part of the proofs can be verified by a pseudo-Boolean proof checker like VeriPB [35, 5] or the formally verified CakePB [8].

1.4 Lean 4

Lean 4 is an interactive proof assistant and functional programming language based on the Calculus of Inductive Constructions [26]. In this section we give a quick overview of the basic syntax of Lean 4.

Types in Lean can be constructed using inductive definitions. As an example, the definition of natural numbers in this syntax can be seen below. The terms of the type `Nat` are `Nat.zero`, `Nat.succ Nat.zero`, `Nat.succ (Nat.succ Nat.zero)`, and so on.

```
inductive Nat : Type
| zero : Nat
| succ : Nat → Nat
```

The type `Nat` has two constructors `Nat.zero` and `Nat.succ`. For types having only one constructor, it can be more convenient to use the `structure` syntax. This can for example be used to define a structure `Point` with two coordinates of type α. Given two terms `a` and `b` of type α, `Point.mk a b` will be a term of type `Point` α, and we can obtain its components using `(Point.mk a b).x` and `(Point.mk a b).y`. Here, α can be any type, so for example `Point.mk 0 1` will have type `Point` $\mathbb{N}$, since `0` and `1` have type $\mathbb{N}$.

```
structure Point (α : Type) where
  x : α
  y : α
```

To define functions we use the keyword `def`, as shown below. The curly brackets around the parameter α in `Point.swap` indicate it is an implicit parameter, so we can call these functions using `add_one 3` or `Point.swap x` when `x` has type `Point` α for some type α. For the latter Lean also allows the notation `x.swap`.

```
def add_one (n : Nat) : Nat := n + 1

def Point.swap {α : Type} (p : Point α) : Point α := Point.mk p.y p.x
```

Leans type syntax is quite flexible, and it allows omitting type annotation if the type system can infer the types. The function `Point.swap` from above could for example also be written as follows:

```
def Point.swap' {α} (p : Point α) := Point.mk p.y p.x
```

Likewise, definitions can be used to define types. In the next chapter we will often work with natural numbers smaller than a given number `n`. These can be defined as the subtype of natural numbers `i` which satisfy the proposition `i < n`, as done in `Fin` and the equivalent `Fin'` with full type signature. For subtypes, we can use the methods `Subtype.val` and `Subtype.prop` to get the original value and the property, so if `i` is an element of `Fin n`, then `i.val` is the underlying natural number and `i.prop` gives a proof of `i.val < n`.

```
def Fin n := { i : ℕ // i < n }
def Fin' (n : ℕ) : Type := { i : ℕ // i < n }
```

Propositions like `i < n` or `a = b ↔ b = a` have type `Prop` and can be proven by constructing a term of type `i < n` or `a = b ↔ b = a`. Instead of having to provide this term explicitly, the keyword `by` can be used to enter tactic mode, where tactic commands can be used to prove the goal incrementally. In the example below the tactic `tauto` (short for tautology) immediately proves the goal `a = b ↔ b = a`.

```
theorem Eq.symm {α} (a b : α) : a = b ↔ b = a :=
  by tauto
```

Lean 4 also has keywords `lemma` which works the same as `theorem` and `abbrev` which works like `def`, but additionally instructs Lean to always unfold the definition. The last feature we want to highlight are *type classes*, which allow defining polymorphic functions that can be overloaded. For example, Lean provides a type class `Membership` with method `mem`, where the purpose of the class is to provide the notation `a ∈ s` as a shorthand for `mem s a`. What it exactly means that `a` is a member of `s` depends on the type of `s` and `a`, and can be specified by declaring an instance for the class `Mem`, as is done below for lists. Here, `List.Mem` defines what it means to be a member of a list using an inductively defined proposition, where the first case states that the head of a list is a member of the list, and the second one that any member of the tail is a member of the list.

```
class Membership (α γ : Type) where
  mem : γ → α → Prop

inductive List.Mem (a : α) : List α → Prop
| head (as : List α) : Mem a (a :: as)
| tail (b : α) {as : List α} : Mem a as → Mem a (b :: as)

instance {α} : Membership α (List α) where
  mem as a := List.Mem a as
```

Chapter 2

Implementation in Lean 4

In this chapter we discuss the formalization of the proof system and a formalized validator for this proof system. The full Lean 4 code can be found on GitHub[1]. On multiple occasions we will make the distinction between *runtime* and *proof time*. Code that is used at proof time is only used during verification of the validator as a whole, and it is not executed during validation of a specific certificate. On the other hand, code that is used at runtime can be executed during validation, and should therefore be efficient.

Figure 2.1 shows the dependency graph of the project and Table 2.1 shows the number of lines of some files. Because of time constraints the implementation has not been finished, therefore some files still contain *sorries* (a sorry is a placeholder for a missing proof or term in Lean 4). In particular, the following still need to be implemented:

- Verification of the basic rule **B4**, this is the sorry in `BasicRules`.
- The different formalisms for representing state sets have not yet been implemented. This explains the sorries in `Bdd`, `Horn` and in `Mods`. Helve, the C++ implementation of the proof system by Eriksson [14], does not contain an implementation for 2CNF. We use the same format for parsing the certificates as Helve, and therefore there is no file for 2CNF-formulas yet. Still, the plan is to extend this format and implement 2CNF once the other formalisms have been implemented.
- In the file `Certificate.Parser`, the parsers for state sets in the different formalisms are missing.

Our implementation contains two versions of the proof system, one that is nicer for theoretical purposes, and one suited for actually validating certificates. The first version closely resembles the definitions of Section 1.3.3, and is used to prove completeness and soundness of the system. It is implemented in the file `ProofSystem`. The second version is implemented in the file `Certificate` and follows the format of the certificates used in Helve.

In the remainder of this chapter we will explain the different files in more detail. When referencing files we will drop the extension `.lean` and the folder `Validator`, so we will write `PlanningTask.Parser` (or even `Parser` if the folder is clear from the context) instead of `Validator.PlanningTask.Parser.lean`. We will only briefly discuss the files `Basic` and `Error`. `Basic` defines additional functionality for

[1] `https://github.com/AmosNico/validator/tree/Master-Thesis`

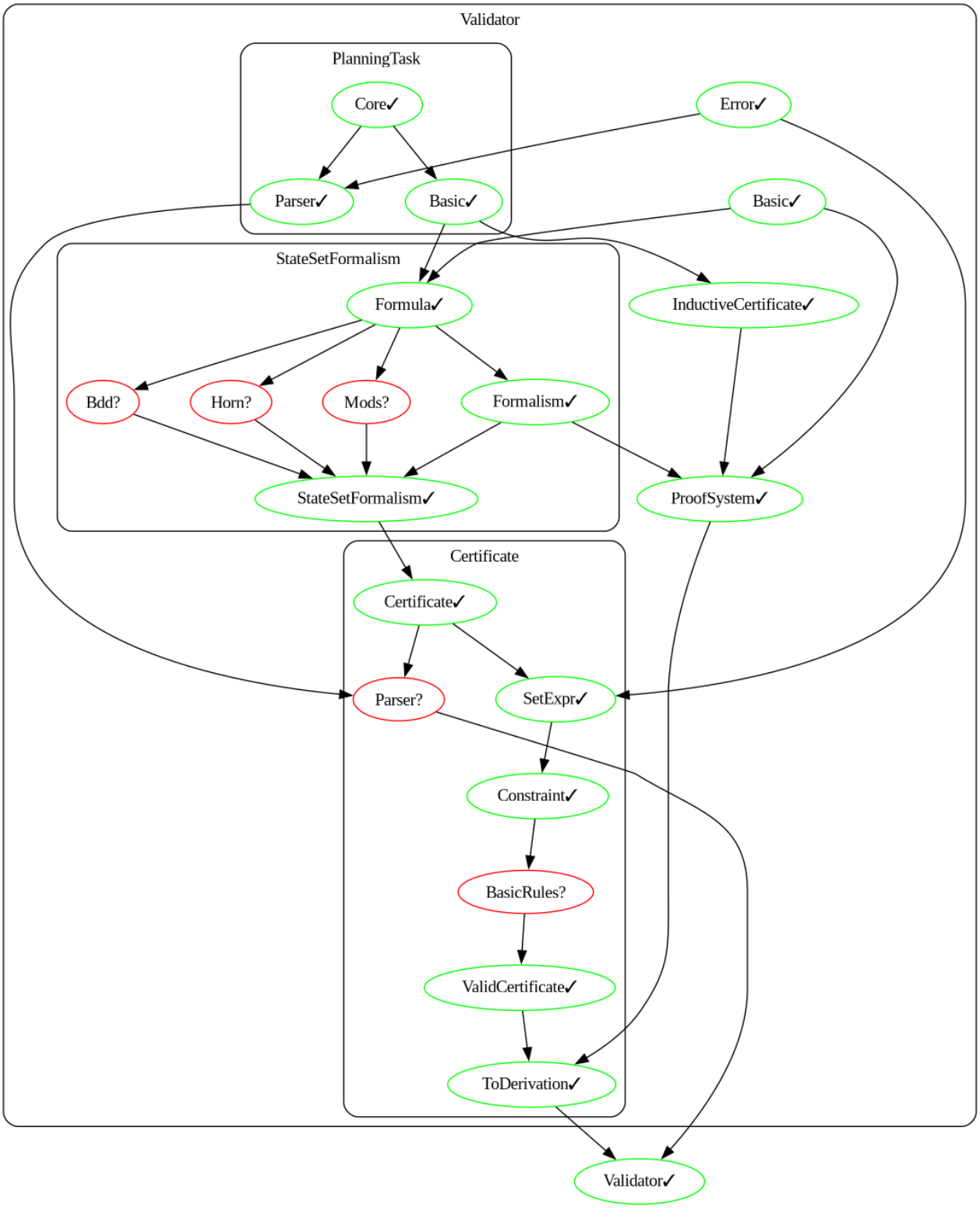


Figure 2.1: The dependency graph containing all files of the project. The nodes represent files, and the surrounding boxes different subfolders of the project. An arrow from file A to file B means that file A is imported by file B. The green files with checkmarks don't contain any "sorry", whereas the red files with question marks still contain sorries.

File	Lines of Code
`Certificate.BasicRules`	783
`StateSetFormalism.Formula`	771
`Certificate.Constraint`	605
`Certificate.ToDerivation`	571
Core	411
Total	6281
Total Helve	7866

Table 2.1: The lines of code for the four longest files, the files in the core and the total number of lines. The core consists of the files which together guarantee the correctness of the validator, being `PlanningTask.Core`, `PlanningTask.Parser` and `Validator`. Note that these numbers also include lines with documentation, which varies widely among files. As a reference, the total number of lines of Helve is also shown.

`List`, `Array` and `Set` that is not in Mathlib. `Error` implements error handling for the validator and the parsers. The files in the folder `PlanningTask`, defining the STRIPS formalism, are discussed in Section 2.1. Next, Section 2.2 describes files `InductiveCertificate` and `ProofSystem` which define the simple inductive certificates and the proof system, and prove their soundness and completeness. In Section 2.3 the folder `StateSetFormalism` implementing the formalisms for representing states is discussed. Finally, Section 2.4 explains the folder `Certificate` and the main file `Validator` which implement the functionality for validating the certificates.

2.1 Planning Tasks

The folder `PlanningTask` contains three files related to the formalization of planning tasks:

- `Core` formalizes the definition of a STRIPS planning task, and the definition of what it means to be unsolvable. It contains all definitions essential for the correctness of the validator.
- `Basic` contains additional definitions about planning tasks that are not essential for the correctness of the validator, like progression and regression. Additionally, it gives various lemmas to work with the definitions in `Core` and `Basic`.
- `Parser` implements a parser for STRIPS planning problems.

We will go over the files one by one to discuss the most important features and some design choices.

2.1.1 Core

There are multiple ways to formalize the definitions in Section 1.1. For example, we could formalize actions (Definition 4) as follows:

```
abbrev Variable := ℕ

structure VarSet (V : Finset Variable) where
  set : Finset Variable
  property : set ⊆ V
```

```
structure Action V where
  pre : VarSet V
  add : VarSet V
  del : VarSet V
  cost : ℕ
```

While this definition stays close to Definition 4, it has the disadvantage that `VarSet` is not actually a set, so if we want to state that `vars1` $\subseteq$ `vars2` for two elements of `VarSet`, then we would either have to define $\subseteq$ for `VarSet` or use `vars1.set` $\subseteq$ `vars2.set` instead. To avoid this issue we can use a type to represent the set of all variables `V`. A simple choice is `Fin n`, so the variables are $0, 1, \ldots, n-1$.

```
abbrev VarSet (n : ℕ) := Finset (Fin n)

structure Action n where
  pre : VarSet n
  add : VarSet n
  del : VarSet n
  cost : ℕ
```

This formalization is convenient for reasoning, but it is still not optimal to use in an actual validator, as it would be slightly more efficient to just represent the preconditions in a sorted `List`. An additional reason is that there is no easy way to print a `Finset`, as this would require ordering its elements in some way. Therefore, the validator uses a version of `VarSet` using a subtype of `List` to actually store the preconditions and the adding and deleting effects. To reason about `VarSet` we use `Set` instead of `Finset`, since `Set` is slightly easier to work with, and the use of the finite type `Fin n` already guarantees that the sets are finite.

```
abbrev VarSet n := Set (Fin n)
abbrev VarSet' n := { vars : List (Fin n) // vars.Sorted (· < ·) }

def convertVarSet {n} (V : VarSet' n) : VarSet n :=
  V.val.toFinset

structure Action n where
  name : String
  pre' : VarSet' n
  add' : VarSet' n
  del' : VarSet' n
  cost : ℕ
  deriving Repr, DecidableEq

namespace Action
  def pre {n} (a : Action n) : VarSet n := convertVarSet a.pre'
  def add {n} (a : Action n) : VarSet n := convertVarSet a.add'
  def del {n} (a : Action n) : VarSet n := convertVarSet a.del'
end Action
```

Similarly, other concepts like states (Definition 2), sets of actions and STRIPS planning tasks (Definition 5) themselves are implemented, with separate methods for verifying and theoretical purposes. As before, the primed versions are used at runtime and the unprimed versions only at proof time.

```
abbrev Actions n := Set (Action n)
abbrev Actions' n := List (Action n)

abbrev State n := Set (Fin n)
abbrev State' n := BitVec n
abbrev States n := Set (State n)

def convertState {n} (s' : State' n) : State n :=
  { i | s'[i] }

structure STRIPS n where
  varNames : Vector String n
  actions' : Actions' n
  init' : State' n
  goal' : VarSet' n
  deriving Repr

namespace STRIPS
def actions {n} (pt : STRIPS n) : Actions n :=
  List.toFinset pt.actions'

def init {n} (pt : STRIPS n) : State n :=
  convertState pt.init'

def GoalState {n} (pt : STRIPS n) (s : State n) : Prop :=
  convertVarSet pt.goal' ⊆ s
end STRIPS
```

We use bit vectors to represent states during verification because of their high performance. In the case of `STRIPS.actions'` it is impossible to use `Set` or `Finset` during verification, because the certificates identify the actions by their array index. For goal states it makes more sense to use a `Prop` as a primitive for theoretical purposes. We continue with the formalized definitions for applicability and successor states.

```
abbrev Applicable {n} (s : State n) (a : Action n) : Prop :=
  a.pre ⊆ s
abbrev Successor {n} (a : Action n) (s s' : State n) : Prop :=
  Applicable s a ∧ s' = (s \ a.del) ∪ a.add
```

For `Successor`, we could also use a definition which, given a state `s`, an action `a` and a proof of `Applicable s a`, returns the successor state `(s \ a.del) ∪ a.add`. However, the definition above is easier to work with because it naturally contains the requirement of applicability (otherwise we would have to supply a proof every time we talk about the successor state). We finish this section with the formalization of paths in the state-space, plans and unsolvability of states and planning tasks.

```
inductive Path {n} (pt : STRIPS n) : State n → State n → Type
| empty s : Path pt s s
| cons a {s1} s2 {s3}
  (ha : a ∈ pt.actions) (succ : Successor a s1 s2) (π : Path pt s2 s3) : Path pt s1 s3

structure Plan {n} (pt : STRIPS n) (s : State n) where
  last : State n
  path : Path pt s last
  goal : pt.GoalState last

abbrev UnsolvableState {n} (pt : STRIPS n) (s : State n):=
  IsEmpty (Plan pt s)
abbrev Unsolvable {n} (pt : STRIPS n) :=
  UnsolvableState pt pt.init
```

2.1.2 Basic

The file `Basic` starts by implementing some functionality related to paths. Note that `Path.cons` in the definition of `Path` extends paths at the front. This is needed when we want to prove something by structural induction on a path where the last state is fixed, for example because it is a goal state (e.g. `InductiveCertificate.soundness'` on page 28 and `progression_aux`[2]) or when we need to access the second state of the path (e.g. `regression_aux`[3]). However, sometimes we need to append a state at the end (e.g. `InductiveCertificate.completeness'`, see page 29) or we need to do induction where the first state is fixed (e.g. because it is the initial state like in `regression_aux`[3]). For these cases it is useful to have a definition which extends paths at the back, as follows:

```
def Path.snoc {n} {pt : STRIPS n} a {s1} s2 {s3} (ha : a ∈ pt.actions)
 (π : Path pt s1 s2) (succ : Successor a s2 s3) : Path pt s1 s3 :=
   match π with
   | empty s => cons a s3 ha succ (empty s3)
   | cons a' s4 ha' succ' π' =>
       let π'' := snoc a s2 ha π' succ
       cons a' s4 ha' succ' π''
```

The name `snoc` comes from reading `cons` backwards. To make `Path.snoc` more usable, the file provides a lemma `Path.snocCases` which allows doing cases on `Path.empty` and `Path.snoc` and a lemma allowing to convert a path constructed by `cons` to a path constructed by `snoc`. Additionally, the file contains definitions of the length of paths (`Path.length`), the concatenation of two paths (`Path.append`), membership of states in paths (`Path.Mem`) and multiple lemmas to work with these definitions. We will highlight `Path.Mem` and the lemma `Path.split`, which can be used to split a path at a given state on the path.

[2] https://github.com/AmosNico/validator/blob/12646dc3f866a3fac0720b4319a003ecbacef608/Validator/ProofSystem.lean#L172

[3] https://github.com/AmosNico/validator/blob/5366bb69a0db48ea5b7ae5cadaee9c17a38ba6bb/Validator/ProofSystem.lean#L232

```
def Path.Mem {n} {pt : STRIPS n} {s1 s2} (s : State n) : (π : Path pt s1 s2) → Prop
| empty s' => s = s'
| cons _ _ _ _ π => s = s1 ∨ Mem s π

instance {n} {pt : STRIPS n} {s1 s2} : Membership (State n) (Path pt s1 s2) where
  mem π s := Path.Mem s π

lemma Path.split {n} {pt : STRIPS n} {s1 s2 s} (π : Path pt s1 s2) (h : s ∈ π) :
  Nonempty (Path pt s1 s × Path pt s s2) := ...
```

Intuitively, it would make sense to make `Path.split` a definition returning the two paths instead of a lemma, but the lemma suffices for our purposes and is slightly easier to implement. When the specific path itself is irrelevant, we can use the notion of reachability instead.

```
abbrev Reachable {n} (pt : STRIPS n) (s s' : State n) : Prop :=
  Nonempty (Path pt s s')
```

The file also contains definitions for the set of all goal states, the progression of a set of actions (`STRIPS.progression`, Definition 10) and regression of a set of actions (`STRIPS.regression`, Definition 11). Recall that the set of variables is defined as the type `Fin n`, therefore the states are exactly the elements of `Set (Fin n)` (i.e. `State n`). For progression and regression, we first define the progression and regression of a single action and then take the union over all actions in the given set `A`. Finally, there are multiple lemmas describing monotonicity of `STRIPS.progression` and `STRIPS.regression` and their interaction with membership, unions and each other, which we will not describe here.

```
namespace STRIPS
def goal_states {n} (pt : STRIPS n) : States n :=
  { s | pt.GoalState s }

def progression' {n} (_ : STRIPS n) (S : States n) (a : Action n) : States n :=
  { s | ∃ s' ∈ S, Successor a s' s }

def progression {n} (pt : STRIPS n) (S : States n) (A : Actions n) : States n :=
  { s | ∃ a ∈ A, s ∈ progression' pt S a }

def regression' {n} (_ : STRIPS n) (S : States n) (a : Action n) : States n :=
  { s | ∃ s' ∈ S, Successor a s s' }

def regression {n} (pt : STRIPS n) (S : States n) (A : Actions n) : States n :=
  { s | ∃ a ∈ A, s ∈ regression' pt S a }
end STRIPS
```

2.1.3 Parser

The file `Parser` contains some general parsing functions and a parser for STRIPS planning tasks. The parser is a combinatorial parser built using the lean4-parser library [11]. The planning task should be provided in a separate file in the format below, which is the same format as Helve uses.

```
begin_atoms: <#atoms>
<atom 0>
<atom 1>
... (names of all atoms, one on each line)
end_atoms
begin_init
<initital state atom index 0>
<initital state atom index 1>
... (indexes of atoms that are true in initial state, one on each line)
end_init
begin_goal
<goal atom index 0>
<goal atom index 1>
... (indexes of atoms that are true in goal, one on each line)
end_goal
begin_actions: <#actions>
begin_action
<action_name>
cost: <action_cost>
PRE: <precondition atom index 0>
ADD: <added atom index 0>
DEL: <deleted atom index 0>
... (more PRE, ADD and DEL in any order, one on each line)
end_action
... (more actions)
end_actions
```

While the STRIPS formalism is widely used for theoretical purposes, the more expressive PDDL formalism is used for representing planning tasks in practice [21, 30]. Additionally, as an intermediate format when solving planning problems, the SAS+ format [29] is often used. The format above is non-standard, which means that solvers need to output the description of the planning task themselves. As discussed in Section 1.3.6, this is not ideal. In the future it would be better to be able to parse planning tasks in the PDDL format, or at least the SAS+ format.

2.2 Inductive Certificates and Proof System

The files `InductiveCertificate` and `ProofSystem` define inductive certificates and the certificates in the proof system of unsolvability and prove their soundness and completeness. Note that these definitions are only used at proof time, not at runtime. Later, in Section 2.4.8, we will relate the content of `ProofSystem` to the certificates used at runtime.

2.2.1 InductiveCertificate

The file starts by defining inductive sets (Definition 14) and inductive certificates (Definition 15):

```
abbrev InductiveSet {n} (pt : STRIPS n) (S : States n) :=
  pt.progression S pt.actions ⊆ S

abbrev InductiveCertificateState {n} (pt : STRIPS n) (s : State n) (S : States n) :=
    s ∈ S ∧ (∀ s ∈ S, ¬ pt.GoalState s) ∧ InductiveSet pt S

abbrev InductiveCertificate {n} (pt : STRIPS n) (S : States n) :=
  InductiveCertificateState pt pt.init S
```

Next up are the proofs of soundness and completeness, which we will discuss in more detail. We first prove the statement for inductive certificates of an arbitrary state `s` in `soundness'`, and use this to prove soundness for the initial state in `soundness`.

The proof starts by introducing the hypotheses `hs : s ∈ S`, `h1 : ∀ s ∈ S, ¬ pt.GoalState s` and `h2 : InductiveSet pt S`, corresponding to the conditions in `InductiveCertificateState`. Recall from Section 2.1.1 that `UnsolvableState pt s` is defined as the inductive type `IsEmpty (Plan pt s)`, hence we can use the tactic `constructor` to apply the constructor of `IsEmpty` to the goal. This changes the goal to `∀ (a : Plan pt s), False`. We introduce and decompose the plan in the goal, giving us a state `s'`, a path π from `s` to `s'` and a proof `h3` that `s'` is a goal state. The goal now becomes `False`, which means that we need to show that the hypotheses lead to a contradiction. We do this by structural induction on the path π. If the path is empty, then `s` and `s'` are the same state, and the combination of `h1`, `hs` and `h3` leads to a contradiction. In the inductive case it suffices to show that `s2 ∈ S` by the induction hypothesis `ih : s2 ∈ S → pt.GoalState s3 → False` combined with `h3`. Since `S` is an inductive set (by the hypothesis `h2`), it is sufficient to show that `s2` is in the progression of `S` with the actions `pt.actions`. Finally, this follows from the lemma `mem_progression_of_successor` which states that if `s1 ∈ S a ∈ A` and `Successor a s1 s2`, then `s2` is in the progression of `S` with `A`.

```
theorem soundness' {n} {pt : STRIPS n} {s S} :
  InductiveCertificateState pt s S → UnsolvableState pt s :=
  by
    rintro ⟨hs, h1, h2⟩
    constructor
    rintro ⟨s', π, h3⟩
    induction π with
    | empty s' => exact h1 s' hs h3
    | @cons a s1 s2 s3 ha h π ih =>
      refine ih ?_ h3
      show s2 ∈ S
      apply h2
      exact mem_progression_of_successor hs ha h

theorem soundness {n} {pt : STRIPS n} {S} :
  InductiveCertificate pt S → Unsolvable pt :=
  soundness'
```

For the proof of `completeness'` below, the hypothesis `h1` states that no plan for the state `s` exists (i.e. `∀ (a : Plan pt s), False`). We show that this implies that the set of states reachable form `s` is an inductive certificate. The tactic `simp` simplifies the goal and `split_ands` splits all conditions from `InductiveCertificateState` into separate goals. First we need to show `Reachable s s`, which follows immediately by the lemma `reachable_self`. Next, we need to show that none of the reachable states is a goal state, which in Lean corresponds to the goal `∀ (s' : State n) (π : Path pt s s'), ¬pt.GoalState s'`. To prove this, we assume that π is a path from `s` to a goal state `s'` with `h3 : pt.GoalState s'`. By `h1` it is sufficient to give a plan for `s`, for which we can use π and `h3`.

Lastly, we need to show that the set of reachable states is inductive, i.e. from `h : s' ∈ pt.progression {s' | Reachable pt s s'} pt.actions` we need to prove that `s'` is reachable from `s`. After simplification, the hypothesis `h` yields an action `a`, a state `s''` and hypotheses `h2 : Reachable pt s s''` and `h3 : Successor a s'' s'`. From `h2` we obtain a path from `s` to `s''`, which we can combine with `h3` to obtain a path from `s` to `s'` showing that `s'` is reachable from `s`.

```
theorem completeness' {n} {pt : STRIPS n} {s} :
  UnsolvableState pt s → ∃ S, InductiveCertificateState pt s S :=
  by
    unfold UnsolvableState
    rintro ⟨h1⟩
    use { s' | Reachable pt s s' }
    simp [InductiveCertificateState]
    split_ands
    · exact reachable_self s
    · intro s' π h3
      apply h1
      exact Plan.mk s' π h3
    · intro s' h
      simp_all [STRIPS.progression, STRIPS.progression']
      rcases h with ⟨a, ha, s'', h2, h3⟩
      obtain π : Path pt s s'' := Classical.choice h2
      constructor
      show Path pt s s'
      exact Path.snoc a s'' ha π h3

theorem completeness {n} {pt : STRIPS n} :
  Unsolvable pt → ∃ S, InductiveCertificate pt S :=
  completeness'
```

Initially `States n` was defined as `Finset (State n)`. For the overall project it hardly made a difference whether this definition uses `Set` or `Finset`, except for the line `use { s' | Reachable pt s s' }` in `completeness'`. The use of `Finset` required showing that `Reachable pt s s'` is decidable, which essentially requires constructing the set of all reachable states. This was done by taking the union of the sets `expand pt {s} k` for all $k \in \mathbb{N}$, where `expand pt S k` contains all states that are reachable in `k` steps from a state in `S`. While this construction is not needed any more, it can still be found at the end of the file.

2.2.2 Prerequisites from Formalism

While `ProofSystem` depends on `Formalism` (as can be seen in Figure 2.1), it makes more sense to first discuss `ProofSystem`. For now, it is sufficient to know that there is a class `Formalism pt R` stating that the type `R` can be used as a formalism for the planning task `pt` and that `Formalism.toStates pt R` returns the set of states corresponding to the formula. This is then used to inductively define which sets of states are state set variables and state set literals, as in Definition 19.

```
inductive IsVariable {n} (pt : STRIPS n) R [Formalism pt R] : States n → Prop
| empty : IsVariable pt R ∅
| init : IsVariable pt R {pt.init}
| goal : IsVariable pt R pt.goal_states
| explicit (φ : R) : IsVariable pt R (Formalism.toStates pt φ)

inductive IsLiteral {n} (pt : STRIPS n) R [Formalism pt R] : States n → Prop
| pos {S} : IsVariable pt R S → IsLiteral pt R S
| neg {S} : IsVariable pt R S → IsLiteral pt R (Sᶜ)
```

Using `IsVariable` and `IsLiteral` we can define finite unions of state set literals, and finite intersections of state set variables and state set literals, which are used in the basic rules **B1**-**B3**. Lastly, we define the propositions `IsProgrInter` and `IsRegrInter` defining which state sets have the format used in the left-hand-side of the rules **B2** and **B3**. Here separate cases are used to also allow the intersection $\bigcap_{L_{\mathbf{R}} \in \mathcal{L}} L_{\mathbf{R}}$ to be empty.

```
inductive IsLiteralUnion {n} (pt : STRIPS n) R [Formalism pt R] : States n → Prop
| single {S} : IsLiteral pt R S → IsLiteralUnion pt R S
| union {S S'} : IsLiteralUnion pt R S → IsLiteralUnion pt R S' →
  IsLiteralUnion pt R (S ∪ S')

inductive IsVariableInter {n} (pt : STRIPS n) R [Formalism pt R] : States n → Prop
| single {S} : IsVariable pt R S → IsVariableInter pt R S
| inter {S S'} : IsVariableInter pt R S → IsVariableInter pt R S' →
  IsVariableInter pt R (S ∩ S')

inductive IsProgrInter {n} (pt : STRIPS n) R [Formalism pt R] : States n → Prop
| empty {S A} : IsVariableInter pt R S → IsProgrInter pt R (pt.progression S A)
| inter {S S' A} : IsVariableInter pt R S → IsLiteralInter pt R S' →
  IsProgrInter pt R (pt.progression S A ∩ S')
```

2.2.3 ProofSystem

We start by defining dead states and dead state sets, as in Definition 18.

```
abbrev DeadState {n} (pt : STRIPS n) (s : State n) : Prop :=
  ∀ plan : Plan pt pt.init, s ∉ plan.path
abbrev Dead {n} (pt : STRIPS n) (S : States n) : Prop :=
  ∀ s ∈ S, DeadState pt s
```

Next we define a type for derivations in the proof system, based on Definition 20 and Definition 21. Below we only show a selection of the rules, the full definition can be found in Appendix A.1. We use a *shallow embedding*, meaning that existing types and functions in Lean (such as $\subseteq$ and $\cup$) are used for representing the syntax of the proof system. The alternative would be to use a *deep embedding*, where we would use custom types for representing the syntax. The latter is more powerful because it allows reasoning about the syntax, but it would require explicitly defining the syntax and its semantics, whereas for the shallow embedding we can use the existing semantics in Lean 4.

An element of `Derivation pt conclusion` is a proof tree in the proof system, where the conclusion of the proof tree is `conclusion`. For the basic rules the constructors require that the conclusion of the rule holds semantically. We use the inductive definitions from the previous section to limit the sets `S` and `S'` to the correct format for the rules **B1**-**B4**. The rules from basic set theory are polymorphic, so technically they are not restricted to sets of states or actions.

Note that `Derivation` uses `Type 1` instead of `Type` in its type declaration, which is the universe above `Type` is Leans type hierarchy. This is needed because the constructors for the rules **B1**-**B4** depend on the type `R` and those for the rules from basic set theory depend on the type α.

```
inductive Derivation {n} (pt : STRIPS n) : (conclusion : Prop) → Type 1
| B2 R [Formalism pt R] {S S'} :
  IsProgrInter pt R S →
  IsLiteralUnion pt R S' →
  (S ⊆ S') →
  Derivation pt (S ⊆ S')
| B5 (A A' : Actions n) : A ⊆ A' → Derivation pt (A ⊆ A')
| UD S S' :
  Derivation pt (Dead pt S) →
  Derivation pt (Dead pt S') →
  Derivation pt (Dead pt (S ∪ S'))
| PG S S' :
  Derivation pt (pt.progression S pt.actions ⊆ S ∪ S') →
  Derivation pt (Dead pt S') →
  Derivation pt (Dead pt (S ∩ pt.goal_states)) →
  Derivation pt (Dead pt S)
| CI : Derivation pt (Dead pt {pt.init}) → Derivation pt (Unsolvable pt)
| CG : Derivation pt (Dead pt pt.goal_states) → Derivation pt (Unsolvable pt)
| UR {α} (E E' : Set α) : Derivation pt (E ⊆ E ∪ E')
| PT S S' S'' A :
  Derivation pt (pt.progression S A ⊆ S'') →
  Derivation pt (S' ⊆ S) →
  Derivation pt (pt.progression S' A ⊆ S'')
...
```

We show that the proof system is sound by induction (or technically by recursion) on the rules of `Derivation`. For the basic rules the correctness follows immediately, since the conclusion of the rule is an argument of the constructors. Soundness of other rules (in particular **PG**, **PI**, **RG** and **RI**) is a bit more difficult, and the for these rules auxiliary lemmas are used (which we will not discuss here). We will have a closer look at the soundness proof for the rules **UD** and **CI**.

For **UD**, we first apply the tactic `simp`, which changes the goal into `∀ (s : State n), s ∈ S ∨ s ∈ S' → DeadState pt s`. The tactic `intro s hs` adds the state `s` and the hypothesis `s ∈ S ∨ s ∈ S'` to the environment en changes the goal into `DeadState pt s`. We distinguish cases based on `hs`, and in both cases we use the soundness of the subderivations `d1` and `d2` to change the goal in `s ∈ S` and `s ∈ S'` respectively. These goals correspond to hypotheses `hs'`, which we obtained after splitting cases based on `hs`.

For **CI** the tactic `constructor` is used to change the goal into `∀ (a : Plan pt pt.init), False`. By using the soundness of the subderivation we know that the initial state is dead. Then `simp` is used with the definition of `Dead` and the lemma `Path.first_mem` (which states that `s1 ∈ π` for every path `π : Path pt s1 s2`) to rewrite the hypothesis `h` into `∀ (a : Plan pt pt.init), False`. This can then be used to solve the goal.

```
theorem Derivation.soundness {n} {pt : STRIPS n} {conclusion} : (d : Derivation pt
    conclusion) → conclusion
| B2 _ _ _ h => h
| UD S S' d1 d2 =>
  by
    simp [Dead]
    intro s hs
    cases hs with
    | inl hs' =>
      apply d1.soundness
      exact hs'
    | inr hs' =>
      apply d2.soundness
      exact hs'
| PG S S' d1 d2 d3 =>
  by
    apply progression_goal
    · exact d1.soundness
    · exact d2.soundness
    · exact d3.soundness
| CI d =>
  by
    constructor
    have h : DeadState pt pt.init :=
      d.soundness pt.init (by simp)
    simp [DeadState, Path.first_mem] at h
    exact h
| UR E E' => by simp
...
```

For completeness, we first show that any inductive certificate `S` can be transformed to a derivation in the proof system. The derivation constructed in `Derivation.fromInductiveCertificate` corresponds to the proof tree in Figure 2.2. For the basic rules we use an instance `Formalism pt (States n)` defined in `Formalism`.

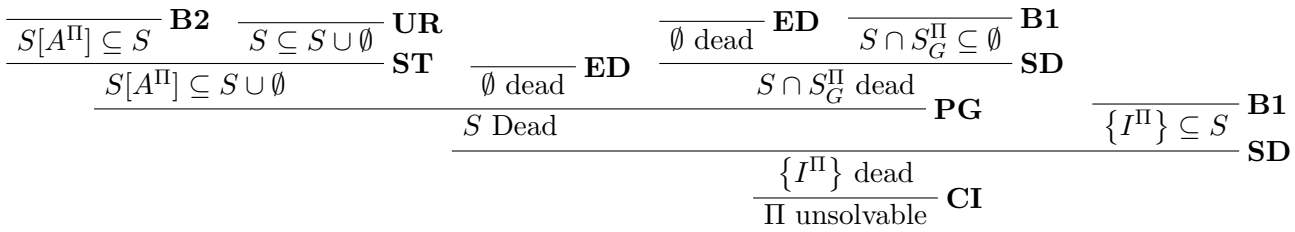


Figure 2.2: A derivation in the proof system that can be constructed given an inductive certificate S. The three conditions in Definition 15 are used for showing that the basic statements hold.

```
open IsLiteral IsVariable in
def Derivation.fromInductiveCertificate {n} {pt : STRIPS n} {S} :
  InductiveCertificate pt S → Derivation pt (Unsolvable pt)
| ⟨h1,h2,h3⟩ =>
  by
    apply CI
    · apply SD {pt.init} S
      · apply PG S ∅
        · apply ST (pt.progression S pt.actions) S (S ∪ ∅)
          · apply B2 (States n)
            · exact IsProgrInter.empty <| IsVariableInter.single <| explicit S
            · exact IsLiteralUnion.single <| pos <| explicit S
            · exact h3
          · exact UR S ∅
        · exact ED
        · apply SD (S ∩ pt.goal_states) ∅
          · exact ED
          · apply B1 (States n)
            · exact IsLiteralInter.inter
                (IsLiteralInter.single <| pos <| explicit S)
                (IsLiteralInter.single <| pos <| goal)
            · exact IsLiteralUnion.single <| pos <| empty
            · simp [Set.eq_empty_iff_forall_notMem, STRIPS.goal_states]
              exact h2
      · apply B1 (States n)
        · exact IsLiteralInter.single <| pos <| init
        · exact IsLiteralUnion.single <| pos <| explicit S
        · simp [h1]
```

To show that the proof system is complete, we need to prove that there is a derivation in the proof system if the planning task is unsolvable. By the reduction above it is sufficient to give an inductive certificate, and for this we use the completeness of inductive certificates which we showed in Section 2.2.1.

```
theorem completeness {n} {pt : STRIPS n} :
  Unsolvable pt → Nonempty (Derivation pt (Unsolvable pt)) :=
  by
    intro h
    constructor
    apply fromInductiveCertificate
    exact Classical.choose_spec (InductiveCertificate.completeness h)
```

2.3 State Set Formalisms

The folder `StateSetFormalism` implements functionality for working with the different formalisms discussed in Section 1.3.1. It contains the following files:

- `Formula` formalizes definitions for concepts like literals, clauses and CNF-formulas. It provides type classes for formulas over the variables `Fin n` and for the operations from Definition 13.
- `Bdd` will provide the instances needed for working with BDDs.
- `Horn` will implement Horn formulas and the operations needed for working with Horn formulas.
- `Mods` will implement the MODS formalism and the operations for working with MODS formulas.
- `Formalism` makes the connection between formulas and sets of states. It contains various definitions and lemmas that are used for the verification for the basic rules, like state set variables, state set literals and their unions and intersections.
- `StateSetFormalism` enables case distinctions on the different formalism, and provides wrapper functions for types in `Formalism`.

As mentioned in the introduction of this chapter, the formalisms themselves have not yet been implemented. For this reason we will only briefly look at the files `BDD`, `Horn` and `MODS`.

2.3.1 Formula

The file `Formula` starts by giving some definitions related to `VarSet'` (which we will not discuss), assignments of variables (`Model`) and partial assignments (`PartialModel`). The name (`Partial`)`Model` indicates that these types are mainly used to indicate which assignments satisfy a given formula. `Model` is only used at proof time, whereas `PartialModel` is also used at runtime in some operations and in MODS, which explains why `PartialModel` uses bit vectors and `Model` does not. Below, `PartialModel.models` gives all assignments to which the given partial assignment can be expanded.

```
abbrev Model n := Fin n → Prop
abbrev Models n := Set (Model n)

abbrev PartialModel {n} (V : VarSet' n) := BitVec V.val.length
def PartialModel.models {n} {V : VarSet' n} (M : PartialModel V) : Models n :=
  { M' | ∀ i : Fin V.val.length, M' ⟨V.val[i], by simp⟩ ↔ M[i] }
```

Next, literals are defined, where we use the pair `(i, true)` to represent the positive literal for the variable `i` and `(i, false)` for the negation of `i`. `Literal.models` defines the models of the given literal (i.e. the assignments making the literal true), and `Literals.mem_models` characterizes these models. After unfolding the definition of `Literal.models` using `simp`, the tactic `split` is used to split the goal into a goal for positive literals and one for negative literals, which can then both be solved using `simp`.

```
abbrev Literal n := Fin n × Bool

def Literal.models {n} : Literal n → Models n
| (i, true) => { M | M i }
| (i, false) => { M | ¬M i }
```

```
lemma Literal.mem_models {n} (l : Literal n) M : M ∈ l.models ↔ (M l.1 ↔ l.2) :=
  by
    simp [models]
    split
    all_goals simp
```

Similarly, clauses, cubes, CNF-formulas and DNF-formulas are defined, here we highlight the definition for clauses and models of clauses. The attribute `@[simp]` tells Lean that the lemma should be automatically used in the `simp` tactic if applicable.

```
abbrev Clause n := List (Literal n)

def Clause.models {n} (γ : Clause n) : Models n :=
  { M | ∃ l ∈ γ, M ∈ l.models }

@[simp]
lemma Clause.mem_models {n} (γ : Clause n) M :
  M ∈ γ.models ↔ ∃ l ∈ γ, M ∈ l.models :=
  by simp [Clause.models]
```

The type class `Formula` requires each type of formulas `R` to be equipped with two operations `vars`, which should return the variables of a given formula, and `models`, which should give the models for each formula. To ensure that an element φ of type `R` only depends on variables in `Formula.vars` φ, `models_equiv_right` asserts that if two assignments match on the variables in `Formula.vars` φ, then the second is a model of φ if the first one is. `Formula.models_equiv` turns this implication into a logical equivalence, which is often easier to use.

```
class Formula n (R : Type) where
  vars : (φ : R) → VarSet' n
  models : (φ : R) → Formula.Models n
  models_equiv_right (φ : R) (M M' : Formula.Model n) :
    (∀ i ∈ (vars φ).val, M i = M' i) → M ∈ models φ → M' ∈ models φ

lemma Formula.models_equiv {n} {R} [h : Formula n R] {φ : R} {M M' : Model n}
  (h1 : ∀ i ∈ (h.vars φ).val, M i = M' i) : M ∈ h.models φ ↔ M' ∈ h.models φ :=
  by ...
```

The main part of the file `Formula` are type classes for the operations of Definition 13. For each of the operations the type signature is provided and a statement of what it means for this operation to be correct. If a specific formalism efficiently supports an operation it should have an instance for the type class, where the first method contains the implementation for the specific formalism and the second on a proof that this specific implementation is correct. For example, `ClausalEntailment` requires a method `entails`, which given a formula φ and a clause γ returns whether $\varphi \models \gamma$. The field `entails_correct` ensures that `entails` behaves as we would expect, i.e. `entails` φ γ is true if all models of φ are also models of γ. Note that `Bot.bot_correct` requires that the set of variables associated with `bot` is empty, whereas `Top` does not require this for `top`. Later we want to use `Bot` for constructing the empty set of states (which is needed for the proof system, see Definition 19), and as we will see in Section 2.3.3 it can be important which variables are *not* used in a state set expression.

```
class Top n R [F : Formula n R] where
  top : R
  top_correct : F.models top = Set.univ

class Bot n R [F : Formula n R] where
  bot : R
  bot_correct : F.models bot = ∅ ∧ F.vars bot = VarSet'.empty

class ClausalEntailment n R [F : Formula n R] where
  entails : (φ : R) → (γ : Clause n) → Bool
  entails_correct {φ γ} : entails φ γ ↔ F.models φ ⊆ γ.models

class Implicant n R [F : Formula n R] where
  entails : (δ : Cube n) → (φ : R) → Bool
  entails_correct {δ φ} : entails δ φ ↔ δ.models  ⊆ F.models φ

class SententialEntailment n R [F : Formula n R] where
  entails : (φ ψ : R)  → Bool
  entails_correct {φ ψ} : entails φ ψ ↔ F.models φ ⊆ F.models ψ

class BoundedConjuction n R [F : Formula n R] where
  and : R → R → R
  and_correct {φ ψ} : F.models (and φ ψ) = F.models φ ∩ F.models ψ

class BoundedDisjunction n R [F : Formula n R] where
  or : R → R → R
  or_correct {φ ψ} : F.models (or φ ψ) = F.models φ ∪ F.models ψ
```

The methods `BoundedConjuction.and` and `BoundedDisjunction.or` are binary, but the classes also provide methods for obtaining the conjunction and disjunction of a list of formulas, as is shown below for `BoundedConjuction`. Note that an efficient implementation for `and` does not imply that `andList` is efficient. If for example the size of `and` φ ψ is of the order $\|\varphi\| \cdot \|\psi\|$, then the size of `andList l` can be exponential in the length of `l`.

```
namespace BoundedConjuction
def andList {n} {R} [Formula n R] [Top n R] [h : BoundedConjuction n R] : List R → R
| [] => Top.top n
| [φ] => φ
| φ :: ψ :: tail => h.and φ (h.andList (ψ :: tail))

lemma andList_correct {n} {R} [F : Formula n R] [Top n R] [h : BoundedConjuction n R] {l} :
  models (h.andList l) = { M | ∀ φ ∈ l, M ∈ F.models φ } :=
  by ...
end BoundedConjuction
```

The next two type classes correspond to the operations **CL** and $\mathbf{RN}_{\prec}$. From a theoretical point of view it would make most sense to implement **CL** in terms of `Cube`, but for efficiency we use `VarSet'`. Remember that `VarSet'` is defined as the subtype `{ vars : List (Fin n) // vars.Sorted (· < ·) }`,

which ensures that there are no duplicate variables and that the variables are ordered. Since we already represent sets of variables as `VarSet'` elsewhere, and in `Mods` we would convert the conjunction into a `PartialModel`, it makes most sense to implement the operation like this. For $\mathbf{RN}_{\prec}$, `rename` should replace the variables in `Formula.vars` φ by the variables in `vars'`, where the definition of `VarSet'` ensures that this substitution is monotone. The lemma `mem_rename_models` gives an alternative formulation for the correctness of `rename` in terms of `renameModel`. Currently, `rename_correct` is only used to prove `mem_rename_models`, so depending on which formulation is easier to prove when working with the specific formalisms, the former might be replaced by the latter.

```
class OfPartialModel n R [F : Formula n R] where
  ofPartialModel : (V : VarSet' n) → PartialModel V → R
  ofPartialModel_correct {V M} :
    F.models (ofPartialModel V M) = M.models ∧ F.vars (ofPartialModel V M) = V

class Renaming n R [F : Formula n R] where
  rename : (φ : R) → (vars' : VarSet' n) → vars'.val.length = (F.vars φ).val.length → R
  rename_correct {φ vars' h} :
    F.vars (rename φ vars' h) = vars' ∧
    F.models (rename φ vars' h) = { M | ∃ M' ∈ F.models φ,
      ∀ i : Fin vars'.val.length, M vars'.val[i] ↔ M' (F.vars φ).val[i] }

def Renaming.renameModel {n}
  (V V' : VarSet' n) (h : V'.val.length = V.val.length) (M : Model n) : Model n :=
  fun i ↦
    match V.val.finIdxOf? i with
    | none => M i
    | some j => M (V'.val[j]'(by omega))

lemma Renaming.mem_rename_models {n R} [F : Formula n R] [Renaming n R] {φ vars' h M} :
  M ∈ F.models (rename φ vars' h) ↔ renameModel (F.vars φ) vars' (by simp [h]) M ∈
    F.models φ :=
  by ...
```

The last two type classes are `ToCNF` and `ToDNF`, allowing us to convert a formula to an equivalent CNF- or DNF-formula. For the verification of **B1**-**B3** we don't just need to convert one formula to a CNF-formula, but a disjunction of formulas to one CNF-formula. This requires first converting every formula to a CNF-formula, and then converting the resulting disjunction of CNF-formulas into one CNF-formula using distributivity of $\vee$ over $\wedge$. Note that the size of the resulting formula can be exponential in the number of original formulas, motivating the restrictions on the number of conjuncts and disjuncts in Definition 20. In Lean this operation is implemented in `disjunctionToCNF`, and there is an analogous definition doing the same for converting a conjunction of formulas into one DNF-formula.

```
class ToCNF n R [F : Formula n R] where
  toCNF : R → CNF n
  toCNF_correct {φ} : (toCNF φ).models = F.models φ
```

```
class ToDNF n R [F : Formula n R] where
  toDNF : R → DNF n
  toDNF_correct {φ} : (toDNF φ).models = F.models φ

def ToCNF.disjunctionToCNF {n} {R} [Formula n R] [ToCNF n R] (l : List R) : CNF n :=
  (l.map toCNF).multiply

lemma ToCNF.disjunctionToCNF_correct {n} {R} [F : Formula n R] [h : ToCNF n R] {φs} :
  (disjunctionToCNF φs).models = { M | ∃ φ ∈ φs, M ∈ F.models φ } :=
  by ...
```

2.3.2 BDD, Horn and Mods

Currently, only the definition for the MODS formalism is finished, together with its instance for `Formula` and the one for `OfPartialModel`. A MODS-formula consists of a set of variables and a disjunction of partial models over these variables, which is represented by a list of the type `PartialModel` from page 34.

```
structure MODS n where
  vars : VarSet' n
  mods : List (PartialModel vars)
  deriving DecidableEq, Repr

namespace MODS
def models {n} (φ : MODS n) : Models n :=
  { M | ∃ M' ∈ φ.mods, M ∈ PartialModel.models M' }

instance {n} : Formula n (MODS n) where
  vars φ := φ.vars
  models := models
  models_equiv_right := by
    simp [models, PartialModel.models]
    grind
```

For `MODS`, `OfPartialModel` is one of the operations to that is easy to implement. Here, it serves as an example of what the instances should eventually look like. Most operations currently use `sorry`-placeholders, as is shown below for `SententialEntailment`.

```
instance {n} : OfPartialModel n (MODS n) where
  ofPartialModel V M := ⟨V, [M]⟩
  ofPartialModel_correct := by
    simp [Formula.models, models, Formula.vars]

instance {n} : SententialEntailment n (MODS n) where
  entails φ ψ := sorry
  entails_correct := sorry
```

We will briefly discuss how we plan to implement the other formalisms when discussing future work in Section 3.3.

2.3.3 Formalism

We will use the operations implemented in `Formula` to verify the rules **B1**-**B4**. The obvious way to represent a set of states for a planning task Π is to use a formula over the variables of the planning task V^Π. While this works for the rules **B1** and **B4**, we cannot represent the progression and regression needed in **B2** and **B3** this way. For these rules we additionally need primed versions of the variables in V^Π, and we need to be able to replace a given subset of variables in V^Π by their primed version. At the end of the section we will explain in more detail why we need this. Since the BDD-formalism only supports $\mathbf{RN}_\prec$ and not **RN** (see Table 1.1), we will use $2 \cdot i$ for representing the variable i, and $2 \cdot i + 1$ for representing its primed version. The file starts by defining these translations, which are shown below. We omit the lemmas about these definitions.

```
def Fin.toUnprimed {n} : Fin n → Fin (2 * n) :=
  fun i ↦ ⟨2 * i.val, by omega⟩
def Fin.toPrimed {n} (i : Fin (2 * n)) (h : Even i.val) : Fin (2 * n) :=
  ⟨i.val + 1, by grind⟩

namespace Validator.VarSet'
abbrev IsUnprimed {n} (V : VarSet' (2 * n)) : Prop :=
  ∀ i ∈ V.val, Even i.val
def toUnprimed {n} (V : VarSet' n) : VarSet' (2 * n) :=
  ...
def unprimedVars n : VarSet' (2 * n) :=
    let vars := List.ofFn Fin.toUnprimed
    have h : vars.Sorted (· < ·) := by
      simp [vars, List.Sorted, Fin.toUnprimed]
    ⟨vars, h⟩
end VarSet'
```

When translating a model to a state (which is implemented by `unprimedState` below), we look at the behaviour of the model on the unprimed variables. If the model assigns the value `True` to the unprimed variable corresponding to i (i.e. $2 \cdot i$), then i is in the state, and otherwise it is not. The lemma `exists_model_of_state` essentially shows that this translation is surjective.

```
namespace Formula.Model
def unprimedState {n} (M : Model (2 * n)) : State n :=
  { i | M i.toUnprimed }

lemma exists_model_of_state {n} s : ∃ M : Model (2 * n), s = M.unprimedState :=
  by
    let M : Model (2 * n) := fun i => ⟨i / 2, by omega⟩ ∈ s
    use M
    simp [M, Model.unprimedState, Fin.toUnprimed]
end Formula.Model
```

We finally get to the class `Formalism`, which we already used for the basic rules in `ProofSystem`. The class builds on the class `Formula (2 * n) R`, where the variables in `Formula` should be interpreted as described above. Additionally, it provides a method `toStates`, which by default takes the models of

the given formula, and transforms them to states using `Model.unprimedState`. While it is allowed to give a different implementation, `toStates_eq` enforces that this implementation is still equal to the default implementation.

```
class Formalism {n} (pt : STRIPS n) R extends Formula (2 * n) R where
  toStates (φ : R) : States n := (Formula.models φ).image Model.unprimedState
  toStates_eq (φ : R) : toStates φ = (Formula.models φ).image Model.unprimedState := by
    simp
```

We only override this default implementation for the following instance, which is used in Section 2.2.3 for `Derivation.fromInductiveCertificate`. The reason for overriding the default implementation is to allow that `Formalism.toStates S` reduces directly to `S` for any state set `S`.

```
@[simp]
instance {n} {pt : STRIPS n} : Formalism pt (States n) where
  vars _ := VarSet'.unprimedVars n
  models φ := { M | M.unprimedState ∈ φ }
  models_equiv_right := by
    ...
  toStates := id
  toStates_eq φ := by
    ext s
    have ⟨M, hM⟩ := Model.exists_model_of_state s
    simp
    grind
```

The largest part of the files consists of definitions and lemmas for state set variables, state set literals, and unions and intersections of state set literals and state set variables. The names of some of these definitions match the names in `Formula`, but they have difference namespaces. For example, the full name of the type `Literal` in `Formula` is `Validator.Formula.Literal`, whereas the full name of the type `Literal` which we will use for state set literals is `Validator.Formalism.Literal`. The definitions from `Formula` are only used for defining the different operations and implementing them for the different formalisms, whereas the types defined here are used for the verification of the basic rules (Definition 20). Below the keyword `variable` is used to declare `n`, `pt` and `R` implicit variables, which means that they will be implicitly added to type declarations if needed.

In contrast to Definition 19, where we had separate cases for the constant sets $\{I^\Pi\}$, S_G^Π and $\emptyset$, here state set variables only have one constructor for $\mathbf{R}$-formulas. If needed, constant sets will be transformed to $\mathbf{R}$-formulas, so they are also covered by this case.

```
namespace Formalism
variable {n} {pt : STRIPS n} {R}

abbrev Variable (pt : STRIPS n) (R : Type) [Formalism pt R] := R

namespace Variable
def models [Formalism pt R] : Variable pt R → Models (2 * n) :=
  Formula.models
```

```
abbrev vars [Formalism pt R] : Variable pt R → VarSet' (2 * n) :=
  Formula.vars

def toStates [Formalism pt R] : Variable pt R → States n :=
  Formalism.toStates pt

@[simp]
instance [F : Formalism pt R] : Membership (Fin (2 * n)) (Variable pt R) where
  mem x i := i ∈ x.vars.val
end Variable
```

For each of the concepts described in this file we will also define an unprimed version, which requires that all variables associated with the involved formulas are unprimed. This version is used to make the connection with the type `Model`, as illustrated by the lemma `mem_models_iff_of_eq_unprimedState`. Ironically, the lemma does not use `UnprimedVariable` but `Variable` combined with the hypothesis `x.vars.IsUnprimed`, because the statement `M ∈ x.val.models ↔ M' ∈ x.val.models` would be more difficult to use.

```
abbrev UnprimedVariable (pt : STRIPS n) (R : Type) [F : Formalism pt R] :=
  { x : Variable pt R // x.vars.IsUnprimed }

lemma UnprimedVariable.mem_models_iff_of_eq_unprimedState [Formalism pt R]
  {x : Variable pt R} {M M' : Model (2 * n)} :
  x.vars.IsUnprimed → M.unprimedState = M'.unprimedState →
  (M ∈ x.models ↔ M' ∈ x.models) := ...
```

Below are the definitions of `Literal`, `Variables`, `Literals`, and their unprimed versions. For `Literals` and `UnprimedLiterals` the first list contains all positive literals and the second all negative ones.

```
abbrev Literal (pt : STRIPS n) R [Formalism pt R] :=
  Variable pt R × Bool
abbrev UnprimedLiteral (pt : STRIPS n) R [Formalism pt R] :=
  UnprimedVariable pt R × Bool

abbrev Variables (pt : STRIPS n) R [Formalism pt R] :=
  List (Variable pt R)
abbrev UnprimedVariables (pt : STRIPS n) R [Formalism pt R] :=
  List (UnprimedVariable pt R)

abbrev Literals (pt : STRIPS n) R [Formalism pt R] :=
  Variables pt R × Variables pt R
abbrev UnprimedLiterals (pt : STRIPS n) R [Formalism pt R] :=
  UnprimedVariables pt R × UnprimedVariables pt R
```

The file contains to many definitions and lemmas to discuss them all, hence we will only highlight a few. For `Literal` there are methods for getting the models and the states, based on those of `Variable` we saw above. `Variables` and `Literals` have functions returning the states when interpreting the collection as a union or intersection, which are shown below for `Variables`. Note that these definitions

work with `Variable.models` and not `Variable.states`, since we also need to take the primed variables into account. For `UnprimedLiteral`, `UnprimedVariables` and `UnprimedLiterals` there are methods `val` for converting the types to `Literal`, `Variables` and `Literals` respectively. These methods implicitly use `Subtype.val` (which is used for `UnprimedVariable`) for the conversion.

```
def Variables.inter [F : Formalism pt R] (X : Variables pt R) : States n :=
  { s | ∃ M : Model (2 * n), M.unprimedState = s ∧ ∀ x ∈ X, M ∈ x.models }

def Variables.union [F : Formalism pt R] (X : Variables pt R) : States n :=
  { s | ∃ M, M.unprimedState = s ∧ ∃ x ∈ X, M ∈ x.models }

def UnprimedVariables.val [Formalism pt R] : UnprimedVariables pt R → Variables pt R :=
  fun X ↦ X
```

There are multiple lemmas for working with these unions and intersections, below we show some of those for `Variables`. For both of these lemmas, one of the lists of variables needs to be unprimed. In the first case the left-hand-side of the equivalence talks about states, and the right-hand-side about models. Because `X2` is unprimed, the value of the primed variables of `M` is irrelevant for deciding whether `∃ x ∈ X2, M ∈ x.val.models`, which is crucial for the lemma to hold.

Similarly, suppose that both `X1` and `X2` would have type `Variable` in the second lemma. Then for every state `s` in `X1.inter ∩ X2.inter`, there are models `M1` and `M2` with `M1.unprimedState = s`, `M2.unprimedState = s`, `M1 ∈ x.models` for all variables `x ∈ X1` and `M2 ∈ x.models` for all variables `x ∈ X2`. However, these models `M1` and `M2` could differ on the primed variables, and therefore `s ∉ (X1 ++ X2).inter`. If `X2` is unprimed, the lemma `mem_models_iff_of_eq_unprimedState` we discussed on page 41 can be used to conclude from `M1.unprimedState = M2.unprimedState` that `M1 ∈ x.val.models` for all `x ∈ X2`, and therefore `s ∈ (X1 ++ X2.val).inter`.

```
lemma UnprimedVariables.inter_subset_union_iff_models [F : Formalism pt R]
  (X1 : Variables pt R) (X2 : UnprimedVariables pt R) :
  X1.inter ⊆ X2.val.union ↔
    (∀ M, (∀ x ∈ X1, M ∈ F.models x) → ∃ x ∈ X2, M ∈ x.val.models) :=
  by ...

@[simp]
lemma UnprimedVariables.inter_variables_append [Formalism pt R]
  {X1 : Variables pt R} {X2 : UnprimedVariables pt R} :
  (X1 ++ X2.val).inter = X1.inter ∩ X2.val.inter :=
  by ...
```

Unions and intersections of `UnprimedLiterals` can be stated in terms of `UnprimedVariables` using the lemmas below. Again these lemmas don't work for `Literals`, for similar reasons as the previous lemmas.

```
@[simp low]
lemma UnprimedLiterals.union_val [Formalism pt R] {L : UnprimedLiterals pt R} :
  L.val.union = L.1.val.union ∪ L.2.val.interᶜ :=
  by ...
```

```
@[simp low]
lemma UnprimedLiterals.inter_val [Formalism pt R] {L : UnprimedLiterals pt R} :
  L.val.inter = L.1.val.inter ∩ L.2.val.union^c :=
  by ...
```

As mentioned in the introduction of this section, we need to be able to replace the unprimed variables corresponding to a given set of variables with their primed version. The definitions below define this using the `Renaming` class from page 37. The lemma `mem_inter_toPrimed` already indicates how `UnprimedVariables.toPrimed` will be used when verifying **B2** and **B3**. Recall from Section 2.1.2 that the progression of a state set `S` through an action `a` is formalized as `{ s | ∃ s' ∈ S, Successor a s' s }`. The primed variables are used to represent the variables in `s'` where `s` and `s'` differ (or potentially differ), corresponding to the variables in `V` below. For the variables where `s` and `s'` are guaranteed to be the same (i.e. the variables not in `V`), we use the unprimed variables.

```
def UnprimedVariable.toPrimed [Formalism pt R] [Renaming (2 * n) R]
  (x : UnprimedVariable pt R) (V : VarSet' n) : Variable pt R := ...

def UnprimedVariables.toPrimed [F : Formalism pt R] [Renaming (2 * n) R]
  (X : UnprimedVariables pt R) (V : VarSet' n) : Variables pt R :=
  X.map (UnprimedVariable.toPrimed · V)

lemma UnprimedVariables.mem_inter_toPrimed [F : Formalism pt R] [Renaming (2 * n) R]
  {X : UnprimedVariables pt R} {V s} : s ∈ (toPrimed X V).inter ↔
    ∃ s' ∈ X.val.inter, ∀ i ∉ V.val, i ∈ s' ↔ i ∈ s :=
  by ...
```

2.3.4 StateSetFormalism

For the verification of the basic rules we need to be able to make a case distinction on the formalism that is used to represent the formula. It is not possible to make a case distinction based on which types have an instance for `Formalism`, hence we define an inductive type `StateSetFormalism` for representing the different options. The method `type` is used to link the different options with the actual type used for representing the state sets.

```
inductive StateSetFormalism
| bdd
| horn
| mods
  deriving DecidableEq

abbrev StateSetFormalism.type {n} (_ : STRIPS n) : StateSetFormalism → Type
| bdd => BDD (2 * n)
| horn => Horn (2 * n)
| mods => MODS (2 * n)
```

The file contains instances of `Formalism` for `BDD`, `Horn`, and `MODS`. Since all these types already have instances for `Formula`, and the additional methods from `Formalism` have default implementations, these instances follow immediately. Next, there is a wrapper instance for `Formalism pt (R.type pt)`. There are similar wrapper instances for the operations `Bot`, `OfPartialModel` and `Renaming` from Section 2.3.1, since these operations are needed in places where it is not necessary to perform a case distinction on the formalisms.

```
instance BDD.instFormalism {n} {pt : STRIPS n} : Formalism pt (BDD (2 * n)) where

instance Horn.instFormalism {n} {pt : STRIPS n} : Formalism pt (Horn (2 * n)) where

instance MODS.instFormalism {n} {pt : STRIPS n} : Formalism pt (MODS (2 * n)) where

instance {n} {pt : STRIPS n} : {R : StateSetFormalism} → Formalism pt (R.type pt)
| bdd => BDD.instFormalism
| horn => Horn.instFormalism
| mods => MODS.instFormalism
```

Furthermore, the file contains wrapper types around the types from the previous section, whose purpose is making the type signatures in Section 2.4.6 less convoluted.

```
variable {n} (pt : STRIPS n) (R : StateSetFormalism)
abbrev UnprimedVariable' :=
  UnprimedVariable pt (R.type pt)
abbrev UnprimedLiteral' :=
  UnprimedLiteral pt (R.type pt)
abbrev Variables' :=
  Variables pt (R.type pt)
abbrev UnprimedVariables' :=
  UnprimedVariables pt (R.type pt)
abbrev UnprimedLiterals' :=
  UnprimedLiterals pt (R.type pt)
```

`StateSetFormalism` concludes with functions to construct the unprimed variables corresponding to $\emptyset$, `{pt.init}` and `pt.goal_states`. The variables for `{pt.init}` and `pt.goal_states` are constructed using the operation `OfPartialModel`, and the one for $\emptyset$ is constructed using `Bot`, as shown below.

```
def mkEmpty (pt : STRIPS n) R : UnprimedVariable' pt R :=
  ⟨bot (2 * n), by simp [bot_correct]; exact VarSet'.isUnprimed_empty⟩

@[simp]
lemma toStates_mkEmpty (pt : STRIPS n) R : (mkEmpty pt R).val.toStates = ∅ :=
  by
    simp [mkEmpty, Variable.toStates_eq, Variable.models,  bot_correct]
```

2.4 Certificates and their Validation

The validator itself can be divided into the following five parts:

1. A structure for representing the certificate.
2. A parser to parse the certificate into this representation.
3. A low-level definition of what it means for the certificate to be valid.
4. A validator which checks whether the stored certificate satisfies the definition of validity. Running the validator should either result in a reason why the certificate is not valid, or in a proof that it is valid.
5. A translation from a proof that a certificate is valid to the `Derivation` type discussed in Section 2.2.3. From the soundness of `Derivation` it follows that an accepted certificate implies that the corresponding planning task is unsolvable.

These parts are implemented in the different files of the folder `Certificate`:

- `Certificate` defines the representation of the certificates.
- `Parser` contains a parser for parsing the certificates.
- `SetExpr` defines what it means for action set expressions and state set expressions to be valid, and it implements functions for verifying whether they are valid.
- `Constraint` defines constraints, which can be used to define what it means for knowledge to be valid and to verify whether a given piece of knowledge is valid. It also gives various constraints that will be used for the syntactic rules.
- `BasicRules` defines the constraints for the basic rules.
- `ValidCertificate` gives the constraints for the syntactic rules and defines what it means for a certificate to be valid.
- `ToDerivation` gives a translation from a proof that a certificate is valid to an element of `Derivation`.

Finally, the main file `Validator` combines the STRIPS parser, the parser for the certificates and the verification of the certificates into one validator.

2.4.1 Certificate

Each line of the certificate is either an action set, a state set or a piece of knowledge. Each line is assigned a natural number as its ID, which is unique in its category (so an action set can have the same ID as a state set, but no two state sets have the same ID). Naturally such a certificate can be stored in three arrays (Helve uses deques in C++), where each type of statement has its own array, and the IDs correspond with the indices in the arrays. In Helve there are no restrictions on the indices (only that each ID is unique for the given category), but we will require that the IDs for each category start with 0 and increment by one. In Helve, it was sufficient to test whether an ID is defined every

time an element of the array is accessed, but for the definition of validity we also need to have a proof that certain indices are valid. Without the additional requirement it would be more difficult to do this.

The file starts by defining action set expressions and state set expressions following the input format defined in Helve, which is similar to the types defined in Definition 19. Below, `ActionSetExpr.enum` uses a list of references to actions in the planning task `pt` (without bounds proof), instead of one single action like in Definition 19. The arguments in `ActionSetExpr.union`, and the second arguments of `StateSetExpr.progr` and `StateSetExpr.regr` are action set IDs. The other natural numbers in `StateSetExpr` are state set IDs.

```
inductive ActionSetExpr : Type
| enum : List ℕ → ActionSetExpr
| union : ℕ → ℕ → ActionSetExpr
| all : ActionSetExpr

inductive StateSetExpr {n} (pt : STRIPS n) : Type
| empty : StateSetExpr pt
| init : StateSetExpr pt
| goal : StateSetExpr pt
| bdd : UnprimedVariable pt (BDD (2 * n)) → StateSetExpr pt
| horn : UnprimedVariable pt (Horn (2 * n)) → StateSetExpr pt
| mods : UnprimedVariable pt (MODS (2 * n)) → StateSetExpr pt
| neg : ℕ → StateSetExpr pt
| inter : ℕ → ℕ → StateSetExpr pt
| union : ℕ → ℕ → StateSetExpr pt
| progr : ℕ → ℕ → StateSetExpr pt
| regr : ℕ → ℕ → StateSetExpr pt
```

For the knowledge statements we use separate inductive definitions for each kind of knowledge. This makes it easier to check whether a piece of knowledge is for example dead knowledge, and to assign a semantic meaning to the rules, as this can then be defined for each kind of knowledge. Note that in contrast to `Derivation` (see Section 2.2.3), the rules from basic set theory have been split in separate rules for state sets and action sets. Since intersections are not defined for action set expressions, not all of these rules have a variant for action sets. The natural number in the type signature of `DeadKnowledge` is the ID of the state set that is dead, and the natural numbers in `StateSubsetKnowledge`/`ActionSubsetKnowledge` are the IDs of the state/action set expressions E and E' in the corresponding statement $E \subseteq E'$. The natural numbers after the colons are the knowledge IDs of the premises of the rule. From now on, we will use $\texttt{A}_i$, $\texttt{S}_i$ and $\texttt{K}_i$ (and their variations) for denoting actions set IDs, state set IDs and knowledge IDs respectively.

```
inductive DeadKnowledge : ℕ → Type
| ED Sᵢ : DeadKnowledge Sᵢ
| UD Sᵢ: ℕ → ℕ → DeadKnowledge Sᵢ
| SD Sᵢ: ℕ → ℕ → DeadKnowledge Sᵢ
| PG Sᵢ: ℕ → ℕ → ℕ → DeadKnowledge Sᵢ
| PI Sᵢ: ℕ → ℕ → ℕ → DeadKnowledge Sᵢ
| RG Sᵢ: ℕ → ℕ → ℕ → DeadKnowledge Sᵢ
| RI Sᵢ: ℕ → ℕ → ℕ → DeadKnowledge Sᵢ
```

```
inductive StateSubsetKnowledge : ℕ → ℕ → Type
| B1 S_i S'_i : StateSubsetKnowledge S_i S'_i
| B2 S_i S'_i : StateSubsetKnowledge S_i S'_i
| B3 S_i S'_i : StateSubsetKnowledge S_i S'_i
| B4 S_i S'_i : StateSubsetKnowledge S_i S'_i
| URS S_i S'_i : StateSubsetKnowledge S_i S'_i
| ULS S_i S'_i : StateSubsetKnowledge S_i S'_i
| IRS S_i S'_i : StateSubsetKnowledge S_i S'_i
| ILS S_i S'_i : StateSubsetKnowledge S_i S'_i
| DIS S_i S'_i : StateSubsetKnowledge S_i S'_i
| SUS S_i S'_i : ℕ → ℕ → StateSubsetKnowledge S_i S'_i
| SIS S_i S'_i : ℕ → ℕ → StateSubsetKnowledge S_i S'_i
| STS S_i S'_i : ℕ → ℕ → StateSubsetKnowledge S_i S'_i
| AT S_i S'_i : ℕ → ℕ → StateSubsetKnowledge S_i S'_i
| AU S_i S'_i : ℕ → ℕ → StateSubsetKnowledge S_i S'_i
| PT S_i S'_i : ℕ → ℕ → StateSubsetKnowledge S_i S'_i
| PU S_i S'_i : ℕ → ℕ → StateSubsetKnowledge S_i S'_i
| PR S_i S'_i : ℕ → StateSubsetKnowledge S_i S'_i
| RP S_i S'_i : ℕ → StateSubsetKnowledge S_i S'_i

inductive ActionSubsetKnowledge : ℕ → ℕ → Type
| B5 A_i A'_i : ActionSubsetKnowledge A_i A'_i
| URA A_i A'_i : ActionSubsetKnowledge A_i A'_i
| ULA A_i A'_i : ActionSubsetKnowledge A_i A'_i
| SUA A_i A'_i : ℕ → ℕ → ActionSubsetKnowledge A_i A'_i
| STA A_i A'_i : ℕ → ℕ → ActionSubsetKnowledge A_i A'_i

inductive UnsolvableKnowledge
| CI : ℕ → UnsolvableKnowledge
| CG : ℕ → UnsolvableKnowledge

inductive Knowledge
| dead S_i : DeadKnowledge S_i → Knowledge
| actionSubset A_i A'_i : ActionSubsetKnowledge A_i A'_i → Knowledge
| stateSubset S_i S'_i : StateSubsetKnowledge S_i S'_i → Knowledge
| unsolvable : UnsolvableKnowledge → Knowledge
```

The certificate itself is defined as a structure containing three arrays with the action set expressions, state set expressions and pieces of knowledge, as described in the beginning of the section.

```
structure Certificate {n} (pt : STRIPS n) where
  actions : Array ActionSetExpr
  states : Array (StateSetExpr pt)
  knowledge : Array Knowledge
```

2.4.2 Parser

Each line of the certificate is expected to either be an action set expression, a state set expression, a piece of knowledge or a comment (starting with #). The format for the state set expressions, action set expression and knowledge closely resembles the definitions from the previous section, therefore we will not discuss them in detail here, but a description can be found in Appendix A.2. Note that the parsers for BDDs, Horn formulas and MODS have not yet been implemented.

2.4.3 SetExpr

We start with the definitions for validity of `ActionSetExpr` and `StateSetExpr`. This involves mostly stating that the mentioned action set IDs and state set IDs are within bounds of the certificate `C`. To ensure that the certificate does not contain cyclic dependencies, we enforce the stronger condition that action sets can only reference action sets with a lower ID, and similar for state sets. This works since the action sets never reference state sets, so cyclic references between action sets and state sets are not possible.

```
def Certificate.validActionSetExpr (C : Certificate pt) (Aᵢ : Fin C.actions.size) : Prop :=
  match C.actions[Aᵢ] with
  | ActionSetExpr.enum as => ∀ a ∈ as, a < pt.actions'.length
  | ActionSetExpr.union A'ᵢ A''ᵢ => A'ᵢ < Aᵢ ∧ A''ᵢ < Aᵢ
  | ActionSetExpr.all => True

def Certificate.validStateSetExpr (C : Certificate pt) (Sᵢ : Fin C.states.size) : Prop :=
  match C.states[Sᵢ] with
  | StateSetExpr.empty => True
  | StateSetExpr.init => True
  | StateSetExpr.goal => True
  | StateSetExpr.bdd φ => True
  | StateSetExpr.horn φ => True
  | StateSetExpr.mods φ => True
  | StateSetExpr.neg S'ᵢ => S'ᵢ < Sᵢ
  | StateSetExpr.inter S'ᵢ S''ᵢ => S'ᵢ < Sᵢ ∧ S''ᵢ < Sᵢ
  | StateSetExpr.union S'ᵢ S''ᵢ => S'ᵢ < Sᵢ ∧ S''ᵢ < Sᵢ
  | StateSetExpr.progr S'ᵢ Aᵢ => S'ᵢ < Sᵢ ∧ Aᵢ < C.actions.size
  | StateSetExpr.regr S'ᵢ Aᵢ => S'ᵢ < Sᵢ ∧ Aᵢ < C.actions.size
```

For checking whether the action and state set expressions are valid, we need the following two types which are defined in the file `Error`. `Result` either contains an error of type `Error` (which is also defined in the file `Error`), or a subtype containing a value `a` of type α and a proof that `p a` holds. The type can be used if we want to return some data (for example an ID) during verification, while simultaneously ensuring that the property `p` hold for this data (for example that the given ID is within bounds). The type `Result'` can be used if we only want to verify some property, without returning data. Here `Unit` is a type which contains only one element `()`.

```
abbrev Result.{u} (α : Type u) (p : α → Prop) :=
  Except Error { a // p a }
abbrev Result' (p : Prop) :=
  Result Unit (fun _ ↦ p)
```

The file `SetExpr` contains various definitions for verifying bounds and throwing error messages, which we will not discuss. Below are two other definitions, which are used to combine verification of different properties and to verify the validity of `ActionSetExpr.enum`. For `verify_and` we use that `Except` (and therefore `Result'`) is a monad, which enables the convenient `do`-notation.

```
def verify_and {p1 p2} (res1 : Result' p1) (res2 : Result' p2) : Result' (p1 ∧ p2) := do
  let ⟨(), h1⟩ ← res1
  let ⟨(), h2⟩ ← res2
  return ⟨(), And.intro h1 h2⟩

def verifyActionsEnum (pt : STRIPS n) (as : List ℕ) : Result' (∀ a ∈ as, a <
    pt.actions'.length) :=
    if h : ∀ a ∈ as, a < pt.actions'.length then
      return ⟨(), h⟩
    else
      throwUnvalid s!"Not all given actions ids exist in the planning task.\
      The following action ids are out of bound:\n{as.filter (· < pt.actions'.length)}"
```

These functions are then combined to verify that the actions set expression with the given ID is valid, as shown below. Here `verify_action_bounds` A_i A'_i has type `Result' (A'`$_i$ `<` A_i`)`. The verification of the state set expressions is similar.

```
def Certificate.verifyActionSetExpr (C : Certificate pt) (A_i : Fin C.actions.size) :
  Result' (C.validActionSetExpr A_i) :=
  by
    unfold validActionSetExpr
    cases C.actions[A_i] with
    | enum as => exact verifyActionsEnum pt as
    | union A'_i A''_i =>
        exact verify_and (verify_action_bounds A_i A'_i) (verify_action_bounds A_i A''_i)
    | all => exact pure ⟨(), True.intro⟩
```

Next we use the definitions of validity of action set expressions and state set expressions to define an intermediate version of validity for certificates, where only the set expressions need to be valid.

```
structure validSets (C : Certificate pt) : Prop where
  validActions : ∀ A_i, C.validActionSetExpr A_i
  validStates : ∀ S_i, C.validStateSetExpr S_i
```

This definition can then be used to construct the set of actions or states for a given state set ID or action set ID. These constructions will be needed to semantically define what it means for the basic rules to be valid, and in Section 2.4.8 to translate the certificates to `Derivation`. Below we show a selection of the cases in the definition. For the formulas in the formalisms we use the method `Variable.toStates` from Section 2.3.3. The `have` expressions with the bounds are used to show termination of the function, and by the tactic `omega` for proving that the IDs are valid.

```
def getStates {C : Certificate pt} (hC : C.validSets) (S_i : Fin C.states.size) : States n :=
  have h := hC.validStates S_i
  match heq : C.states[S_i] with
  | StateSetExpr.empty => ∅
  | StateSetExpr.init => {pt.init}
  | StateSetExpr.goal => pt.goal_states
  | StateSetExpr.bdd φ => φ.val.toStates
  | StateSetExpr.union S'_i S''_i =>
    have : S'_i < S_i ∧ S''_i < S_i := by
      simp_all [Certificate.validStateSetExpr]
    hC.getStates ⟨S'_i, by omega⟩ ∪ hC.getStates ⟨S''_i, by omega⟩
  | StateSetExpr.progr S'_i A_i =>
    have h' : S'_i < S_i ∧ A_i < C.actions.size := by
      simp_all [Certificate.validStateSetExpr]
    pt.progression (hC.getStates ⟨S'_i, by omega⟩) (hC.getActions ⟨A_i, h'.2⟩)
  ...
```

In contrast to the state set expressions, the variant for action set expressions is also used at runtime to verify the rule **B5**. For this reason we first define a version `getActions'` which collects the action IDs (i.e. the indices of the actions in `pt.actions'`). This version is used at runtime. `getActions` translates this list of action IDs to a set of actions, and will only be used at proof time.

```
abbrev ActionIds (pt : STRIPS n) :=
  List (Fin pt.actions'.length)
abbrev ActionIds.toActions (A : ActionIds pt) : Actions n :=
  (A.map (pt.actions'[·])).toFinset

def getActions' {C : Certificate pt} (hC : C.validSets)
  (A_i : Fin C.actions.size) : ActionIds pt :=
  ...
def getActions {C : Certificate pt} (hC : C.validSets) (A_i : Fin C.actions.size) :
  Actions n :=
  (getActions' hC A_i).toActions
```

For all constructors except for `ActionSetExpr.enum`, the file also gives lemmas for simplifying these definitions, below we highlight the one for `StateSetExpr.union`. The tactic `split` makes a separate goal for each of cases in the definition `getStates` and adds a hypothesis stating that `C.states[S_i]` is matches the corresponding case. For all constructors except `union` this leads to a contradiction with `h`, and for `union` the two sides of the equality in the goal match.

```
lemma getStatesUnion {C : Certificate pt} (hChypothesis : C.validSets)
  (S_i S'_i S''_i : Fin C.states.size)
  (h : C.states[S_i]? = some (StateSetExpr.union S'_i S''_i)) :
  hC.getStates S_i = hC.getStates S'_i ∪ hC.getStates S''_i :=
  by
    rw [getStates]
    split
    all_goals simp_all
```

2.4.4 Naive Attempt for Verifying the Syntactic Rules

One of the earlier attempts for verifying the syntactic rules uses the same approach as the action and state set expressions. As an example, the definition of validity for the rule **UD** is shown below. Note that in `validUD` we use existential quantifiers to assert that the state set expression `C.states[S1ᵢ]` is constructed using the constructor `StateSetExpr.union`. The same approach is used to express that `C.knowledge[K1ᵢ]` and `C.knowledge[K2ᵢ]` are pieces of dead knowledge stating that the state set expressions with IDs `Sᵢ` and `S'ᵢ` are dead. The hypotheses `hK1ᵢ` and `hK2ᵢ` are used implicitly for the array accesses `C.knowledge[K1ᵢ]` and `C.knowledge[K2ᵢ]`.

```
abbrev validUD (C : Certificate pt) (Kᵢ : Fin C.knowledge.size) (S1ᵢ K1ᵢ K2ᵢ : ℕ) : Prop :=
  K1ᵢ < Kᵢ ∧ K2ᵢ < Kᵢ ∧ ∃ hK1ᵢ : K1ᵢ < Kᵢ, ∃ hK2ᵢ : K2ᵢ < Kᵢ,
  ∃ Sᵢ S'ᵢ, C.states[S1ᵢ]? = StateSetExpr.union Sᵢ S'ᵢ ∧
  ∃ K1 : DeadKnowledge Sᵢ, C.knowledge[K1ᵢ] = Knowledge.dead K1 ∧
  ∃ K2 : DeadKnowledge S'ᵢ, C.knowledge[K2ᵢ] = Knowledge.dead K2
```

For verifying whether a piece of **UD**-knowledge is valid, we first verify the bounds, then we verify that the given state set expression is a union using `verify_states_union`, and finally we check that the two premises are correct using `verify_knowledge_dead` (which is not shown here). In `verify_states_union` we use the `Result` type defined on page 48 to return the IDs `S'ᵢ` and `S''ᵢ` of the left and right state set expression, together with a proof that `C.states[Sᵢ]? = StateSetExpr.union S'ᵢ S''ᵢ`.

```
def verify_states_union (C : Certificate pt) (Sᵢ : ℕ) :
  Result (ℕ × ℕ) (fun (S'ᵢ, S''ᵢ) ↦ C.states[Sᵢ]? = StateSetExpr.union S'ᵢ S''ᵢ) :=
  match C.states[Sᵢ]? with
  | some (StateSetExpr.union S'ᵢ S''ᵢ) => pure ⟨(S'ᵢ, S''ᵢ), by simp⟩
  | some S => throw s!"The state set #{Sᵢ} is expected to be a union, but it is {S}."
  | none => throw (state_out_of_bounds Sᵢ)

def verifyKnowledge (C : Certificate pt) (Kᵢ : Fin C.knowledge.size) :
  Result' (C.validKnowledge Kᵢ) :=
  by
    unfold validKnowledge
    cases heq : C.knowledge[Kᵢ] with
    | @dead Sᵢ K =>
      apply verify_and (verify_state_bounds C.states.size Sᵢ)
      cases K with
      | UD K'ᵢ K''ᵢ => exact do
        let ⟨(), h1⟩ ← verify_knowledge_bounds Kᵢ K'ᵢ
        let ⟨(), h2⟩ ← verify_knowledge_bounds Kᵢ K''ᵢ
        let ⟨(S'ᵢ, S''ᵢ), h3⟩ ← verify_states_union C Sᵢ
        let ⟨(), h4⟩ ← verify_knowledge_dead C ⟨K'ᵢ, by simp at h1; omega⟩ S'ᵢ
        let ⟨(), h5⟩ ← verify_knowledge_dead C ⟨K''ᵢ, by simp at h2; omega⟩ S''ᵢ
        return ⟨(), by simp_all⟩
      ...
    ...
```

The existential quantifiers inside the definition of `validUD` pose an additional challenge, because we will have to eliminate them when transforming the proof of validity into a `Derivation` in Section 2.4.8.

This can be done by converting the existential quantification into a subtype, as shown below for `StateSetExpr.union`. In the proof we make a case distinction on `C.states[S_i]`, and for all cases except `union` this leads to a contradiction with the given hypothesis `h`, since elements constructed using different constructors cannot be equal.

```
def stateUnionOfExists (C : Certificate pt) (S_i : Fin C.states.size)
  (h : ∃ S'_i S''_i, C.states[S_i]? = StateSetExpr.union S'_i S''_i) :
  {p : ℕ × ℕ // C.states[S_i] = (StateSetExpr.union p.1 p.2)} := by
  cases h : C.states[S_i]
  all_goals simp_all
  case union S'_i S''_i =>
  exact ⟨(S'_i, S''_i), by simp⟩
```

While the approach works for **UD**, eliminating the existential quantifiers becomes a lot harder for more complicated syntactic rules (e.g. **PG**), which need nested existential quantifiers. There was a brief attempt with writing custom tactics to eliminate the existential quantifiers, but without much success. Ultimately, a more elegant solution was chosen, which is discussed in the next section.

2.4.5 Constraint

Instead of writing separate definitions for validity, verification and elimination of the existential quantifiers, we can also group these definitions into one structure. While we still need to implement the three operations separately on a low level (like checking bounds, or checking that the state set expression of a given ID is a union), we will only need to combine the different components once for each rule. Furthermore, the elimination of the existential quantifiers becomes a lot more manageable.

The structure `Constraint` shown below provides this functionality. The parameter α in the type signature is the type of the data that the constraint should return. For example, in the case of checking that the state set expression with a given ID is a union, this would be $\mathbb{N} \times \mathbb{N}$, for the IDs of the left and right state sets. The field `prop` contains the proposition that the returned data should satisfy, and `verify'` specifies how this should be checked. When actually verifying the certificate the method `verify` is used, which gives an optional error message if the constraint is not valid. Validity is defined by `valid`, and the field `elim_exists` is used to turn the existential quantifier in `valid` into a subtype. The additional requirement `∀ a', prop a' → a' = a` states that there exists only one element of type α satisfying the requirement, which is needed for combining constraints.

```
structure Constraint (α : Type) : Type where
  prop : α → Prop
  verify' :  Result α prop
  elim_exists :
    (∃ a, prop a) → {a // prop a ∧ ∀ a', prop a' → a' = a}
  message : Option String := none

namespace Constraint
abbrev valid {α} (h : Constraint α) : Prop :=
  ∃ a, h.prop a
def verify {α} (h : Constraint α) : Result α (h.prop) :=
  withErrorMessage h.message h.verify'
```

The constraint for checking whether the state set with ID `S`$_i$ is a union of states is shown below. Here `elim_exists_2` is a macro for eliminating existential quantifiers when the constructor for the input argument `C.states[S`$_i$`]?` has two arguments. The functions for error handling are defined in the file `StateSetExpr`, but we did not discuss them in more detail.

```
def isStateUnion (C : Certificate pt) (S_i : ℕ) : Constraint (ℕ × ℕ) where
  prop := fun (S'_i, S''_i) ↦ C.states[S_i]? = some (StateSetExpr.union S'_i S''_i)
  verify' :=
    match C.states[S_i]? with
    | some (StateSetExpr.union S'_i S''_i) => pure ⟨(S'_i, S''_i), by simp⟩
    | some S => StateSetConstraint.throw_unexpected S_i "a union of states" S
    | none => StateSetConstraint.throw_out_of_bounds S_i
  elim_exists := by elim_exists_2 C.states[S_i]?
  message := s!"Verifying that state set #{S_i} is the union of two state sets"
```

There are similar constraints for `ActionSetExpr.all`, `ActionSetExpr.union`, and all constructors of `StateSetExpr` except `bdd`, `horn` and `mods`. Additionally, there are constraints for dead knowledge and for the two types of subset knowledge. Below we show the constraint for verifying that a given knowledge ID corresponds to dead knowledge. The argument `K` of `dead S`$_i$ `K` is not returned because it is irrelevant for the verification. For example, in **UD** we don't need to know which rule is used to deduce that S is dead. For the elimination of the existential quantifier, we first make a case distinction to get rid of the `Option` wrapper introduced by the `_[_]?` notation, and then we make a case distinction on the knowledge inside the wrapper. Since elements constructed by different constructors are never equal, after simplification only the option corresponding to the constructor `dead` remains, and then we can return its argument `S`$_i$.

```
def isDeadKnowledge (C : Certificate pt) (K_i : ℕ) : Constraint ℕ where
  prop := fun S_i ↦ ∃ K, C.knowledge[K_i]? = dead S_i K
  verify' :=
    match C.knowledge[K_i]? with
    | some (dead S_i K) => return ⟨S_i, by simp⟩
    | some K => KnowledgeConstraint.throw_unexpected K_i "dead knowledge" K
    | none => KnowledgeConstraint.throw_out_of_bounds K_i
  elim_exists h := by
    cases heq : C.knowledge[K_i]? with
    | some K =>
      cases K
      all_goals simp_all only [Option.some.injEq, reduceCtorEq, exists_false]
      rename_i S_i K
      exact ⟨S_i, by simp_all⟩
    | none => simp_all
  message := s!"Verifying that knowledge #{K_i} is dead-knowledge"
```

The main benefit of the `Constraint` type is that it easily allows combining different constraints, which is implemented the definitions below. The constraint `and` is used two state and verify the conjunction of two properties, whereas the constraint `seq` allows nesting constraints.

```
def and {α1 α2} (h1 : Constraint α1) (h2 : Constraint α2) : Constraint (α1 × α2) where
  prop := fun ⟨a1, a2⟩ ↦ h1.prop a1 ∧ h2.prop a2
  verify' := do
    let ⟨a1, ha1⟩ ← h1.verify
    let ⟨a2, ha2⟩ ← h2.verify
    return ⟨(a1, a2), by simp_all⟩
  elim_exists h0 := by
    simp_all
    obtain ⟨v1, hv1, h1⟩ := h1.elim_exists h0.1
    obtain ⟨v2, hv2, h2⟩ := h2.elim_exists h0.2
    exact ⟨(v1, v2), by simp_all⟩
infixr:70 "∧ᶜ" => and

def seq {α1 α2} (h1 : Constraint α1) (h2 : α1 → Constraint α2) : Constraint (α1 × α2)
  where
  prop := fun ⟨a1, a2⟩ ↦ h1.prop a1 ∧ (h2 a1).prop a2
  verify' := do
    let ⟨a1, ha1⟩ ← h1.verify
    let ⟨a2, ha2⟩ ← (h2 a1).verify
    return ⟨(a1, a2), by simp_all⟩
  elim_exists h0 := by
    ...
```

Lastly, the file contains a variety of constraints for checking that a value returned by a given constraint `h` equals a given value $\texttt{E}_i$. The argument `T` is only used for error messages, to indicate whether the constraint is comparing action or state set IDs. There are multiple variations when the constraint `h` returns two arguments.

```
def eq (h : Constraint ℕ) (T : SetType) (E_i : ℕ) : Constraint Unit where
    prop := fun _ ↦ h.prop E_i
    verify' := do
      let ⟨E'_i, h'⟩ ← h.verify'
      if heq : E_i = E'_i
      then return ⟨(), by simp_all⟩
      else T.toConstraintType.throw_not_eq E_i E'_i
    elim_exists := elim_exists_0
    message := h.message

abbrev eqState (h : Constraint ℕ) (E_i : ℕ) :=
  eq h SetType.States E_i
abbrev eqAction (h : Constraint ℕ) (E_i : ℕ) :=
  eq h SetType.Actions E_i
```

While the actual constraints for the syntactic rules are defined in the file `ValidCertificate`, it only makes sense to conclude this section with the constraint for **UD**. Note that the structure is similar to that of `validUD` and its verification in Section 2.4.4, but this time we only need one definition, and we don't need to worry about eliminating existential quantifiers.

```
def constraintUD (K_i S1_i K1_i K2_i : ℕ) :=
  knowledgeBounds K_i K1_i ∧ᶜ
  knowledgeBounds K_i K2_i ∧ᶜ
  (isStateUnion C S1_i).seq fun (S_i, S'_i) ↦
    (isDeadKnowledge C K1_i).eqState S_i ∧ᶜ
    (isDeadKnowledge C K2_i).eqState S'_i
```

2.4.6 BasicRules

`BasicRules` first defines auxiliary functions that are needed for verifying the basic rules, and it concludes with the constraints for the basic rules. The first functions are used to infer the formalism for a given state set ID and a list of state set IDs. `get_formalism'` returns the option in `StateSetFormalism` (defined on page 43) corresponding to the first `bdd`, `horn` or `mods` state set encountered when traversing the syntax tree for the given state set ID, or `none` if it does not contain any of those. The function `getFormalism` does the same for a list of state set IDs, but it returns `mods` if no formalism has been found. This is a relatively arbitrary choice, and it might be changed if another formalism turns out to be more efficient.

```
namespace Certificate.validSets
def get_formalism' (hC : C.validSets) (S_i : Fin C.states.size) : Option StateSetFormalism :=
  match  heq : C.states[S_i] with
  | .empty => none
  | .init => none
  | .goal => none
  | .bdd _ => bdd
  | .horn _ => horn
  | .mods _ => mods
  | .neg S'_i =>
    have : S'_i < S_i := by
      have := hC.validStates S_i
      simp_all [Certificate.validStateSetExpr]
    hC.get_formalism' ⟨S'_i, by omega⟩
  | .union S'_i S''_i =>
    have : S'_i < S_i ∧ S''_i < S_i := by
      have := hC.validStates S_i
      simp_all [Certificate.validStateSetExpr]
    match hC.get_formalism' ⟨S'_i, by omega⟩ with
    | none => hC.get_formalism' ⟨S''_i, by omega⟩
    | R => R
  ...

def get_formalism (hC : C.validSets) : List (Fin C.states.size) → StateSetFormalism
| [] => mods -- Fallback if all sets are constant
| S_i :: tail =>
  match hC.get_formalism' S_i with
  | none => hC.get_formalism tail
  | some F => F
```

Next up are functions which verify whether the state set expression for the given ID have the formats defined on page 30 and return the corresponding state set variables or state set literals. Below, `get_union_literals` verifies that the state set expression with ID $\texttt{S}_i$ is a union of state set literals. These state set literals are returned as an element of `UnprimedLiterals'` (which has been defined on page 44). Furthermore, the function returns proofs the state set expression with ID $\texttt{S}_i$ is the union of the returned state set literals, and that it has the desired format. The functions `get_variable`, `get_literal`, `get_inter_literals`, `get_inter_variables`, `get_progression_variables`, `get_progression_inter`, `get_regression_variables` and `get_regression_inter` all have similar functionality as suggested by their names.

```
def get_union_literals
  (hC : C.validSets) (R : StateSetFormalism) (S_i : Fin C.states.size) :
  Result (UnprimedLiterals' pt R) fun L ↦
    hC.getStates S_i = L.val.union ∧ IsLiteralUnion pt (R.type pt) (hC.getStates S_i) :=
  withErrorMessage s!"Verifying that the state set #{S_i} is a union of {R} literals" <|
  match heq : C.states[S_i] with
  | .union S'_i S''_i => do
    have ⟨hS'_i, hS''_i⟩ : S'_i < S_i ∧ S''_i < S_i := by
      have := hC.validStates S_i
      simp_all [Certificate.validStateSetExpr]
    let ⟨L1, h1, h2⟩ ← hC.get_union_literals R ⟨S'_i, by omega⟩
    let ⟨L2, h3, h4⟩ ← hC.get_union_literals R ⟨S''_i, by omega⟩
    have h5 : hC.getStates S_i = (L1 ++ L2).val.union := by
      simp only [UnprimedLiterals.val_append, Literals.union_append]
      rw [← h1, ← h3]
      exact hC.getStatesUnion S_i ⟨S'_i, by omega⟩ ⟨S''_i, by omega⟩ (by simp_all)
    have h6 : IsLiteralUnion pt (type pt R) (hC.getStates S_i) := by
      simp_all only [UnprimedLiterals.val_append, Literals.union_append]
      exact IsLiteralUnion.union h2 h4
    return ⟨L1 ++ L2, h5, h6⟩
  | _ => do
    let ⟨l, h1, h2⟩ ← hC.get_literal R ⟨S_i, by omega⟩
    return ⟨UnprimedLiterals.single l, by simp_all; exact IsLiteralUnion.single h2⟩
end Certificate.validSets
```

As in Theorem 22, the file implements three ways to check whether the intersection of variables `X1` is a subset of the union of (unprimed) variables `X2`. Below we show how this is implemented using the operations from Section 2.3.1 in the case where the formalism `R` supports **toCNF**, **CE**, $\wedge$**BC** and $\top$**C**. First we construct an `R`-formula `x1` which is equivalent to the conjunction `X1` and we convert the disjunction `X2` to one CNF-formula. After this, it suffices to check that `x1` entails every clause of the CNF-formula, which can be done using **CE**. To show that this is correct, `check_variables_subset2_correct` first rewrites the statement to a statement about models using the lemma `inter_subset_union_iff_models` from page 42, which requires `X2` to be unprimed. Simplifying the resulting statement using the correctness lemmas of the operations then proves the goal.

```
def Formalism.check_variables_subset2 {R} [F : Formalism pt R]
  [h1 : ClausalEntailment (2 * n) R]
  [h2 : BoundedConjuction (2 * n) R] [Top (2 * n) R]
  [h3 : ToCNF (2 * n) R]
  (X1 : Variables pt R) (X2 : UnprimedVariables pt R) : Bool :=
  let x1 := h2.andList X1
  let φ := h3.disjunctionToCNF X2
  φ.all (fun γ ↦ h1.entails x1 γ)

lemma Formalism.check_variables_subset2_correct {R} [F : Formalism pt R]
  [h1 : ClausalEntailment (2 * n) R]
  [h2 : BoundedConjuction (2 * n) R] [Top (2 * n) R]
  [h3 : ToCNF (2 * n) R]
  (X1 : Variables pt R) (X2 : UnprimedVariables pt R) :
  check_variables_subset2 X1 X2 ↔ X1.inter ⊆ X2.val.union :=
  by
    rw [UnprimedVariables.inter_subset_union_iff_models]
    simp [check_variables_subset2, Variable.models,
      h1.entails_correct, h2.andList_correct, h3.disjunctionToCNF_correct]
```

Similarly, the other two methods of checking the inclusion `X1.inter ⊆ X2.val.union` are implemented. The following definition and lemma decide based on the formalism which method should be used.

```
def StateSetFormalism.check_variables_subset (R : StateSetFormalism)
  (X1 : Variables' pt R) (X2 : UnprimedVariables' pt R) : Bool :=
  match R with
  | .bdd => check_variables_subset1 X1 X2
  | .horn => check_variables_subset2 X1 X2
  | .mods => check_variables_subset2 X1 X2

lemma StateSetFormalism.check_variables_subset_correct (R : StateSetFormalism)
  (X1 : Variables' pt R) (X2 : UnprimedVariables' pt R) :
  check_variables_subset R X1 X2 ↔ X1.inter ⊆ X2.val.union :=
  match R with
  | .bdd => check_variables_subset1_correct X1 X2
  | .horn => check_variables_subset2_correct X1 X2
  | .mods => check_variables_subset2_correct X1 X2
```

Notice that the logical statements

$$\bigwedge_{\varphi_i \in L_1^+} \varphi_i \wedge \bigwedge_{\varphi_i \in L_1^-} \neg\varphi_i \models \bigvee_{\varphi_i \in L_2^+} \varphi_i \vee \bigvee_{\varphi_i \in L_2^-} \neg\varphi_i \quad \text{and} \quad \bigwedge_{\varphi_i \in L_1^+} \varphi_i \wedge \bigwedge_{\varphi_i \in L_2^-} \varphi_i \models \bigvee_{\varphi_i \in L_2^+} \varphi_i \vee \bigvee_{\varphi_i \in L_1^-} \varphi_i$$

are equivalent. We use this to translate the statement **B1**, which uses literals, to a statement about variables, which can then by verified using `check_variables_subset` from above. The `simp` tactic in `checkB1_correct` implicitly uses some lemmas from Section 2.3.3, including `union_val` and `inter_val`.

```
def checkB1 R (L1 L2 : UnprimedLiterals' pt R) : Bool :=
  R.check_variables_subset (L1.1 ++ L2.2).val (L2.1 ++ L1.2)

lemma checkB1_correct R {L1 L2 : UnprimedLiterals' pt R} :
  checkB1 R L1 L2 ↔ L1.val.inter ⊆ L2.val.union :=
  by
    simp [checkB1, check_variables_subset_correct, Set.inter_compl_subset_union_compl]
```

We are finally able to define the constraint for **B1**. The property of the constraint expresses that the state set expressions with IDs $\texttt{S1}_i$ and $\texttt{S2}_i$ have the correct format, and it states that the state set corresponding to $\texttt{S1}_i$ is a subset of the one corresponding to $\texttt{S2}_i$. For the verification we first check that $\texttt{S1}_i$ and $\texttt{S2}_i$ are valid state set IDs. Next we obtain the formalism and the literals of both sides using the definitions discussed at the beginning of the section. Lastly, we verify semantically that the inclusion holds using `checkB1`.

```
def constraintB1 (hC : C.validSets) (S1ᵢ S2ᵢ : ℕ) : Constraint Unit where
  prop := fun _ ↦ ∃ hS1ᵢ hS2ᵢ,
    have R := hC.get_formalism [⟨S1ᵢ, hS1ᵢ⟩, ⟨S2ᵢ, hS2ᵢ⟩]
    IsLiteralInter pt (R.type pt) (hC.getStates ⟨S1ᵢ, hS1ᵢ⟩) ∧
    IsLiteralUnion pt (R.type pt) (hC.getStates ⟨S2ᵢ, hS2ᵢ⟩) ∧
    hC.getStates ⟨S1ᵢ, hS1ᵢ⟩ ⊆ hC.getStates ⟨S2ᵢ, hS2ᵢ⟩
  verify' :=
    do
      let ⟨⟨⟩, hS1ᵢ⟩ ← (stateBounds' C S1ᵢ).verify
      let ⟨⟨⟩, hS2ᵢ⟩ ← (stateBounds' C S2ᵢ).verify
      let R := hC.get_formalism [⟨S1ᵢ, hS1ᵢ⟩, ⟨S2ᵢ, hS2ᵢ⟩]
      let ⟨L1, h1, h2⟩ ← hC.get_inter_literals R ⟨S1ᵢ, hS1ᵢ⟩
      let ⟨L2, h3, h4⟩ ← hC.get_union_literals R ⟨S2ᵢ, hS2ᵢ⟩
      if h5 : R.checkB1 L1 L2 then
        have h6 : hC.getStates ⟨S1ᵢ, hS1ᵢ⟩ ⊆ hC.getStates ⟨S2ᵢ, hS2ᵢ⟩ := by
          simp_all only [checkB1_correct]
        return ⟨(), by use hS1ᵢ, hS2ᵢ, h2, h4, h6⟩
      else
        throwUnvalid s!"The state set #{S1ᵢ} is not a subset of #{S2ᵢ}"
  elim_exists := elim_exists_0
```

Note that functions `get_inter_literals`, `get_inter_variables` and `checkB1` depend on the operations defined in Section 2.3.1. Since these operations have not yet been implemented, Lean displays the message `INTERNAL PANIC: executed 'sorry'` when trying to verify **B1**. To make this message more clear, the last part of `verify'` has been replaced by a custom error message. The same has been done for the constraints of the rules **B2**-**B4**.

For the rules **B2** and **B3**, we need unprimed variables corresponding to the preconditions, the adding effects and the deleting effects of the action with a given ID $\texttt{a}_i$, as in shown for preconditions below. Additionally, there are versions working with `UnprimedVariables'` which group the adding and deleting effects, and there is a method to get the original variables of the planning task that appear in the adding and deleting effects.

```
def preVariable R (a_i : Fin pt.actions'.length) : UnprimedVariable' pt R :=
  UnprimedVariable.ofVarset' (R.type pt) pt.actions'[a_i].pre'

def preVariables R (a_i : Fin pt.actions'.length) : UnprimedVariables' pt R :=
  [preVariable R a_i]
def effectVariables R (a_i : Fin pt.actions'.length) : UnprimedVariables' pt R :=
  [addVariable R a_i, delVariable R a_i]

def effectVarSet' (a_i : Fin pt.actions'.length) : VarSet' n :=
  VarSet'.union pt.actions'[a_i].add' pt.actions'[a_i].del'
```

To verify **B2**, it is sufficient to verify the statement for each action a individually, as is done in `checkB2`. We use the same trick as for **B1** to transform the statement about state set literals into a statement about state set variables, which is then verified using `checkB2'`. For `checkB2'`, we start by making the unprimed variables corresponding to the original variables in $\operatorname{add} a \cup \operatorname{del} a$ primed in `X0` and `preVariables R a_i`. From `mem_inter_toPrimed` on page 43 it follows that a state s is in `X0'.inter` if and only if there exists a state s' in `X.val.inter` with $\operatorname{pre} a \subseteq s'$ which matches s on all variables not in $\operatorname{add} a \cup \operatorname{del} a$. If we additionally require that the effects of a hold in s, then this is equivalent with s being in the progression of `X0.val.inter` with the action a. This is done in the definition of `X1'`, where we also include the variables `X1`. Finally, we verify the inclusion `X1'.inter ⊆ X2.val.union` using the same method as for **B1**.

```
def checkB2' R (a_i : Fin pt.actions'.length) (X0 X1 X2 : UnprimedVariables' pt R) : Bool :=
  let X0' := UnprimedVariables.toPrimed (preVariables R a_i ++ X0) (effectVarSet' a_i)
  let X1' := X0' ++ (effectVariables R a_i ++ X1).val
  R.check_variables_subset X1' X2

lemma checkB2'_correct {R a_i} {X0 X1 X2 : UnprimedVariables' pt R} :
  checkB2' R a_i X0 X1 X2 ↔
    pt.progression' X0.val.inter pt.actions'[a_i] ∩ X1.val.inter ⊆ X2.val.union :=
  by ...

def checkB2 R (X : UnprimedVariables' pt R) (A : ActionIds pt)
  (L1 L2 : UnprimedLiterals' pt R) : Bool :=
  A.all (fun a_i ↦ checkB2' R a_i X (L1.1 ++ L2.2) (L2.1 ++ L1.2))

lemma checkB2_correct {R X A} {L1 L2 : UnprimedLiterals' pt R} :
 checkB2 R X A L1 L2 ↔ pt.progression X.val.inter A.toActions ∩ L1.val.inter ⊆
    L2.val.union :=
  by ...
```

The constraint for **B2** is very similar to `constraintB1`, and is therefore omitted. We will only briefly discuss the rules **B3**-**B5**. The verification and constraint of **B3** are very similar to those of **B2**. For **B4**, the constraint is similar to `constraintB1`, with the main difference being that there are two formalism instead of one. The method `checkB4` has not yet been implemented because of time limitations. The verification for **B5** uses `getActions'` (from page 50) to check that all action IDs of the left action set are in the right action set.

Recall that Theorem 23 only listed two options to verify the statements **B2** and **B3**. The reason for this is that the corresponding proof in [12, Theorem 5.6] claimed that $\wedge$**BC** is needed either way to construct the progression (or regression). However, our formalization shows that it is possible to verify **B2** (and similarly **B3**) without constructing the intersection of the variables in `X0` or `X0'`. This means that it is possible to verify **B2** and **B3** without using $\wedge$**BC** by choosing `check_variables_subset3` (which corresponds to the third option in Theorem 22) to verify `X1'.inter` $\subseteq$ `X2.val.union`. Therefore, in addition of the two options in Theorem 23, **B2** and **B3** can also be verified using the operations **toDNF**, **IM**, $\vee$**BC**, **CL**, $\bot$**C** and **RN**$_{\prec}$.

2.4.7 ValidCertificate

The file `ValidCertificate` starts with the constraints for the syntactic rules. We already saw the constraint for **UD** in Section 2.4.5, below we show the constraint for **PI**, one of the more complicated rules. The constraint first requires that all premises have IDs smaller than $\texttt{K}_i$, after which it checks that the state set in the conclusion is the negation of a state set $\texttt{S}_i$. To check that the first premise is of the form $S[A^{\Pi}] \subseteq S \cup S'$, we first check that it has the form $S_2 \subseteq S_3$, then we check that S_2 is of the form $S[A]$, that A is of the form A^{Π} and that S_3 is of the form $S \cup S'$. The other premises are checked similarly.

```
def constraintPI (K_i S1_i K1_i K2_i K3_i : ℕ) :=
  knowledgeBounds K_i K1_i ∧ᶜ
  knowledgeBounds K_i K2_i ∧ᶜ
  knowledgeBounds K_i K3_i ∧ᶜ
  (isStateNeg C S1_i).seq fun S_i ↦
    ((isStateSubsetKnowledge C K1_i).seq fun (S2_i, S3_i) ↦
      (((isStateProgr C S2_i).leftEqState S_i).seq fun A_i ↦
        isActionAll C A_i) ∧ᶜ
      ((isStateUnion C S3_i).leftEqState S_i).seq fun S'_i ↦
        (isDeadKnowledge C K2_i).eqState S'_i) ∧ᶜ
    ((isStateSubsetKnowledge C K3_i).rightEqState S_i).seq fun SI_i ↦
      isStateInit C SI_i
```

The constraints for all basic rules and derivation rules are grouped in the definition below. The return type of `constraintKnowledge` includes a sigma type, as the parameter α depends on the rule. Since most rules use a combination of multiple constraints, α can be relatively complicated. Therefore, we allow Lean to automatically infer α using the placeholder "_".

```
def constraintKnowledge {C : Certificate pt} (hC : C.validSets)
  (K_i : Fin C.knowledge.size) : Σ α, Constraint α :=
  match C.knowledge[K_i] with
  | dead S_i K =>
    match K with
    | ED S_i => ⟨_, isStateEmpty C S_i⟩
    | UD S_i K1_i K2_i => ⟨_, constraintUD C K_i S_i K1_i K2_i⟩
    | SD S_i K1_i K2_i => ⟨_, constraintSD C K_i S_i K1_i K2_i⟩
    ...
  ...
```

Using `constraintKnowledge` we can finally express what it means for knowledge to be valid. We define validity of certificates by extending the definition of `validSets`. Here, `tovalidSets` gives the underlying structure `C.validSets`, which is needed for verifying the basic rules.

```
def validKnowledge {C : Certificate pt} (hC : C.validSets) (K_i : Fin C.knowledge.size) :
  Prop :=
  (constraintKnowledge hC K_i).snd.valid

structure valid (C : Certificate pt) extends C.validSets where
  validKnowledge : ∀ K_i, C.validKnowledge tovalidSets K_i
```

The file contains various lemmas for obtaining bounds proofs from a valid certificate, like following lemma for `StateSetExpr.union`:

```
lemma stateUnionBounds {C : Certificate pt} (hC : C.valid) (S_i : Fin C.states.size)
  {S'_i S''_i} (h : C.states[S_i]? = some (StateSetExpr.union S'_i S''_i)) :
  S'_i < C.states.size ∧ S''_i < C.states.size :=
  by
    have h' := hC.validStates S_i
    rcases S_i with ⟨S_i, hS_i⟩
    simp_all [Certificate.validStateSetExpr]
    omega
```

Note that so far we did not require the certificate to actually claim that the planning task is unsolvable, we only required that all statements are valid. `IsUnsolvable` below formalizes this requirement, and `verifyIsUnsolvable` verifies it by checking for all knowledge statements in the certificate whether they claim unsolvability. The optional parameter of `verifyIsUnsolvable` is by default initialized to `Fin.last C.knowledge.size` which has value `C.knowledge.size`. It decrements with each recursive function call until either a statement with constructor `unsolvable` has been reached, or all knowledge statements have been checked.

```
abbrev IsUnsolvable : Prop :=
  ∃ K_i : Fin C.knowledge.size, ∃ K, C.knowledge[K_i] = unsolvable K

def verifyIsUnsolvable : optParam (Fin (C.knowledge.size + 1)) (Fin.last C.knowledge.size)
    →
  Result' (IsUnsolvable C)
| 0 => throwUnvalid "Unsolvability NOT proven"
| ⟨K_i + 1, h⟩ =>
  match heq : C.knowledge[K_i] with
  | unsolvable K => return ⟨(), by use ⟨K_i, by omega⟩, K, heq⟩
  | _ => verifyIsUnsolvable ⟨K_i, by omega⟩
```

To validate the certificate, first all action set expressions are validated, then all state set expressions, followed by all knowledge expressions, and finally it is checked that the certificate claims unsolvability. For each of the types of expressions the methods `verifyAll` and `verifyAll'` check the expressions in order of increasing ID. Helve uses a different approach, and verifies the expressions in the order in which they appear in the file. This implies that action set expressions are potentially checked after

state set expressions and knowledge expressions and so on. Intuitively this approach makes sense, as one can incrementally build the certificate and verify each expression before adding it. However, for a formally verified validator this would require proving that the certificate remains valid when adding a new valid expression to the certificate. We have tried this, and it seems doable, but we decided against it as the current approach is simpler.

```
def verify : Result' (C.valid ∧ IsUnsolvable C) :=
  do
    let ⟨(), h1⟩ ← verifyAll' C.verifyActionSetExpr
    let ⟨(), h2⟩ ← verifyAll' C.verifyStateSetExpr
    let hC : C.validSets := ⟨h1, h2⟩
    let ⟨(), h3⟩ ← verifyAll (constraintKnowledge hC)
    let ⟨(), h4⟩ ← verifyIsUnsolvable C
    return ⟨(), ⟨hC, h3⟩, h4⟩
```

2.4.8 ToDerivation

We still need to show that `Certificate.valid` together with `Certificate.IsUnsolvable` is sufficient to conclude that the corresponding planning task is unsolvable. We show this by making a translation from a valid certificate to a `Derivation`. This requires specifying the semantic meaning of the different types of knowledge, which is done by the following definition.

```
namespace Validator.Certificate.valid
def conclusion {C : Certificate pt} {hC : C.valid} (Kᵢ : Fin C.knowledge.size) : Prop :=
  match heq : C.knowledge[Kᵢ] with
  | dead Sᵢ K =>
    have hSᵢ : Sᵢ < C.states.size :=
      hC.deadKnowledgeBound Kᵢ (by use K; simp_all)
    let S := hC.getStates ⟨Sᵢ, hSᵢ⟩
    Dead pt S
  | actionSubset Aᵢ A'ᵢ K =>
    have hᵢ : Aᵢ < C.actions.size ∧ A'ᵢ < C.actions.size :=
      hC.actionSubsetKnowledgeBounds Kᵢ (by use K; simp_all)
    let A := hC.getActions ⟨Aᵢ, hᵢ.1⟩
    let A' := hC.getActions ⟨A'ᵢ, hᵢ.2⟩
    A ⊆ A'
  | stateSubset Sᵢ S'ᵢ K =>
    have hᵢ : Sᵢ < C.states.size ∧ S'ᵢ < C.states.size :=
      hC.stateSubsetKnowledgeBounds Kᵢ (by use K; simp_all)
    let S := hC.getStates ⟨Sᵢ, hᵢ.1⟩
    let S' := hC.getStates ⟨S'ᵢ, hᵢ.2⟩
    S ⊆ S'
  | unsolvable _ => Unsolvable pt
```

Next, the file includes lemmas which allow simplifying this definition when the constructor for the knowledge is known. For dead knowledge this is the following lemma:

```
lemma conclusionDead
  {C : Certificate pt} (hC : C.valid)
  (K_i : Fin C.knowledge.size) {S_i : Fin C.states.size}
  (h : ∃ K, C.knowledge[K_i]? = dead S_i K) :
  hC.conclusion K_i ↔ Dead pt (hC.getStates S_i) :=
  by
    simp [conclusion]
    split
    all_goals simp_all
```

In what follows, we describe the translation from a valid certificate the type `Derivation`. Spanning 487 lines and covering 32 cases for all different rules, this is by far the longest definition of the project. Below the following explanation we show the translation for the rule **UD**. After unfolding some definitions and making a case distinction on the knowledge `C.knowledge[K_i]`, most rules do the following:

1. If needed, rename variables to match those of the constraints.
2. Simplify the hypothesis `h`, containing the proof that the knowledge with ID `K_i` is valid, to a statement like `(C.constraintUD K_i S1_i K1_i K2_i).valid`.
3. Eliminate the existential quantifiers at `h` using `apply Constraint.elim_exists at h`.
4. Once the quantifiers are eliminated, we can use `rcases` to get the all IDs returned by the constraint (for the example below these are `S_i` and `S'_i`), and the property about this data `h'`. Note that this is not possible if the data would be existentially quantified, because existential quantifiers can only be decomposed if the goal is a proposition, or using the axiom of choice. The latter would make the definition non-computable, and it would be a lot more convoluted.
5. Simplify `h'`, and split it into different parts. In the example below `hK1_i`, `hK2_i`, `hS1_i` are bounds for IDs, `hS1` is a proof of `C.states[S1_i]? = some (StateSetExpr.union S_i S'_i)`, `hK1` is a proof for `∃ K, C.knowledge[K1_i]? = some (dead S_i K)` and `hK2` is a proof for `∃ K, C.knowledge[K2_i]? = some (dead S'_i K)`.
6. `h'` only contains bounds for the IDs of expressions decomposed during verification. If needed, bounds for other IDs can be obtained using lemmas from `ValidCertificate` (Section 2.4.7).
7. If the conclusion of the rule contains a composite set expression, then we need to decompose it using lemmas from the file `SetExpr` (Section 2.4.3). In the example below, the lemma `getStatesUnion` is used to rewrite the goal `Derivation pt (Dead pt (….getStates ⟨S1_i, …⟩))` into `Derivation pt (Dead pt (….getStates ⟨S_i, hS_i⟩ ∪ ….getStates ⟨S'_i, hS'_i⟩))`.
8. We apply the rule from `Derivation` corresponding to our current rule, after which we need to give derivations for all premises. The rule `Derivation.UR` is used for both `StateSubsetKnowledge.URS` and `ActionSubsetKnowledge.URA`. The same applies for the other rules from basic set theory.
9. If the premises contain composite set expressions, we do the same as in step 7, but in the other direction. This is not the case for **UD**.

10. We use the lemmas from above to get `conclusion` into the goal statement, and then we use recursion to get a derivation with this conclusion. For the first premise in the example below we rewrite the goal to `Derivation pt (conclusion ⟨K1ᵢ, ⋯⟩)`, which we can solve by recursion using `K1ᵢ < Kᵢ`.

The process for the basic rules is a bit shorter because there are no premises and `h'` contains all proofs needed for constructing the corresponding derivation.

```
def toDerivation {C : Certificate pt} (hC : C.valid) (Kᵢ : Fin C.knowledge.size) :
  Derivation pt (hC.conclusion Kᵢ) :=
  by
    unfold conclusion
    have h := hC.validKnowledge Kᵢ
    unfold Certificate.validKnowledge constraintKnowledge at h
    rcases Kᵢ with ⟨Kᵢ, hKᵢ⟩
    split -- cases on C.knowledge[Kᵢ]
    -- case dead Sᵢ K
    case h_1 Sᵢ K heq =>
      rw [heq] at h
      simp_all
      cases K with
      | ED => ...
      | UD K1ᵢ K2ᵢ =>
        rename' Sᵢ => S1ᵢ
        simp only at h
        apply Constraint.elim_exists at h
        rcases h with ⟨⟨⟨⟩, ⟨⟩, ⟨Sᵢ, S'ᵢ⟩, ⟨⟩, ⟨⟩⟩, ⟨h', _⟩⟩
        simp [constraintUD] at h'
        rcases h' with ⟨hK1ᵢ, hK2ᵢ, ⟨hS1ᵢ, hS1⟩, hK1, hK2⟩
        have ⟨hSᵢ, hS'ᵢ⟩: Sᵢ < C.states.size ∧ S'ᵢ < C.states.size :=
          hC.stateUnionBounds ⟨S1ᵢ, hS1ᵢ⟩ hS1
        rw [hC.getStatesUnion ⟨S1ᵢ, hS1ᵢ⟩ ⟨Sᵢ, hSᵢ⟩ ⟨S'ᵢ, hS'ᵢ⟩ hS1]
        apply Derivation.UD
        · rw [← hC.conclusionDead ⟨K1ᵢ, by omega⟩ hK1]
          exact hC.toDerivation ⟨K1ᵢ, by omega⟩
        · rw [← hC.conclusionDead ⟨K2ᵢ, by omega⟩ hK2]
          exact hC.toDerivation ⟨K2ᵢ, by omega⟩
      ...
    ...
```

For the correctness of the validator, we apply the soundness theorem for `Derivation` from Section 2.2.3 to the `Derivation` corresponding to a given valid certificate. If valid certificate states that the given planning task is unsolvable, we can use this to conclude that the planning task is indeed unsolvable.

```
theorem soundness' {C : Certificate pt} (hC : C.valid) (Kᵢ : Fin C.knowledge.size) :
  hC.conclusion Kᵢ :=
  Derivation.soundness (hC.toDerivation Kᵢ)
```

```
theorem soundness {C : Certificate pt} (hC : C.valid) (h : C.IsUnsolvable) : Unsolvable pt
    :=
  by
    rcases h with ⟨K_i, hK⟩
    rw [← hC.conclusionUnsolvable K_i hK]
    exact hC.soundness' K_i
end Validator.Certificate.valid
```

2.4.9 Validator

The different components of the validator are combined in the file `Validator`. For now, we use a planning task and certificate in the folder `test`, which has been copied from Helve [14]. These files are parsed using the parsers implemented in `PlanningTask.Parser` (Section 2.1.3) and `Certificate.Parser` (Section 2.4.2). Finally, the certificate is verified using the function `Certificate.verify` implemented in `Certificate.ValidCertificate` (Section 2.4.7), and if it is valid the soundness lemma from `Certificate.ToDerivation` (Section 2.4.8) allows us to conclude that the given planning task is unsolvable. This definition is obviously not final, and in the future it should allow parsing planning tasks and certificates in user-specified files.

```
def main : IO Unit :=
  do
    try
      let path ← IO.currentDir
      let pt_path := path / "test" / "success-task.txt"
      let ⟨n, pt⟩ ← STRIPS.parse pt_path
      IO.println (repr pt)
      IO.println s!"initial state : {(List.finRange n).filter (pt.init'[·])}"
      let certificate_path := path / "test" / "success-certificate.txt"
      IO.println "Parsing the certificate"
      let C ← Certificate.parse pt certificate_path
      IO.println "Verifying the certificate"
      match C.verify with
      | .ok ⟨(), hC, h⟩ =>
        have : Unsolvable pt := hC.soundness h
        IO.println "The certificate is valid!"
      | .error e =>
        throw (IO.userError (e.show ""))
    catch e =>
      IO.println e
```

Chapter 3

Discussion

Given that none of the formalisms has been implemented yet, it is not possible to actually run the validator on certificates (or at least not successfully). Therefore, it is not yet possible to evaluate the performance of the validator and compare it to the performance of Helve. However, there are other interesting topics to discuss. In Section 3.1 we review a critical bug that we found in Helve, and in Section 3.2 we update Theorem 23 with the findings from Section 2.4.6. In Section 3.3 we consider some possible directions for extending the validator.

3.1 Bug in Helve

While working on the thesis we discovered a bug in Helve [14], the C++ validator for the proof system[1]. The bug occurred in the file `helve/rules/basic_statements/basic_statement_5.cc`, for the verification of the basic rule **B5**, which is shown below. The pointer `*left_set` is initialized using the identifier `right_id` instead of `left_id`.

```
std::unique_ptr<Knowledge> basic_statement_5(SetID left_id, SetID right_id,
                                             std::vector<KnowledgeID> &,
                                             const ProofChecker &proof_checker) {

    std::unordered_set<size_t> left_indices, right_indices;
    const ActionSet *left_set = proof_checker.get_set<ActionSet>(right_id);
    const ActionSet *right_set = proof_checker.get_set<ActionSet>(right_id);
    left_set->get_actions(proof_checker, left_indices);
    right_set->get_actions(proof_checker, right_indices);

    for (int index: left_indices) {
        if (right_indices.find(index) == right_indices.end()) {
            throw std::runtime_error("Statement B5 is false.");
        }
    }
    return std::unique_ptr<Knowledge>(new SubsetKnowledge<ActionSet>(left_id,right_id));
}
```

[1]The GitHub issue for this bug can be found at `https://github.com/salome-eriksson/helve/issues/2`.

$$\dfrac{\dfrac{\dfrac{}{\left\{I^\Pi\right\}[\emptyset] \subseteq \left\{I^\Pi\right\} \cup \emptyset}\,\mathbf{B2} \quad \dfrac{}{A^\Pi \subseteq \emptyset}\,\mathbf{B5}\ (\text{bug})}{\left\{I^\Pi\right\}[A^\Pi] \subseteq \left\{I^\Pi\right\} \cup \emptyset}\,\mathbf{AT} \quad \dfrac{}{\emptyset\ \text{dead}}\,\mathbf{ED} \quad \dfrac{\dfrac{}{\emptyset\ \text{dead}}\,\mathbf{ED} \quad \dfrac{}{\left\{I^\Pi\right\} \cap S^\Pi_G \subseteq \emptyset}\,\mathbf{B1}}{\left\{I^\Pi\right\} \cap S^\Pi_G\ \text{dead}}\,\mathbf{SD}}{\dfrac{\left\{I^\Pi\right\}\ \text{dead}}{\Pi\ \text{unsolvable}}\,\mathbf{CI}}\,\mathbf{PG}$$

Figure 3.1: A derivation showing that if $\left\{I^\Pi\right\} \cap S^\Pi_G \subseteq \emptyset$, then the bug can be exploited to derive that Π is unsolvable. The main idea is to use **AT** with the bug to derive $\left\{I^\Pi\right\}[A^\Pi] \subseteq \left\{I^\Pi\right\} \cup \emptyset$ from $\left\{I^\Pi\right\}[\emptyset] \subseteq \left\{I^\Pi\right\} \cup \emptyset$, where the latter trivially holds by the definition of progression (Definition 10).

Because of this bug, the validator will always return true when verifying **B5**, as it checks whether $A' \subseteq A'$ instead of checking whether $A \subseteq A'$. To illustrate that this is a critical bug, Figure 3.1 shows how this bug could in theory be used to show the unsolvability of any planning task in which the initial state is not a goal state. Helve does not accept the exact certificate corresponding to this derivation, since it does not allow both sides of **B1** and **B2** to be constant, but the derivation can be modified to work for specific planning tasks.

While this bug would probably be discovered relatively quickly in case of unexpected results, it does illustrate that validators themselves are not immune to bugs.

3.2 Theorem 23 revisited

As discussed at the end of Section 2.4.6, the operation $\wedge$**BC** is not needed to verify the rules **B2**-**B3**. Because of this, there is an additional option for verifying these rules, which was not mentioned in Theorem 23.

Theorem 27 (Theorem 23 revised)**.** *The statements* **B2** *and* **B3** *can be validated in time polynomial in the total size of the involved formulas if* **R** *efficiently supports one of the following:*

- ***SE***, $\wedge$***BC***, $\vee$***BC***, ***CL***, $\bot$***C*** *and* $\boldsymbol{RN}_\prec$
- ***toCNF***, ***CE***, $\wedge$***BC***, ***CL***, *and* $\boldsymbol{RN}_\prec$
- ***toDNF***, ***IM***, $\vee$***BC***, ***CL***, $\bot$***C*** *and* $\boldsymbol{RN}_\prec$

As the discussion about `inter_variables_append` on page 42 illustrates, working with primed variables can be quite subtle. In her PhD thesis, Eriksson interpreted these variables using Lemma 4.1 of [12], stating the following.

Lemma. *Let φ, ψ and χ be propositional formulas, and X and X' be propositional variables, where φ, ψ and χ do not mention any variable from X'. The statement $(\exists X.\varphi) \wedge \psi \models \chi$ holds iff $\varphi[X \to X'] \wedge \psi \models \chi$ does.*

The format of the entailment in the lemma is the same as the one needed to verify **B2** and **B3**, hence there is no need to further manipulate the conjunctions with renamed variables after applying this lemma. In Lean, it often makes more sense to break up lemmas in smaller parts. In this case it meant trying to interpret the state set corresponding to $\exists X.\varphi$ first, and then reasoning about the intersections and inclusions later on using separate lemmas. After a few attempts, this was characterized by the lemma `mem_inter_toPrimed` from Section 2.3.3, shown below.

```
lemma UnprimedVariables.mem_inter_toPrimed [F : Formalism pt R] [Renaming (2 * n) R]
  {X : UnprimedVariables pt R} {V s} : s ∈ (toPrimed X V).inter ↔
    ∃ s' ∈ X.val.inter, ∀ i ∉ V.val, i ∈ s' ↔ i ∈ s := ...
```

Using the lemma `inter_variables_append` mentioned earlier, it was possible to reduce the verification of **B2** and **B3** to `check_variables_subset` (from page 57) without needing ∧**BC**. This third option of Theorem 27 was only discovered because of the formalization.

3.3 Future work

The obvious next step for the project would be to implement the different formalisms (BDDs, Horn formulas, 2CNF formulas and MODS) and their operations. For BDD we are planning to use the BDD-library implemented by Yaron [38, 37]. Since this library still contains some `sorry`-placeholders, we will probably first implement the other formalisms. For MODS, we will add the missing instances for the type discussed in Section 2.3.2. For Horn formulas and 2CNF formulas we might use subtypes of the `CNF`-type from Section 2.3.1, or implement new types. After this there are several possible ideas for extending the validator.

Currently, a STRIPS description is used to read the problem description. Since this is an unconventional format, it needs to be generated by the solvers themselves, or by Fast-Downward [24] if it is used to ground the problem. As discussed in Section 1.3.6, this is not ideal for a fully formalized validator, and it would be better to have a formally verified grounder. Alternatively, one could try to extend the proof system to work with more features of the PDDL language, and directly parse the problem description in PDDL. This might also make producing the certificates easier for algorithms which do not fully translate the input problem to STRIPS.

Another option would be to expand the validator to also work with certificates of optimality. Out of the three systems discussed in Section 1.3.4, the second system [28], which is also implemented in Helve, would make the most sense, since it is already based on the proof system. Note that the optimality certificates from [10] seem to have better efficiency guarantees, making this system potentially more interesting. While CakePB [8] is already a formally verified pseudo-Boolean proof checker, it is implemented in CakeML, and not in Lean. Adding support for this system to the validator would likely require implementing a formalized pseudo-Boolean proof checker in Lean. It might be more interesting to implement such a validator in CakeML.

Chapter 4

Conclusion

The goal of this thesis was to implement a formalized validator for certificates of unsolvability using the proof system developed by Eriksson, Röger and Helmert [18, 12] in Lean 4. We discussed the relevant background and related work in Chapter 1 and our formalization in Chapter 2. The formalization highlights two aspects of formal verification.

First, there is the theoretical aspect, which in our case mainly consists of formalized proof for soundness and completeness of the proof system. Additionally, the implementation showed that the operation $\wedge\mathbf{BC}$ is not needed to verify the rules **B2** and **B3** as discussed in Section 3.2, resulting in a new way of verifying these rules. We want to emphasize that the formalization played a key role in noticing this, illustrating that formalizations can lead to new theoretical insights.

The second aspect is the computational component, consisting of the validator itself. As explained in Section 3.1, the discovery of a critical bug in the C++ validator for the proof system highlights the benefit of a formalized validator.

Because of time constraints, the validator itself has not been finished yet. Most importantly, the formalisms for representing sets of states are not yet implemented. In Section 3.3 we discussed some possible extensions of the validator that could be implemented after finishing the formalisms. For a fully formalized validator, the grounding of the planning problem should also be formalized. This could either be done by integrating or implementing a formalized grounder, or by extending the proof system to work with a more general subset of PDDL. Another option would be to extend the validator to also verify certificates of optimality.

Bibliography

[1] Mohammad Abdulaziz and Peter Lammich. 'A Formally Verified Validator for Classical Planning Problems and Solutions'. In: *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*. 2018, pages 474–479. https://doi.org/10.1109/ICTAI.2018.00079 (cited on pages 7, 17).

[2] Vidal Alcázar and Álvaro Torralba. 'A Reminder about the Importance of Computing and Exploiting Invariants in Planning'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 25.1 (Apr. 2015), pages 2–6. https://doi.org/10.1609/icaps.v25i1.13708 (cited on page 16).

[3] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn and Christine Rizkallah. 'A Framework for the Verification of Certifying Computations'. In: *Journal of Automated Reasoning* 52 (2014), pages 241–273. https://doi.org/10.1007/s10817-013-9289-2 (cited on page 3).

[4] Anne Baanen, Matthew Robert Ballard, Johan Commelin, Bryan Gin-ge Chen, Michael Rothgang and Damiano Testa. *Growing Mathlib: maintenance of a large scale mathematical library*. 2025. https://doi.org/10.48550/arXiv.2508.21593 (cited on page 3).

[5] Bart Bogaerts, Stephan Gocht, Ciaran McCreesh and Jakob Nordström. 'Certified Dominance and Symmetry Breaking for Combinatorial Optimisation'. In: *Journal of Artificial Intelligence Research* 77 (Aug. 2023). ISSN: 1076-9757. https://doi.org/10.1613/jair.1.14296 (cited on page 18).

[6] Randal Bryant. 'Graph-Based Algorithms for Boolean Function Manipulation'. In: *IEEE Transactions on Computers* C-35.8 (Sept. 1986), pages 677–691. https://doi.org/10.1109/TC.1986.1676819 (cited on page 8).

[7] Tom Bylander. 'The computational complexity of propositional STRIPS planning'. In: *Artificial Intelligence* 69.1 (1994), pages 165–204. ISSN: 0004-3702. https://doi.org/10.1016/0004-3702(94)90081-7 (cited on page 12).

[8] *CakePB*. https://gitlab.com/MIAOresearch/software/cakepb (visited on 01/08/2025) (cited on pages 18, 68).

[9] Wikipedia contributors. *Rush Hour (puzzle) — Wikipedia, The Free Encyclopedia*. https://rushhourgame.thinkfun.com/ (visited on 29/11/2025) (cited on page 4).

[10] Simon Dold, Malte Helmert, Jakob Nordström, Gabriele Röger and Tanja Schindler. 'Pseudo-Boolean Proof Logging for Optimal Classical Planning'. In: *CoRR* abs/2504.18443 (2025). https://doi.org/10.48550/ARXIV.2504.18443 (cited on pages 6, 16, 17, 68).

[11] François G. Dorais, Kyrill Serdyuk and Emma Shroyer. *lean4-parser*. https://github.com/fgdorais/lean4-parser (visited on 25/11/2025) (cited on page 26).

[12] Salomé Eriksson. 'Certifying planning systems : witnesses for unsolvability'. PhD thesis. University of Basel, 2019. https://doi.org/10.5451/unibas-007176138 (cited on pages a, 3, 4, 7–16, 20, 60, 67, 69).

[13] Salomé Eriksson. *Code from the PhD thesis "Certifying Planning Systems: Witnesses for Unsolvability"*. July 2019. https://doi.org/10.5281/zenodo.3355459 (cited on page 17).

[14] Salomé Eriksson. *Helve*. https://github.com/salome-eriksson/helve (visited on 25/06/2025) (cited on pages a, 17, 20, 65, 66).

[15] Salomé Eriksson and Malte Helmert. 'Certified Unsolvability for SAT Planning with Property Directed Reachability'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 30.1 (June 2020), pages 90–100. https://doi.org/10.1609/icaps.v30i1.6649 (cited on page 16).

[16] Salomé Eriksson and Malte Helmert. *Code from Eriksson-Helmert, ICAPS 2020*. Mar. 2020. https://doi.org/10.5281/zenodo.3691796 (cited on page 17).

[17] Salomé Eriksson and Malte Helmert. *Modified PDRplan from Eriksson-Helmert, ICAPS 2020*. Mar. 2020. https://doi.org/10.5281/zenodo.3694110 (cited on page 17).

[18] Salomé Eriksson, Gabriele Röger and Malte Helmert. 'A Proof System for Unsolvable Planning Tasks'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 28.1 (June 2018), pages 65–73. https://doi.org/10.1609/icaps.v28i1.13899 (cited on pages a, 3, 7, 8, 11, 16, 69).

[19] Salomé Eriksson, Gabriele Röger and Malte Helmert. 'Unsolvability Certificates for Classical Planning'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 27.1 (June 2017), pages 88–97. https://doi.org/10.1609/icaps.v27i1.13818 (cited on pages a, 6–8, 11).

[20] Richard E. Fikes and Nils J. Nilsson. 'Strips: A new approach to the application of theorem proving to problem solving'. In: *Artificial Intelligence* 2.3 (1971), pages 189–208. ISSN: 0004-3702. https://doi.org/10.1016/0004-3702(71)90010-5 (cited on page 4).

[21] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun and Daniel Weld. *PDDL - The Planning Domain Definition Language*. Aug. 1998 (cited on pages 17, 27).

[22] Claudia Grundke. 'Extending SymPA with Unsolvability Certificates'. Bachelor's thesis. University of Basel, 2020. https://ai.dmi.unibas.ch/papers/theses/grundke-bachelor-20.pdf (cited on page 17).

[23] Patrik Halsum. *INVAL*. https://github.com/patrikhaslum/INVAL (visited on 29/07/2025) (cited on page 17).

[24] Malte Helmert. 'The Fast Downward Planning System'. In: *Journal of Artificial Intelligence Research* 26 (July 2006), pages 191–246. ISSN: 1076-9757. https://doi.org/10.1613/jair.1705 (cited on pages 17, 68).

[25] Richard Howey, Derek Long and Maria Fox. 'VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL'. In: *16th IEEE International Conference on Tools with Artificial Intelligence*. 2004, pages 294–301. https://doi.org/10.1109/ICTAI.2004.120 (cited on page 17).

[26] Leonardo de Moura and Sebastian Ullrich. 'The Lean 4 Theorem Prover and Programming Language'. In: *Automated Deduction - CADE 28: 28th International Conference on Automated*

Deduction, Virtual Event, July 12-15, 2021, Proceedings. Berlin, Heidelberg: Springer-Verlag, 2021, pages 625–635. ISBN: 978-3-030-79875-8. https://doi.org/10.1007/978-3-030-79876-5_37 (cited on pages 3, 18).

[27] Esther Mugdan, Remo Christen and Salomé Eriksson. *Benchmarks, Code and Data from Mugdan et al, ICAPS 2023*. Mar. 2023. https://doi.org/10.5281/zenodo.7733612 (cited on page 17).

[28] Esther Mugdan, Remo Christen and Salomé Eriksson. 'Optimality Certificates for Classical Planning'. In: *Proceedings of the International Conference on Automated Planning and Scheduling* 33.1 (July 2023), pages 286–294. https://doi.org/10.1609/icaps.v33i1.27206 (cited on pages 6, 16, 68).

[29] *Output of the Fast Downward translator*. https://www.fast-downward.org/latest/documentation/translator-output-format/ (visited on 24/11/2025) (cited on page 27).

[30] *Planning.wiki - The AI Planning & PDDL Wiki*. https://planning.wiki/ (visited on 24/11/2025) (cited on pages 17, 27).

[31] *Rush Hour*. https://rushhourgame.thinkfun.com/ (visited on 29/11/2025) (cited on page 4).

[32] Marcel Steinmetz and Jörg Hoffmann. 'State space search nogood learning: Online refinement of critical-path dead-end detectors in planning'. In: *Artificial Intelligence* 245 (2017), pages 1–37. ISSN: 0004-3702. https://doi.org/10.1016/j.artint.2016.12.002 (cited on pages 11, 16, 17).

[33] Martin Suda. 'Property directed reachability for automated planning'. In: *Journal of Artificial Intelligence Research* 50.1 (May 2014), pages 265–319. ISSN: 1076-9757. https://doi.org/10.5555/2693068.2693076 (cited on page 16).

[34] Álvaro Torralba. 'SymPA : Symbolic Perimeter Abstractions for Proving Unsolvability'. In: 2016. https://api.semanticscholar.org/CorpusID:51735243 (cited on page 17).

[35] *VeriPB*. https://gitlab.com/MIAOresearch/software/VeriPB/ (cited on page 18).

[36] David Wang and Mohammad Abdulaziz. *Formally Verified Certification of Unsolvability of Temporal Planning Problems*. 2025. https://doi.org/10.48550/arXiv.2510.10189 (cited on page 17).

[37] Eshel Yaron. *Binary Decision Diagrams in Lean 4*. https://github.com/eshelyaron/lean4-bdd (visited on 27/06/2025) (cited on page 68).

[38] Eshel Yaron. 'Lean Binary Decision Diagrams'. Masters's thesis. University van Amsterdam, 2025. https://msclogic.illc.uva.nl/theses/archive/publication/5489/Lean-Binary-Decision-Diagrams (cited on page 68).

Appendices

A.1 Full definition of Derivation

```
inductive Derivation {n} (pt : STRIPS n) : (conclusion : Prop) → Type 1
| B1 R [Formalism pt R] {S S'} :
  IsLiteralInter pt R S →
  IsLiteralUnion pt R S' →
  (S ⊆ S') →
  Derivation pt (S ⊆ S')
| B2 R [Formalism pt R] {S S'} :
  IsProgrInter pt R S →
  IsLiteralUnion pt R S' →
  (S ⊆ S') →
  Derivation pt (S ⊆ S')
| B3 R [Formalism pt R] {S S'} :
  IsRegrInter pt R S →
  IsLiteralUnion pt R S' →
  (S ⊆ S') →
  Derivation pt (S ⊆ S')
| B4 R R' [Formalism pt R] [Formalism pt R'] {S S'} :
  IsLiteral pt R S →
  IsLiteral pt R' S' →
  S ⊆ S' →
  Derivation pt (S ⊆ S')
| B5 (A A' : Actions n) : A ⊆ A' → Derivation pt (A ⊆ A')
| ED : Derivation pt (Dead pt ∅)
| UD S S' :
  Derivation pt (Dead pt S) →
  Derivation pt (Dead pt S') →
  Derivation pt (Dead pt (S ∪ S'))
| SD S S' :
  Derivation pt (Dead pt S') →
  Derivation pt (S ⊆ S') →
  Derivation pt (Dead pt S)
| PG S S' :
  Derivation pt (pt.progression S pt.actions ⊆ S ∪ S') →
  Derivation pt (Dead pt S') →
```

```
  Derivation pt (Dead pt (S ∩ pt.goal_states)) →
  Derivation pt (Dead pt S)
| PI S S' :
  Derivation pt (pt.progression S pt.actions ⊆ S ∪ S') →
  Derivation pt (Dead pt S') →
  Derivation pt ({pt.init} ⊆ S) →
  Derivation pt (Dead pt S^c)
| RG S S' :
  Derivation pt (pt.regression S pt.actions ⊆ S ∪ S') →
  Derivation pt (Dead pt S') →
  Derivation pt (Dead pt (S^c ∩ pt.goal_states)) →
  Derivation pt (Dead pt S^c)
| RI S S' :
  Derivation pt (pt.regression S pt.actions ⊆ S ∪ S') →
  Derivation pt (Dead pt S') →
  Derivation pt ({pt.init} ⊆ S^c) →
  Derivation pt (Dead pt S)
| CI : Derivation pt (Dead pt {pt.init}) → Derivation pt (Unsolvable pt)
| CG : Derivation pt (Dead pt pt.goal_states) → Derivation pt (Unsolvable pt)
| UR {α} (E E' : Set α) : Derivation pt (E ⊆ E ∪ E')
| UL {α} (E E' : Set α) : Derivation pt (E ⊆ E' ∪ E)
| IR {α} (E E' : Set α) : Derivation pt (E ∩ E' ⊆ E)
| IL {α} (E E' : Set α) : Derivation pt (E' ∩ E ⊆ E)
| DI {α} (E E' E'' : Set α) : Derivation pt ((E ∪ E') ∩ E'' ⊆ (E ∩ E'') ∪ (E' ∩ E''))
| SU {α} (E E' E'' : Set α) :
  Derivation pt (E ⊆ E'') →
  Derivation pt (E' ⊆ E'') →
  Derivation pt (E ∪ E' ⊆ E'')
| SI {α} (E E' E'' : Set α) :
  Derivation pt (E ⊆ E') →
  Derivation pt (E ⊆ E'') →
  Derivation pt (E ⊆ E' ∩ E'')
| ST {α} (E E' E'' : Set α) :
  Derivation pt (E ⊆ E') →
  Derivation pt (E' ⊆ E'') →
  Derivation pt (E ⊆ E'')
| AT S S' A A' :
  Derivation pt (pt.progression S A ⊆ S') →
  Derivation pt (A' ⊆ A) →
  Derivation pt (pt.progression S A' ⊆ S')
| AU S S' A A' :
  Derivation pt (pt.progression S A ⊆ S') →
  Derivation pt (pt.progression S A' ⊆ S') →
  Derivation pt (pt.progression S (A ∪ A') ⊆ S')
| PT S S' S'' A :
  Derivation pt (pt.progression S A ⊆ S'') →
  Derivation pt (S' ⊆ S) →
  Derivation pt (pt.progression S' A ⊆ S'')
| PU S S' S'' A :
```

```
  Derivation pt (pt.progression S A ⊆ S'') →
  Derivation pt (pt.progression S' A ⊆ S'') →
  Derivation pt (pt.progression (S ∪ S') A ⊆ S'')
| PR S S' A :
  Derivation pt (pt.progression S A ⊆ S') →
  Derivation pt (pt.regression (S'^c) A ⊆ S^c)
| RP S S' A :
  Derivation pt (pt.regression (S'^c) A ⊆ S^c) →
  Derivation pt (pt.progression S A ⊆ S')
```

A.2 Format for Certificate Parser

Each line of the certificate is expected to either be an action set expression, a state set expression, a piece of knowledge or a comment (starting with #). In what follows we use `<AID>`, `<SID>` and `<KID>` to denote action set, state set and knowledge IDs respectively. The first action set expression has ID 0, the second one 1, etc. and similar for state set expressions and knowledge.

Action set expressions have the following formats, where the action set ID after `a` is the ID of the action set itself. The action IDs in the list of actions correspond with the indexes of the actions in the STRIPS description of the planning task, see page 24.

`a <AID> b <amount of actions> <list of action IDs>`	`(list of actions)`
`a <AID> u <AID left> <AID right>`	`(union of actions)`
`a <AID> a`	`(set of all actions)`

The possible formats for a state set expression are shown below, where the first state set ID of each line is the ID of the state set itself:

`e <SID> c e`	`(constant empty set)`
`e <SID> c i`	`(constant initial state set)`
`e <SID> c g`	`(constant goal set)`
`e <SID> b <bdd_filename> <bdd_index>`	`(bdd set)`
`e <SID> t <discription in DIMACS>`	`(horn set)`
`e <SID> e <TODO>`	`(MODS set)`
`e <SID> n <ID of negated state set>`	`(negation)`
`e <SID> i <SID left> <SID right>`	`(intersection)`
`e <SID> u <SID left> <SID right>`	`(union)`
`e <SID> p <SID> <AID>`	`(progression)`
`e <SID> r <SID> <AID>`	`(regression)`

Knowledge expressions can have the formats listed below. The knowledge ID after `k` is the ID of the knowledge itself. For dead knowledge, the ID after `d` is the ID of state set that is dead, and for subset knowledge the IDs after `s` are the IDs corresponding to the left and right state set. The knowledge IDs after the rules are the IDs of the premises.

k <KID> d <SID> ed	(empty set dead)
k <KID> d <SID> ud <KID> <KID>	(union dead)
k <KID> d <SID> sd <KID> <KID>	(subset dead)
k <KID> d <SID> pg <KID> <KID> <KID>	(progression goal)
k <KID> d <SID> pi <KID> <KID> <KID>	(progression initial)
k <KID> d <SID> sd <KID> <KID> <KID>	(regression goal)
k <KID> d <SID> pg <KID> <KID> <KID>	(regression initial)
k <KID> u ci <KID>	(conclusion initial)
k <KID> u cg <KID>	(conclusion goal)
k <KID> s <SID> <SID> urs	(union right state)
k <KID> s <AID> <AID> ura	(union right action)
k <KID> s <SID> <SID> uls	(union left state)
k <KID> s <AID> <AID> ula	(union left action)
k <KID> s <SID> <SID> irs	(intersection right state)
k <KID> s <SID> <SID> ils	(intersection left state)
k <KID> s <SID> <SID> dis	(distributivity state)
k <KID> s <SID> <SID> sus <KID> <KID>	(subset union state)
k <KID> s <AID> <AID> sua <KID> <KID>	(subset union action)
k <KID> s <SID> <SID> sis <KID> <KID>	(subset intersection state)
k <KID> s <SID> <SID> sts <KID> <KID>	(subset transitivity state)
k <KID> s <AID> <AID> sta <KID> <KID>	(subset transitivity action)
k <KID> s <SID> <SID> at <KID> <KID>	(action transitivity)
k <KID> s <SID> <SID> au <KID> <KID>	(action union)
k <KID> s <SID> <SID> pt <KID> <KID>	(progression transitivity)
k <KID> s <SID> <SID> pu <KID> <KID>	(progression union)
k <KID> s <SID> <SID> pr <KID>	(progression regression)
k <KID> s <SID> <SID> rp <KID>	(regression progression)
k <KID> s <SID> <SID> b1	(basic statement 1)
k <KID> s <SID> <SID> b2	(basic statement 2)
k <KID> s <SID> <SID> b3	(basic statement 3)
k <KID> s <SID> <SID> b4	(basic statement 4)
k <KID> s <AID> <AID> b5	(basic statement 5)