Planning Problems as Types, Plans as Programs: A Dependent Types Infrastructure for Verification and Reasoning about Automated Plans in Agda

Alasdair Hill

Submitted for the degree of
Doctor of Philosophy

Heriot-Watt University

Department of Computer Science,
School of Mathematical and Computer Sciences.

June, 2023

The copyright in this thesis is owned by the author. Any quotation from the thesis or use of any of the information contained in it must acknowledge this thesis as the source of the quotation or information.

Abstract

Historically, the Artificial Intelligence and programming language fields have had a mutually beneficial relationship. Typically, theoretical results in the programming language field have practical utility in the Artificial Intelligence field. One example of this that has roots in both declarative languages and theorem proving is AI planning. In recent years, new programming languages have been developed that are founded on dependent type theory. These languages are not only more expressive than traditional programming languages but also have the ability to represent and prove mathematical properties within the language. This thesis will explore how dependently typed languages can benefit the AI planning field. On one side this thesis will show how AI planning languages can be enriched with more expressivity and stronger verification guarantees. On the other, it will show that AI planning is an ideal field to illustrate the practical utility of largely theoretical aspects of programming language theory. This thesis will accomplish this by implementing multiple inference systems for plan validation in the dependently-typed programming language Agda. Importantly, these inference systems will be automated, and embody the Curry-Howard correspondence where plans will not only be proof-terms but also executable functions. This thesis will then show how the dependently-typed implementations of the inference systems can be further utilised to add enriched constraints over plan validation.

I dedicate this thesis to my family and friends. I especially dedicate this thesis to Kirsty for her continuous love and support during this challenging time.

Acknowledgements

To start with I want to thank my supervisors Ekaterina Komendantskaya and Ron Petrick for all the help that they have given me throughout this process. Their joint insight greatly helped me navigate this interdisciplinary thesis. I would also like to thank all of the other academics I have worked with throughout this process with special mentions to Joe Wells, Matthew Daggitt, and František Farka.

I am also grateful for all the support my family has given me throughout this process. I want especially to thank my Dad for forcing me to write up the final parts of my thesis. I also want to mention my appreciation for all my friends in my extended office. I wish this time hadn't been cut short.

I also want to mention my appreciation to Simon Thompson for his support and understanding during the final months of my thesis write-up.

Finally, I want to thank my partner Kirsty for being by my side throughout this process. I can't imagine having completed my thesis without her.

Inclusion of Published Works Form

Please note you are only required to complete this form if your thesis contains published works. If this is the case, please include this form within your thesis before submission.

Declaration

This thesis contains one or more multi-author published works. I hereby declare that the contributions of each author to these publications is as follows:

Citation details	Schwaab, Christopher, Ekaterina Komendantskaya, Alasdair Hill, Franti˘sek Farka, Ronald Petrick, Joe Wells, and Kevin Hammond. “ProofCarrying Plans.” In International Symposium on Practical Aspects of Declarative Languages, pp. 204-220. Springer, Cham, 2019.
Author 1	Contributed many ideas to the initial Agda formalisation and was the main author of the paper.
Author 2	Secondary author of the paper who wrote a lot of the paper.
Author 3	Contributed to the research and Agda formalisation of the paper including writing some of the soundness proofs.

Citation details	Hill, Alasdair, Ekaterina Komendantskaya, and Ronald PA Petrick. “Proof-Carrying Plans: a Resource Logic for AI Planning." In Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming, pp. 1-13. 2020.
Author 1	The primary author of paper and did the research.
Author 2	Advisor for the primary author.

Citation details	Hill, Alasdair, Ekaterina Komendantskaya, Matthew L. Daggitt, and Ronald PA Petrick. ``Actions you can handle: dependent types for AI plans." In Proceedings of the 6th ACM SIGPLAN International Workshop on Type-Driven Development, pp. 1-13. 2021.
Author 1	The primary author of paper and did the research.
Author 2	Advisor for the primary author.

Please included additional citations as required.

Contents

1 Introduction **1**

1.1 Verification of AI Planning 1

1.2 Type-Based Verification 4

1.3 Thesis Aims 7

1.4 Contributions 8

1.4.1 Declaration of Authorship 10

2 Background **12**

2.1 First-Order Logic 12

2.2 The Planning Problem 15

2.3 Planning Domain Definition Language (PDDL) 21

2.3.1 PDDL Domain Definition 21

2.3.2 PDDL Problem Definition 23

2.3.3 Declarative Semantics for PDDL 24

2.3.4 Solving a PDDL Planning Problem 26

2.4 Curry-Howard Correspondence 29

2.5 Agda Introduction 31

2.5.1 Agda Basics 32

2.5.2 Dependent Types 36

2.6 Summary 38

3 Literature Review **39**

3.1 Programming Languages 39

3.1.1 Type-Based Verification 39

3.1.2 Semantics 41

3.1.3 Resource Logic 42
3.1.4 Automated Theorem Proving 42
3.2 AI Planning 44
3.2.1 Logic for AI Planning 44
3.2.2 PDDL Versions 45
3.2.3 Embedding Planning in Other Logics 46
3.2.4 Automated Theorem Provers as Planners 47
3.2.5 Verification of AI Planning 48
3.3 Programming Languages and AI Verification 49
3.3.1 Formal Verification in AI 49
3.3.2 Discussion - Formal Verification in AI Planning 49

4 Planning Problems as Types STRIPS/Operational Approach 51
4.1 Introduction 51
4.2 Example: Proof-Carrying PDDL 52
4.3 Planning Problems as Types 53
4.3.1 Formal Language 53
4.3.2 Operational Semantics, States and Types 57
4.4 Plans as Proof Terms 60
4.4.1 Inference Rules for Planning Problems 60
4.4.2 Computational Characterisation of Plans: Soundness of Plan Execution 65
4.5 Agda Formalisation 69
4.5.1 Approach to Formalisation 69
4.5.2 Verifying and Executing Plans in Agda 73
4.6 Approach to State Consistency 75
4.7 Discussion 76
4.8 Related Work 77

5 Resource Logic for AI Planning 79
5.1 Introduction 79
5.1.1 Results of this chapter by means of an example 81
5.2 The PCPR Logic 83

5.2.1 Syntax of PCPR Logic 83
5.2.2 Subtyping (order on states) 85
5.2.3 Evaluation of Constraint Lists 86
5.2.4 PCPR Logic Rules 87
5.3 Soundness of the PCPR Logic 92
5.4 Lessons Learnt: Effects and States 96
5.5 Agda Formalisation 99
5.6 Automation 102
5.6.1 PDDL to Agda Translation 103
5.6.2 Automation of Derivation for PCPR Logic 106
5.6.3 Automation of the Derivation for PCP Logic 108
5.6.4 Evaluation of the Library Performance 110
5.6.5 Extraction of Plans to Executable Code 113
5.6.6 Applicability to PDDL Domains 113
5.7 Discussion 115
5.8 Related Work 117

6 Dependently Typed Enrichment for AI Plan Verification 119
6.1 Introduction 119
6.1.1 Verifying Extrinsic Properties 120
6.1.2 The Technical Approach 122
6.2 PCP* Logic - Validating Intrinsic Properties of a Plan 123
6.3 Running Example for Extrinsic Verification 126
6.3.1 PDDL to Agda Translation 127
6.3.2 Verification and Evaluation of Plans 131
6.3.3 Expressivity of PDDL 132
6.4 Verifying Extrinsic Properties 133
6.4.1 Example 1: Fuel Consumption 134
6.4.2 Example 2: Fairness 137
6.4.3 Concrete Fairness Example 142
6.4.4 Example 3: Universal Properties 144
6.5 Applications of Agda Library 146
6.6 Discussion 148

6.7 Related Work 148

7 Conclusions **151**

7.1 Challenges 152

7.2 Future Work 153

7.3 Reflection 154

7.4 Supporting Code 155

Bibliography **156**

A Inconsistency Example **170**

B PDDL Examples **172**

B.1 Blocks World 172

B.1.1 Domain Definition 172

B.1.2 Problem Definitions 174

B.2 Logistics 175

B.2.1 Domain Definition 175

B.2.2 Problem Definitions 177

B.3 Satellite 178

B.3.1 Domain Definition 178

B.3.2 Problem Definitions 180

B.4 MPrime 187

B.4.1 Domain Definition 187

B.4.2 Problem Definition 189

Chapter 1

Introduction

Historically, the Artificial Intelligence and programming language fields have had a mutually beneficial relationship. Typically, theoretical results in the programming language field have practical utility in the Artificial Intelligence field. One example of this that has roots in both declarative languages and theorem proving is AI planning [50]. In recent years, new programming languages have been developed that are founded on dependent type theory. These languages are not only more expressive than traditional programming languages but also have the ability to represent and prove mathematical properties within the language. This thesis will explore how dependently typed languages can benefit the AI planning field. On one side this thesis will show how AI planning languages can be enriched with more expressivity and stronger verification guarantees. On the other, it will show that AI planning is an ideal field to illustrate the practical utility of largely theoretical aspects of programming language theory.

1.1 Verification of AI Planning

AI Planning is a research area within AI that studies the automated generation of plans from symbolic domain and problem specifications [39]. AI planners came into existence in the 1970s as an intersection between general problem solvers [31], the situation calculus [77] and theorem proving [43].

Typically, a planning domain is represented by an abstract description of the *world* and a set of actions that can be used to alter the world states. *Actions* are

typically defined by giving:

- *preconditions* that denote the conditions in which an action can be applied to a state.
- *effects* which denote how the action will affect the state it is applied to.

Planning problems in the domain consist of the *initial state* of the world and a *goal state*.

One canonical AI planning domain is *Blocks World* which describes a world containing blocks on a table and actions that rearrange the blocks. A planning problem for this domain would set the blocks in a specific position and have a goal for the blocks to be rearranged in some other position.

Given a domain and problem, a planner seeks to produce a sequence of actions, referred to as a *plan*, that transforms the initial state into a state that achieves the goals set out in the goal state. This transformation is done by applying the first action of the plan to the initial state and then sequentially applying each action in the plan to the resultant state. A planner therefore has to ensure that it produces a plan where all actions can be sequentially applied to the initial state given their preconditions. If the plan can be applied to the initial state and the resultant state of the application achieves the goal state then the plan is *valid*.

Over the years many different planning languages have been created to represent planning problems leading to the creation of the *Planning Domain Definition Language (PDDL)* [78] which aimed to standardise them. PDDL can represent a vast number of planning problems with the latest versions [37, 38] supporting types, numerical functions, equality, conditionals, concurrency, temporal planning and more. Applications include: reasoning about knowledge, belief and causality, planning allocation of resources, modelling perception of the real world, program synthesis and implementations of multi-agent systems [50, 126].

Many of these applications of planning are used in real-world environments where the verification of plan correctness is essential for successful and safe operation.

Verification and validation of AI planning languages is a multifaceted and rich field of research. One may verify domain models, planning algorithms, the produced plans and the execution of plans. The range of methodologies used to achieve this range from standard testing [8, 106], to more 'lightweight" verification [57, 72] and

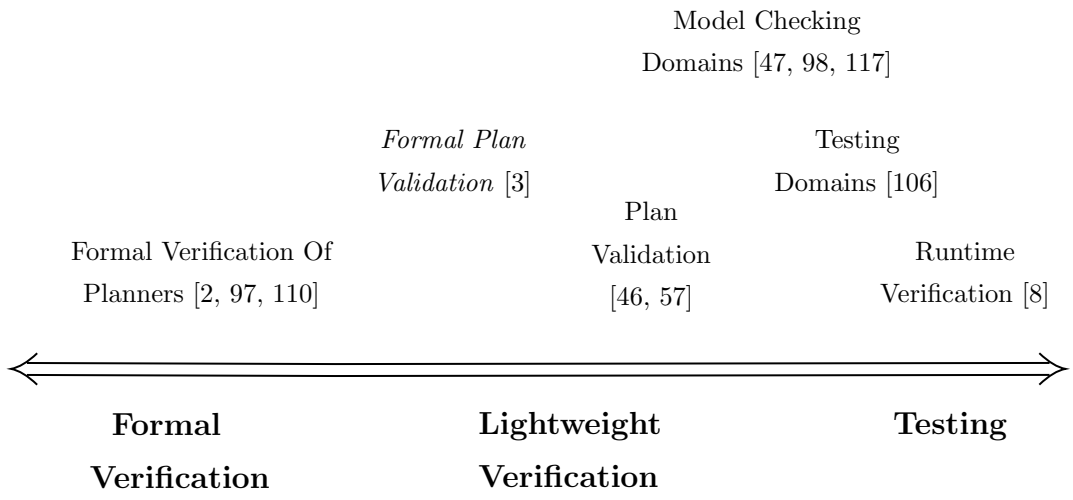

Figure 1.1: *Spectrum of the verification methodologies used in AI planning. The work presented in this thesis is part of Formal Plan Validation research.*

even formal verification [2, 97, 110]. The most common approaches used in AI planning verification sit between lightweight verification and testing as shown in Figure 1.1.

This thesis makes a distinction between more "heavyweight" formal verification approaches versus "lightweight" verification approaches through the idea of the verification of planners versus the verification of plans. Both the verification of planners and plans ultimately want to ensure the correctness of plans. The formal verification of a planner, however, is a complicated and large-scale verification task. In contrast, the verification of a plan is a more lightweight approach and contains the same level of confidence over the correctness of a plan. Plan verification is more lightweight because it is just concerned with the validity of an individual plan according to some planning model whereas the verification of a planner has to ensure the validity of all possible produced plans according to some planning model. Despite being more lightweight, the verification of plans has an equivalent end result to the verification of planners when it comes to the correctness of plans. This thesis argues that the lightweight approach is more flexible as it allows for the integration with a wide range of planners.

To be able to verify existing validators it would be necessary to prove that the implementation of the validator matches the semantics of PDDL. This is very difficult to do as the relation between the intended semantics of PDDL and the actual implementation of the validator is often unclear. One reason for this is that plan

validators often have many performance optimisations that obfuscate their semantics. Another issue is that the semantics of PDDL are not defined in enough detail to be used for formal verification. Many of its semantics are open to interpretation, with planners interpreting parts of PDDL differently.

One example of unclear semantics in PDDL can be found in the effects of actions. It is possible in PDDL to define an effect that will both add and remove the same fact from a state. This effect has undefined behaviour because PDDL does not define the order in which effects should be executed. This means that different planners will come to the conclusion that the fact will either exist or not exist in the resultant state. This is discussed in more detail in Section 5.4.

To be able to create a verified plan validator this thesis will therefore have to define new semantics for PDDL, an implementation of a plan validator, as well as soundness proofs for this validator. This thesis will create a verified plan validator for the STRIPS subset of PDDL 1.2 [78] excluding quantification and disjunction.

A Type-Based approach to AI plan verification, introduced in this thesis, promises to deliver features that existing lightweight approaches [46, 57] do not have. Formal guarantees can be given about the validator itself as well as the plans that it validates. It is also possible to create a validator that can produce checkable evidence for validated plans. In contrast, lightweight approaches that attempt to mimic the semantics of PDDL without obfuscation still suffer from errors. One example of this can be found in Patrik Halsum's INVAL [46] which has known errors and non-termination on some plans [3].

Considering that many applications of AI planning can be executed in the real world and have serious safety implications, this higher standard of formalisation is desirable. The next section will explore why taking a type-based approach to plan validation is a natural solution for the problem.

1.2 Type-Based Verification

Type-based verification is a field of research focused on using the rich type systems of modern programming languages to verify and prove properties of various mathematical problems [42] and computer systems [67, 82, 86]. This is a natural evolution

in the field of programming languages as declarative programming languages have long provided convenient formalisms for knowledge representation and reasoning, ranging from Lisp and Prolog in the 1960s-1980s to modern SMT solvers [27, 79], model checkers [91], and automated planners [32, 78]. Common features of such languages typically include a clear logic-based syntax, a well-understood declarative semantics, and an inference engine that produces sound results with respect to the semantics.

Type-based verification expands on the capability of programming languages by enabling them to be used as interactive theorem provers. An interactive theorem prover is a computer tool that allows users to specify and prove mathematical properties. Typically these systems aid in the mechanisation of proofs by supplying goal information and auto-completing proofs where possible.

This is possible through exploiting the *Curry-Howard correspondence* [118, 122], that states that every proof in intuitionistic first-order logic can be represented as a computable function. In the case of type-based verification first-order formulae are seen as *types*, and proofs are seen as terms that inhabit those types. This means that the expressivity of the typing system limits what mathematical properties you can verify. Until recently, the significance of the Curry-Howard correspondence has been predominantly theoretical; this is because programming languages did not have expressive enough type systems that could fully utilise this idea. Dependently-typed languages that offer the necessary expressivity have now become available and practical, e.g., Coq, Agda, and Idris. The key difference in dependently-typed languages is that types can contain arbitrary values and appear as arguments to the results of ordinary functions. This means that types can not only contain values but also that types can contain propositions about these values. Dependently-typed languages can therefore be used to write programs that carry proofs about their properties. This development has made it possible to re-open the discussion of the actual practical value of the Curry-Howard correspondence.

With the advent of dependently-typed languages, it is possible to encode and prove properties within a programming language. Overall there are many advantages of taking a type-based approach to formalising an AI plan validator. In current lightweight approaches, a user has to trust the equivalence of the semantics of the

implementation of the validator and the clear and concise semantics of PDDL. By taking a type-based approach this thesis shows that it is possible to implement the clear and concise semantics as types and prove them sound in the same language which reduces the chance for errors. Using this approach also gives the ability to prove further sanity-checking theorems within the programming language. For example, a consistency theorem is proven in Figure 5.6 that ensures that a predicate can never be both true and false in the same state. Furthermore, by using this approach it is possible to implement the validator in a *proof-carrying code* style [86], where all relevant verification properties are directly embedded in the source code.

Due to their ability to prove mathematical properties dependently-typed languages also give rise to interactive theorem provers. A related, albeit not dependently-typed, attempt at formalising a PDDL plan validator [3] has been done in the interactive theorem prover Isabelle [94]. Isabelle combines functional programming language with the ability to express higher-order logic. In [3] a verified plan validator was created by defining the validator as a program and then proving properties ensuring the validator is sound. This verified validator was then extracted.

This thesis will take a proof-carrying code approach instead where plan validation is defined as a typing relation rather than a program. A plan is then seen as a proof-term which proves that it is possible to reach the goal state from the initial state given the domain. Since plan validation is proven within the Agda system it is also possible to define the computation of a plan as well within the system. This will allow this thesis to fully realise the Curry-Howard approach as plans will not only be proof-terms but also executable functions. This is advantageous because it will allow for the validation of a plan to contain guarantees about its execution. Critically, the proof-carrying code approach taken in this thesis is not possible in Isabelle because it does not contain proof-terms.

Dependent types will also be used in this thesis to expand on the verification capabilities of PDDL by adding additional verification constraints at the type level of functions describing the execution of plans. This will allow for additional properties to be ensured during plan execution, some of which are inexpressible in PDDL. This will be further detailed in the contributions section and throughout the rest of the thesis.

1.3 Thesis Aims

This thesis aims to demonstrate how modern programming language theory can benefit the field of AI planning and in turn show that AI planning is a natural domain to showcase the practical benefits of dependent types. This thesis sets the following aims in order to accomplish this:

Aim 1. Demonstrate why taking a Curry-Howard approach to plan validation is a useful and natural solution.

Aim 2. Explore how dependently-typed programming languages can be used to improve state-of-the-art plan validation.

To accomplish these aims this thesis sets the following objectives:

Objective 1. Provide new formal foundations for PDDL planning that can be used to verify PDDL plans as well as improve the understanding of them. These foundations should be compatible with existing PDDL domains and problems as well as contain the additional details that are necessary for them to be formalised in Agda.

Objective 2. Demonstrate a Curry-Howard view to plan validation. To accomplish this, a plan should not only be seen as a proof of a planning problem but also as an executable function.

Objective 3. Develop a methodology on how to implement and verify declarative systems in a dependently-typed language. This should provide a basis for future formalisms of declarative systems in dependently-typed languages.

Objective 4. Implement a new verified plan validator for PDDL. This plan validator should give additional proof evidence of validated plans in order to give increased confidence over existing methods.

Objective 5. Introduce automation into the defined systems that show how to mechanise the plan validation process. This automation should be designed so that non-expert users can utilise the Agda formalism to verify plans.

Objective 6. Show how it is possible to utilise richer dependently-typed languages to add additional verification properties as well as demonstrate why these properties are better suited outwith the standard PDDL planning problem. This should include examples of properties that are inexpressible in PDDL to showcase the power of the implementation.

1.4 Contributions

This thesis makes the following contributions:

Chapter 4

Formal Semantics for AI Planning

In order to verify AI planning languages, this thesis defines concrete semantics for AI planning. Section 2.3.3 defines declarative semantics describing entailment in PDDL. Chapter 4 introduces operational semantics for AI planning that describe how to execute AI plans. A representation language based on first-order logic for AI planning problems is defined in Chapter 4. The semantics defined in this thesis adds additional detail and rigour to PDDL's specifications in order to meet Objective 1.

Formalisation of Inference Systems for AI Plan Validation

This thesis defines and formalises multiple inference systems for plan verification. These inference systems can be used to verify plans for a subset of planning problems that can be expressed in PDDL. Crucially all of the inference systems have an accompanying implementation in the dependently-typed programming language Agda. All of the systems have also been proven sound in Agda. The implementation of the inference systems meets Objective 3. The Agda formalisation of these inference systems can validate plans for PDDL planning problems as well as produce proof-terms for why they are valid which addresses Objective 4. Chapter 4 introduces the first inference system called *PCP logic* along with the various definitions of soundness given in this thesis. PCP logic is proven sound. The chapter also gives details of the Agda formalism of PCP logic.

Curry-Howard Approach to Plan Validation

This thesis will show that the Planning Domain Definition Language (PDDL) is a natural domain for the Curry-Howard implementation of declarative reasoning. In particular, specifications of planning problems that are usually written in first-order logic can be expressed naturally as types, and executable plans that are generated by PDDL can be formalised as programs that inhabit those types. Type checking thus verifies that correct executable programs are generated from specifications *via* the automated planning tool. This approach will showcase a practical use case of the Curry-Howard correspondence and address Objective 2. A detailed description of the Curry-Howard approach to plan validation will be given in Chapter 4. All of the inference systems presented in this thesis are implemented in a Curry-Howard style.

Chapter 5

Resource Logic for AI planning

Chapter 5 defines and create a new logic called Proof-Carrying Plan Resource logic (PCPR logic). This logic has clear and intuitive formal semantics which helps to clarify the computational properties of AI plans which further addresses Objective 1. This logic utilises Hoare triples to represent AI planning problems and explores the utility of a resource logic based approach to plan validation. In particular, PCPR logic explores the utility of using a frame rule in the inference system as historically done in early AI planning. Chapter 5 defines PCPR logic as well as proves it sound relative to the declarative and operational semantics given defined in this thesis.

Automating plan validation in a dependently-typed language

This thesis introduces automation for all of the inference systems that it defines. This automation allows a user to validate a plan using the inference systems defined Agda without having to write any Agda code. This is done by automatically converting PDDL domains and problems into Agda code and then automatically generating a proof for a given plan using the defined inference rules. This enables the wider planning community that are less familiar with dependently-typed programming languages to be able to use the inference systems this thesis defines with minimal learning. This contribution addresses Objective 5. Chapter 5 describes the

various strategies for automation.

Chapter 6

Formalisation of Inference Systems for AI Plan Validation

Chapter 6 introduces a modified version of *PCP logic* called *PCP* logic* addresses the weaknesses of the other systems. This inference system is also proven sound in Agda, automated and compared with the other inference systems presented in this thesis.

Verification of extrinsic properties of AI plans

Chapter 6 introduces the notion of extrinsic verification properties to PDDL planning problems. This is done by first making a distinction between intrinsic and extrinsic properties. Chapter 6 then identifies ideal extrinsic properties such as inexpressible properties, unavailable properties and probable properties. Furthermore, Chapter 6 shows how it is possible to leverage the power of dependently-typed languages to add these extrinsic verification properties to the planning problem. This addresses the final objective of this thesis, Objective 6.

1.4.1 Declaration of Authorship

This thesis is the result of my own work. Several of the chapters presented in this thesis are formed from previous publications. All of the chapters have substantial differences from these publications. I am the third author in the "Proof-Carrying plans" publication however I was heavily involved in this research. Chapter 4 is substantially different and my own work. The declarative semantics given in Figure 2.3 is the same as in the "Proof-Carrying plans" paper.

Publications

- Chapter 4: Schwaab, Christopher, Ekaterina Komendantskaya, Alasdair Hill, František Farka, Ronald Petrick, Joe Wells, and Kevin Hammond. "Proof-Carrying Plans." In International Symposium on Practical Aspects of Declarative Languages, pp. 204-220. Springer, Cham, 2019. [114]
- Chapter 5: Hill, Alasdair, Ekaterina Komendantskaya, and Ronald PA Petrick. "Proof-Carrying Plans: a Resource Logic for AI Planning." In Proceedings of

the 22nd International Symposium on Principles and Practice of Declarative Programming, pp. 1-13. 2020. [53]

- Chapter 6: Hill, Alasdair, Ekaterina Komendantskaya, Matthew L. Daggitt, and Ronald PA Petrick. "Actions you can handle: dependent types for AI plans." In Proceedings of the 6th ACM SIGPLAN International Workshop on Type-Driven Development, pp. 1-13. 2021. [52]

Chapter 2

Background

This chapter will cover the necessary preliminaries to understand the rest of this thesis as well as introduce the running examples. This will include a variety of background knowledge from both AI planning and programming languages fields. A brief introduction will be given to first-order logic. The AI planning problem will be defined then an introduction will be given to the Planning Domain Definition Language (PDDL). This will be followed by an introduction to declarative semantics for planning, the Curry-Howard correspondence and the notion of dependent types.

2.1 First-Order Logic

Planning languages in general are based on a fragment of first-order logic. This chapter will now give a brief introduction to first-order logic, taking definitions from [71] and relating them to the Blocks World planning problem defined in Figure 2.2.

A first-order language describes the formulas that can be written in a first-order system known as *well-formed formulas*.

A *grammar* defines the set of all formulas that can be constructed in a language by defining production rules for the language. The grammar for a first-order language without functions is defined in Figure 2.1.

Definition 1 (First-Order Language)**.**

All first-order grammars presented in this thesis are parametrised over a language containing:

1. A set *Predicate* of predicate symbols with their arity.

Binary Connective	*Connective*	$::= \wedge \mid \vee \mid \rightarrow \mid \Longleftrightarrow$
Quantifier	*Quantifier*	$::= \forall \mid \exists$
Term	*Term* $\ni t, t_1, \ldots t_n$	$::= X \mid C$
Atomic Formula	*Atom* $\ni A$	$::= R\ (t_1, \ldots t_n)$
Formula	*Formula* $\ni F$	$::= A \mid (F)$
		$\mid \neg F$
		$\mid F$ *Connective* F
		$\mid$ *Quantifier* $X,\ F$

Figure 2.1: *First-Order Language*

2. A set *Contant* of constant symbols.

Metavariable R and notational variants such as R', R_1, R_2 etc. range over predicate symbols, metavariable c and notational variants such as c', c_1, c_2 etc. over constants, metavariable x and notational variants such as x', x_1, x_2 etc. over variables, metavariable t and notational variants such as t', t_1, t_2 etc. over terms which are either variables or constants, therefore

$$t ::= x \mid c$$

The informal semantics of the quantifiers and connectives is as follows: $\neg$ is negation, $\wedge$ is conjunction (and), $\vee$ is disjunction (or), $\rightarrow$ is implication and $\longleftrightarrow$ is equivalence. Also, $\exists$ is the existential quantifier, so that "$\exists x$" means "there exists an x", while $\forall$ is the universal quantifier, so that "$\forall x$" means "for all x". Thus the informal semantics of $\forall x((R(x) \wedge \neg R_1(x)) \rightarrow R_2(x))$ is "for every x, if $R(x)$ is true and $R_1(x)$ is false then $R_2(x)$ is true". A *literal* is an atomic formula or a negated atomic formula.

Example 1 (First-Order Formulae Example)**.** First-Order Formulae are simply any formulae that can be produced from the grammar rules in Figure 2.1. This includes formulae such as $R(x)$, $R(x) \wedge R(x_1)$ and $(\forall x\ (\exists y\ (R(x, x_1) \rightarrow R_1(x))))$.

Definition 2 (Variable Definitions with Quantifiers)**.**

The *scope* of $\forall x$ (resp. $\exists x$) in $\forall x\ F$ (resp. $\exists x\ F$) is F. A *bound* occurence of a variable in a formula is an occurrence immediately following a quantifier or an occurrence within the scope of a quantifier. Any other occurence of a variable is *free*. A *closed formula* is a formula with no free occurrences of any variable.

Example 2 (Quantifier Example)**.**

Given the formula $\forall x(R(x) \wedge R_1(x, x_1))$ all occurences of x are bound and x_1 is a free variable. The formula $\forall x(R(x) \wedge R_1(x, x_1))$ is not closed however it can be closed by adding quantification for the free variable x_1, e.g., $\forall x \forall x_1(R(x) \wedge R_1(x, x_1))$ is *closed*.

Definition 3 (Universal and Existential Closure)**.**

If F is a formula, then $\forall(F)$ denotes the *universal closure* of F, which is the closed formula obtained by adding a universal quantifier for every variable having a free occurrence in F. Similarly, $\exists(F)$ denotes the *existential closure* of F, which is obtained by adding an existential quantifier for every variable having a free occurrence in F.

Example 3 (Universal Closure Example)**.**

If F is $R(x) \wedge R_1(x, x_1)$, then $\forall(F)$ is $\forall x \forall x_1(R(x) \wedge R_1(x, x_1))$.

Remark on Notation 1

This thesis will use abbreviation $\overline{x}$ to denote a finite list $\{x_1, ... \ x_n\}$ of arbitrary length.

Definition 4 (Substitution)**.**

A substitution is a partial map from variables to constants. Given an atomic formula $R(\overline{x})$ the notation $R(\overline{x})[x_i \backslash c_i]$ will be used to denote a substitution of each occurrence of a variable x_i in $\overline{x}$ by a constant c_i. A *ground substitution* will map all variables in a formula to a constant and will be denoted by the symbols $\{\sigma, \sigma_1, \sigma_2, ...\}$.

Example 4 (Substitution Example)**.**

Given a constant c, the substitution of the variable x in the formula $R(x, x_1)$ is denoted by $R(x, x_1)[x \backslash c]$ and results in the formula $R(c, x_1)$. This formula is not ground as the variable x_1 remains.

Definition 5 (Herbrand Base)**.**

The Herbrand base is the set of all ground atomic formulae that can be formed from the constants of a first-order language.

The grammar presented contains no functions therefore there are no function symbols denoting functions. This is important because the Herbrand base of first-order language with function symbols is generally an infinite set however in the absence of function symbols it becomes a finite set if there are only finitely many predicate symbols and constants.

Example 5 (Herbrand Base Example)**.**

For a first-order language without function symbols and constants $\{c, c_1\}$ and atomic formulae $\{R(x), R_1(x, x_1)\}$ the Herbrand base is the set $\{R(c),$ $R(c), R_1(c, c), R_1(c, c_1), R_1(c_1, c), R_1(c_1, c_1)\}$.

This concludes the generic first-order language definitions that will be used in this thesis.

2.2 The Planning Problem

This section will give a general introduction to the planning problem using definitions from Ghallab et al. [39].

Example 6 (First-Order Language)**.** The language of the Blocks World domain and problem defined in Figure 2.2 is:

$$Predicate = \{handEmpty/0, holding/1, onTable/1, on/2, clear/1\}$$

$$Constant = \{a, b, c\}$$

Definition 6 (Planning Problem)**.**

A planning problem is represented by the triple (Σ, s_0, G) where Σ represents a planning domain, s_0 the initial state and G the goal.

A *state* is a representation of the world and contains information about the truth of all formulae. Typically a state is represented as a set of ground atomic formulae. A *goal* defines the acceptance condition for a planning problem and is typically

Definition of a Planning Domain:

```
(define (domain blocksworld)
    (:requirements :strips)
    (:predicates
        (handEmpty)
        (holding ?x)
        (onTable ?x)
        (on ?x ?y)
        (clear ?x) )

   (:action pickup_from_table
      :parameters
        (?x)
      :precondition
        (and (handEmpty)
             (onTable ?x)
             (clear ?x) )
      :effect
        (and (not (handEmpty))
             (not (onTable ?x))
             (holding ?x) ))
    (:action putdown_on_table
        ... )
    (:action pickup_from_stack
        ... )
    (:action putdown_on_stack
        ...))
```

Definition of a Planning Problem:

```
(define (problem blocksworld1)
  (:domain blocksworld)
  (:objects a b c)
  (:init (onTable a)
         (onTable b)
         (onTable c)
         (clear a)
         (clear b)
         (clear c)
         (handEmpty) )
  (:goal (and (on a b) (on b c))))
```

Figure 2.2: *Blocks World: a code snippet in PDDL defining a planning domain and problem.*

represented by a ground first-order formula. In this thesis a *planning domain* is represented by a *state-transition system.*

Definition 7 (State-Transition System [39])**.**

A *state-transition system* is a triple $\Sigma = (S, \alpha, \gamma)$, where

- S is a finite set of *states* in which the system may be.
- α is a finite set of *actions* that the actor may perform.
- $\gamma : S \times \alpha \rightarrow S$ is a partial function called the *state-transition function.* If (s, a) is in γ's domain (i.e., $\gamma(s, a)$ is defined), then a is *applicable* in s, with $\gamma(s, a)$ being the *predicted* outcome. Otherwise a is *inapplicable* in s.

In order to define the finite set of possible states in a state-transition system

it is necessary in the definition of a state-transition system to define what can be contained in a state. This is given by a set of constant names referred to as *objects* and set of n-ary relations over objects referred to as *predicates*. A *state* is defined as a set of ground atoms denoting all true formulae. The set of possible states is therefore the superset of all possible ground atomic formulae.

Definition 8 (Plan [39]).

A plan is a finite sequence of actions:

$$p = a_1; a_2, ...; a_n$$

An *empty plan* containing no actions is represented by [].

Definition 9 (Plan Application [39]).

A plan $p = a_1; a_2; ...; a_n$ is *applicable* in a state s_0 if there are states $s_1, ..., s_n$ such that $\gamma(s_{i-1}; a_i) = s_i$ for $i = 1, ..., n$. The result of plan application is:

$$\gamma(s_0, p) = s_n$$

.

The empty plan [] is applicable in every state s, with $\gamma(s, []) = s$

Informally, a state *satisfies* a goal if it meets the goal's acceptance condition. In this thesis, a goal's acceptance condition will be given by a conjunction of literals and negated literals. Given a state S and a goal G if all literals in G are true in S and all negated literals in G are false in S then S satisfies G. A formal definition for satisfaction is given in Definition 13.

Definition 10 (Planning Problem Solution [39]).

Given a planning problem (Σ, s_0, G), a solution for the problem is any plan applicable in s_0, $p = a_1; ...; a_n$, such that the state $\gamma(s_0, p)$ satisfies G.

Example 7 (The planning problem as Deterministic Finite Automata (DFA) [55]).

The above definitions of the planning problem as a State-Transition system could also be seen as a DFA $M = (Q, \Sigma, \delta, q_0, F)$ where:

- The set of states Q is the set of states in which the system may be.

- The alphabet Σ is the set of actions.
- The transition function δ describes all applicable action transitions. Note that this function could be made total by adding an error state.
- The initial state q_0 describes the initial state of the planning problem.
- The set of accept states F are all the states in which the goal is satisfied.

The language of the DFA describing a planning problem is therefore the set of all plans that achieve the goal.

This thesis only works with state-transition systems that are deterministic. The set of states and actions in a planning domain also have to be finite. In addition to this, two assumptions are made:

1. Changes of the world state only occur in response to actions.
2. The effects of an action can be predicted with certainty and match the modelled effects.

Both of these are standard assumptions in the AI planning field [39, 50, 85]. The first assumption excludes the possibility of external factors affecting the planning problem such as another actor acting on the world. The second assumption excludes the possibility of execution errors as well as nondeterministic actions.

One of the issues in defining a state-transition system is the definition of the state-transition function. The most simple implementation of this function is to create a lookup table that contains $\gamma(s, a)$ for every s and a however this quickly becomes infeasible with larger domains. This is because every possible state that an action can be applied to has to be defined.

Example 8 (State-transition implementation issues.)**.**

Given Figure 2.2, to define an action that picks up a block a from a table in a domain that has blocks $\{a, b, c\}$ the definition of γ would have to contain definitions for all possible states s that the action a is applicable in. This would include definitions for the states: $\{handEmpty, clear\ a, onTable\ a\}$, $\{handEmpty, clear\ a, onTable\ a, onTable\ b\}$, $\{handEmpty, clear\ a, onTable\ a, onTable\ b, onTable\ c\}$ among many other states.

It is intuitive to understand picking up some block a that is on the table is unaffected by whether some block b is also on the table however definitions need to

be given for both states in the state transition function. Since most actions in AI planning only make small local changes to the world, a more general representation would be more suitable. Ideally, the state transition function could be defined using *local reasoning* where only the atomic formula that are involved in an action are defined. The problem of local reasoning in AI planning is referred to as the *frame problem* [25, 48].

The Stanford Research Institute Problem Solver (STRIPS) [32, 69] provides a general representation for actions using local reasoning.

Definition 11 (Actions in STRIPS)**.**

An action is described by a triple (P, L_D, L_A) where:

- P is the preconditions of the action represented by a first-order formula.
- L_D is a delete list represented by a list of atomic formula.
- L_A is an addition list represented by a list of atomic formula.

An action description is typically parametrised by multiple variables:

$$\forall \overline{x}(P(\overline{x}), L_D(\overline{x}), L_A(\overline{x}))$$

In STRIPS instead of defining all states for which an action can be applied a precondition is given where an action can be applied to a state if it satisfies the precondition. The result of action application is computed by deleting all of the atomic formulae in the delete list from the state then adding all the atomic formulae from the addition list to the state. It is assumed that all atomic formulae not mentioned in the delete and addition lists are unaffected by the application of the action. This assumption is known as the *STRIPS assumption* [124] and is how STRIPS deals with the frame problem. STRIPS also utilises first-order formulae to create generic descriptions of actions. Note that all atomic formulae in STRIPS are given by object and predicate definitions in the same manner as the state-transition system.

Example 9 (STRIPS Action)**.**

The "pickup from table" action could be defined in STRIPS by:

$$P = handEmpty \wedge clear\ x \wedge onTable\ x$$

$$L_D = \{handEmpty, onTable\ x\}$$

$$L_A = \{holding\ x\}$$

Example 10 (STRIPS Action Application)**.**

This example will walk through the application of the pickup from table action. It is only possible to apply ground actions therefore the example definition in Example 9 will first be ground by the object a. This results in the following grounded action:

$$P = handEmpty \wedge clear\ a \wedge onTable\ a$$

$$L_D = \{handEmpty, onTable\ a\}$$

$$L_A = \{holding\ a\}$$

Given the initial state $\{handEmpty, clear\ a, onTable\ a, onTable\ b, onTable\ c\}$ the preconditions of the action are satisfied because all formula in P exist in the initial state. The action is therefore applicable in the given initial state. The first step of the application is to delete the formula contained in the delete list L_D from the initial state. This results in the state:

$$\{clear\ a, onTable\ b, onTable\ c\}$$

The action application is then completed by adding all the formula contained in the addition list L_A. The action application will therefore result in the state:

$$\{clear\ a, onTable\ b, onTable\ c, holding\ a\}$$

.

2.3 Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language is a representation language aimed at providing a common syntax for modelling planning problems. This section will introduce the syntactic representation of the planning problem using the subset of PDDL[78] that this thesis supports. Examples will be given using the classic *Blocks World* domain where a robot is tasked with rearranging blocks on a table. Figure 2.2 gives a PDDL definition of this problem.

PDDL splits a planning problem into a *domain* definition and a *problem* definition. The domain definition represents the state-transition system and the problem definition contains the initial state, goal and a set of constants referred to as objects.

Even though PDDL has its own syntax it is still a language that can be largely represented using the standard first-order language given in Figure 2.1.

Example 11 (First-Order PDDL)**.** In Blocks World both the `:predicates` definition as well as the use of predicates in general can be written using first-order formulae. For example, the predicate `onTable ?x` can simply be written in first-order logic as the atomic formula $onTable(x)$. Preconditions for actions could also be represented by a formula. For example, the preconditions for the *pickup_from_table* action could be represented by the formula $handEmpty \wedge onTable(x) \wedge clear(x)$.

The fragment of PDDL that will be used in this thesis does not include existentials, disjunction or implication so this thesis will define a subset of first-order logic that will be referred to as *PDDL formulae*. Note that the A in the below PDDL formulae definition refers to standard first-order atomic formulae defined in 2.1.

Definition 12 (PDDL Formulae)**.**

$$\text{PDDL Formulae} \quad Form \ni F, F_1 ... F_n \quad := A \mid \neg A \mid F \wedge F_1$$

Thus, PDDL formulae are conjunctions of literals.

2.3.1 PDDL Domain Definition

A PDDL domain definition includes definitions of requirements, predicates, actions and types. Definitions in PDDL use a Lisp like syntax where definitions are given

by a : followed by a name in brackets.

Example 12 (Definitions in PDDL)**.**

Predicates are defined by a definition in the form (: `predicates` *definition*).

Requirements are used to define the subset of PDDL required for the domain. This is given by simply giving a list of requirements in the form (: `requirements` :*name* ...). For example, the : `strips` requirement represents actions in a STRIPS style. All of the PDDL introduced in this thesis will be restricted to the STRIPS subset of PDDL which restricts the first-order formulae used in actions to just contain negation and conjunction. This thesis will also use the : `equality` and : `typing` extensions.

Predicates are a PDDL interpretation of a first-order logic predicate. The predicate definition defines the set of atomic formulae that are permissible for a domain. A predicate is defined by giving a predicate name as well as the number of arguments for that predicate. The number of arguments is given by a list of variables. Parameters are denoted by the ? symbol followed by a variable name.

Example 13 (Predicate)**.**

The predicate (`on` ?`x` ?`y`) defines a predicate named 'on' that has two parameters.

It should be noted that predicate names have no intrinsic meaning i.e. they are declared rather than defined. The meaning of predicates is given by how they are used in actions. Most PDDL definitions will however give meaningful names to predicates such as the `handEmpty` predicate in Blocks World that indicates the robot's hand is empty. For example, in Blocks World all of the actions that include picking up a block will require that the predicate `handEmpty` is true in the preconditions.

Actions are used to perform transformations on a world state. Actions are defined by giving a name as well as a definition for its parameters, preconditions and effects. All definitions except for the name are optional when defining actions. Parameters are defined by giving a list of variables. The parameter definition defines the list of variables that can be used in the subsequent definitions of Preconditions and Effects. In this thesis, both the Preconditions and Effects of an action can be defined by a PDDL formula. Note, that in PDDL negation is denoted by `not` and conjunction by `and`. An example `pickup_from_table` action is defined in Figure 2.2.

Just like in STRIPS preconditions denote requirements that have to be satisfied in order to apply an action. One notable difference in PDDL is that the delete and addition list from STRIPS is combined into one effect list. In an effect list the negated atomic formulae are used to denote items to delete from the state and atomic formulae are used to denote items to add to the state.

The *equality* requirement allows for the comparing of objects in a precondition of an action.

Example 14 (Equality)**.**

Assuming a precondition has parameters `?x` and `?y` they can be compared in a precondition using (= `?x` `?y`) or (`not` (= `?x` `?y`)) indicating that the parameters have to be equal or not equal respectively.

The *typing* requirement allows for types to be added to the domain definition. The typing definition is given by a list of names. Types can then be added to the parameters in the predicate and action definitions. This restricts the allowable bindings of the parameters for actions and predicates, to objects of the corresponding type.

Example 15 (Types)**.**

A block type could be added to Blocks World by the typing definition (: `types block`) then the predicate `holding` could assert that its parameter `?x` is a block by the definition (`holding` `?x` − `block`).

The PDDL typing extension also supports type hierarchy. For example, it is possible to define a cube type and assert that the block type is also a cube type. This thesis will not be using type hierarchy so it will not be defined.

This thesis will use Γ to denote the set of action definitions in a PDDL domain and refer to Γ as the *context* of a PDDL domain.

2.3.2 PDDL Problem Definition

A PDDL problem is defined by giving definitions of the objects, initial world, goal and domain of a problem. The domain is simply defined by giving the domain name e.g. (: `domain blocksworld`).

Objects define the set of constants that can be used for a problem. In Blocks World the objects are the blocks on the table. The object definition is given by a list of names. In domains that use typing the objects are also given types.

Example 16 (Typed Objects)**.**

The objects in Figure 2.2 can be given the `block` type with the definition (: `objects a b c - block`).

Since Blocks World only has one type of object there is no need to use a typed domain.

The **initial world** gives the initial state of the problem. This is defined by giving a list of all true predicates. All predicates not contained in the initial state are assumed to be false because PDDL operates under a *closed world* assumption. The initial world should be considered a set of predicates as the order of the predicates is irrelevant to the planning problem. All of the predicates in the initial world have to be *ground*. This means that all of the free variables in the predicates have to be substituted by objects.

Example 17 (Initial World)**.**

Given the initial world (: `init` (`onTable a`) (`onTable b`)), the predicate (`onTable a`) is true but the predicate (`onTable c`) is false because it is not contained in the world. The predicate (`onTable ?x`) would not be valid to include in the initial world because it contains the free variable `?x`.

A **goal** defines a logical expression that must be *satisfied* for a plan to solve the planning problem. In this thesis a goal is represented by a ground PDDL formula formally defined in Definition 12. The formal definition for satisfaction is defined in Definition 13.

2.3.3 Declarative Semantics for PDDL

This section will introduce declarative semantics for the entailment of ground PDDL formulae which will be used to formally define the notion of satisfaction. This thesis will give declarative semantics and define satisfaction over worlds rather than states. In this thesis *worlds* are sets of ground atomic formulae denoting all true formulae.

$$\text{Both}\ \frac{w \models_z F \quad w \models_z F_1}{w \models_z F \wedge F_1} \qquad \text{Flip}\ \frac{w \models_{-z} A^g}{w \models_z \neg A^g}$$

$$\text{Somewhere}\ \frac{A^g \in w}{w \models_+ A^g} \qquad \text{Nowhere}\ \frac{A^g \notin w}{w \models_- A^g}$$

Figure 2.3: *Declarative interpretation of PDDL formulae.* A^g *represents a ground atom.* $-z$ *is defined by taking* $-+ = -$ *and* $-- = +$.

All formulae that are not in a world are considered false. Worlds are all subsets of Herbrand base. Since there are no function symbols and there is a finite set of predicates and constants there is a finite number of worlds. Whilst a world in PDDL is commonly referred to as a state, this thesis will refer to the state describing a world in PDDL as a world. This is done to ensure that the reader knows when this thesis is referring to states containing only true predicates.

The declarative semantics of ground PDDL formulae are given by an *entailment relation* defined in Figure 2.3. The letter w is used to denote a world. PDDL formulae can be used to represent the preconditions and effects of actions as well as goal formulae.

Example 18. Given the syntax of Definition 12, `handEmpty`$\wedge\neg$`onTable a` is a formula.

In order to solve a planning problem, a planner needs to show that a given goal is satisfied by the final world produced from the application of a plan. In this thesis, if a PDDL formula can be entailed from a world then it is *satisfied* by that world therefore Goal satisfaction is equivalent to entailment.

Definition 13 (Satisfaction)**.** Given a world w, a goal formula F *is satisfied by* w if $w \models_+ F$ can be derived by the rules of Figure 2.3.

Example 19 (Entailment Example)**.**

Given the running example if there existed a world $w_1 = \{$`handEmpty`$\}$ then the only atomic formula that can be positively entailed is `handEmpty` because `handEmpty` $\in w_1$. It would be possible to positively entail any other negated formula in the problem such as $\neg$`onTable a` because `onTable a` $\notin w_1$ e.g. $w_1 \models_+ handEmpty \wedge \neg onTable\ a$.

By using set membership the entailment relation preserves the closed world assumption of PDDL as it assumes that all true formulae exist in the world. It should

also be noted that the order of the formula contained in the goal formula does not affect entailment.

One notable restriction of the PDDL formulae used in this thesis is that only atomic formulae can be negated. This is done to ensure that disjunction is not definable in the systems presented in this thesis. For example, if negation could be applied to arbitrary formulae then it would be possible to define disjunction with the formula $\neg(\neg F \wedge \neg F_1)$. This restriction prevents disjunctive goals as well as disjunctive preconditions and effects in actions. In this thesis, goal formulae are PDDL formulae where all atomic formulae are ground. The goal formulae need to be ground to ensure the decidability of set membership on worlds.

2.3.4 Solving a PDDL Planning Problem

Given a PDDL domain and problem a planner has to find a plan that will reach a state that satisfies the goal state for a problem after being applied to the initial world. This section will give definitions in the context of PDDL for satisfaction, plans and plan application.

PDDL is a language that represents domains and problems so does not supply a representation for plans [37] therefore it is up to plan validators and planners to define their own representation. This thesis will define plans with respect to STRIPS definitions of actions that are parametrised therefore the definitions of plans have to be updated in comparison to Definitions 8, 9 and 10.

In PDDL all action definitions contain definitions of the action name as well as its preconditions and effects. To make it easier to define definitions for plans, this thesis will introduce a notion of context that contains the precondition and effect definitions for all actions. An action $\alpha[\sigma]$ will therefore simply refer to the action name followed by its ground substitution. This can be noticed from the difference between the definitions of plans given in Definition 8 and Definition 16.

Definition 14 (First-Order Actions)**.**

Definition of PDDL actions given the constant and variable definitions in Figure 2.1

$$\text{Term} \quad \mathit{Term} \ni t, t_1, \ldots t_n \quad ::= x \mid c$$

$$\text{Actions} \quad \mathit{Act} \ni a \quad ::= \alpha\ (t_1, \ldots t_n)$$

The context will contain the definitions of preconditions and effects for all actions.

Definition 15 (PDDL Context)**.**

A PDDL context is a function mapping all actions to preconditions and effects $\gamma : \alpha(\overline{x}) \rightarrow F(\overline{x}) \times F_1(\overline{x})$. A PDDL context is similar to the state transition function given in Definition 7 and is part of the definition of a PDDL domain Σ along with definitions of predicates and types. The preconditions and effects for a given action can be accessed by the following functions:

$$\mathit{Pre}(\gamma(\alpha[\sigma])) = F[\sigma]$$

$$\mathit{Post}(\gamma(\alpha[\sigma])) = F_1[\sigma]$$

Definition 16 (PDDL Plan)**.**

A plan is a finite sequence of action names with ground substitutions:

$$\pi = \alpha_1[\sigma_1]; \alpha_2[\sigma_2], \ldots; \alpha_n[\sigma_n]$$

An *empty PDDL plan* containing no actions is represented by $[]$.

In what follows, this thesis will only use PDDL plans and will often just call them plans. Plan applicability in STRIPS is defined based on a satisfaction relation where an action can only be applied to a state if its preconditions are satisfied by that state. In this thesis both the preconditions of actions and the goal formula use the satisfaction relation given in Definition 13.

Example 20 (Goal Satisfaction)**.**

Assume that a plan reaches the final world:

$$\{(\texttt{handEmpty})\ (\texttt{onTable c})\ (\texttt{on a b})\ (\texttt{on b c})\}$$

The PDDL goal (: `goal` (`and` (`on a b`) (`on b c`))) can be written as a formula (`on a b`) $\wedge$ (`on b c`) that would be satisfied since both predicates are contained in the world. The goal formula (: `goal` (`and` (`on a b`) (`not` (`handEmpty`)))) would not be satisfied because the predicate `handEmpty` is in the world when it is required to not be in the world.

Definition 17 (PDDL Plan Application)**.**

This definition assumes the existence of a function $f(\gamma, \alpha[\sigma], w) \rightarrow w_1$ that applies a grounded action to a world given its context.

A plan $\pi = \alpha_1[\sigma_1]; \alpha_2[\sigma_2]; ...; \alpha_n[\sigma_n]$ is *applicable* in a world state w_0 if there are worlds $w_1, ..., w_n$ such that $f(\gamma, \alpha_i, w_{i-1}) = w_i$ and $w_{i-1} \models_+ Pre(\gamma(\alpha_i[\sigma]))$ for $i = 1, ..., n$. The result of plan application is the final world w_n that is produced by $f(\gamma, \alpha_n, w_{n-1})$. The result of plan application is denoted as:

$$[\![f, \gamma, \pi, w_0]\!] = w_n$$

The empty plan [] is applicable in all worlds w and results in the same world w.

Definition 18 (PDDL Planning Problem Solution)**.**

Given a PDDL domain Σ, an initial world w_0 and a goal formulae G then a solution to the PDDL planning problem (Σ, w_0, G), is any plan $\pi = a_1[\sigma_1]; ...; a_n[\sigma_n]$ such that $[\![f, \gamma, \pi, w_0]\!] \models_+ G$.

This thesis will take a STRIPS view on action application [32] therefore the function f in Definition 17 will apply an action to a world by adding all atomic predicates in the effects to the world and removing all negated predicates in the effects from the world. A suitable function is given in Definition 26.

Example 21 (Plan Application)**.**

One solution to the running example described in Figure 2.2 could be the plan:
`pickup_from_table b`;
`putdown_on_stack b c`; `pickup_from_table a`; `putdown_on_stack a b`.

This thesis will say that a plan is applicable to a world if its preconditions are satisfied for all actions when being applied to a world. A plan will solve a PDDL

planning problem if it is applicable to the initial world given the domain and results in a world that satisfies the goal condition.

Example 22 (Action Application)**.**

The application of the action `pickup_from_table a` to the initial world given in Figure 2.2 results in the world: {(`onTable b`) (`onTable c`) (`clear a`) (`clear b`) (`clear c`) (`holding a`)}.

2.4 Curry-Howard Correspondence

This thesis will design and implement a logic for AI planning in the dependently typed programming language Agda. This is possible through the Curry-Howard correspondence [118, 122]. The Curry-Howard correspondence shows that propositions in intuitionistic logic correspond to types in programming languages. Specifically, this correspondence is very closely related to the standard BHK-interpretation of intuitionistic logic [120]. This correspondence was developed over many years, starting with Curry [24] who observed that an implication $F \rightarrow F_1$ is equivalent to a function $F \rightarrow F_1$. The theory was further developed by Howard [56] who extended the correspondence to the other logical connectives $\wedge, \vee$. More importantly, Howard observed the correspondence between the simplification of proofs and the reduction of programs. This observation proved that the correspondence between propositions and types is deep and inspired Martin-Löf's type theory [73] which paved the way for the development of programming languages that fully embody the correspondence.

The Curry-Howard correspondence shows a correspondence between a given logic and a programming language [122]:

There are three aspects to this correspondence:

1. (*Propositions as Types*) For each proposition in the logic there is a corresponding type in the programming language.
2. (*Proofs as Programs*) For each proof of a given proposition, there is a program of the corresponding type.
3. (*Simplification of proofs as reduction of programs*) For each way to simplify a proof there is a corresponding way to reduce a program.

To illustrate that first-order logic propositions can be represented as types in a

programming language consider the logical connectives $\wedge, \vee, \rightarrow$ of first-order logic. A conjunction $F \wedge F_1$ states that both F and F_1 hold. A disjunction $F \vee F_1$ states that either F or F_1 holds. An implication $F \rightarrow F_1$ states that if F holds then F_1 holds. These logical connectives all directly correspond to types in programming languages. This correspondence is particularly clear when considering the intuitionistic interpretation of these connectives.

Intuitionistic logic is based on the notion of construction where a proof of a proposition is a construction of the proposition.

Example 23 (Proof of Construction)**.**

An even number could be defined with the following definition $even(x) = \exists y : \mathbb{N}.y + y = n$. This states that x is even if there is a object of the natural number type y that when added to itself equals x. To prove the proposition $even(8) \rightarrow \top$, which states that the number 8 is even, a construction of a natural number y has to be given as well as a construction of a proof that $y + y = 8$. In this case, the number 4 could be used to prove this proposition.

Example 24 (Conjunction and Cartesian Product)**.**

To prove a conjunction $F \wedge F_1$ a proof of construction for both F and F_1 have to be given. This corresponds to the Cartesian product $F \times F_1$ where a value of type $F \times F_1$ is given by a value of type F and a value of type F_1.

Example 25 (Disjunction and Disjoint Sum)**.**

To prove a disjunction $F \vee F_1$ then a proof of construction has to be given for either F or F_1. This corresponds to the disjoint sum $F + F_1$ where a value of type $F + F_1$ is given by a value of type F or a value of type F_1.

Example 26 (Implication and Functions)**.**

A proof of an implication $F \rightarrow F_1$ is a method for transforming every construction of F into a construction of F_1. This corresponds to a function type $F \rightarrow F_1$ where a function of type $F \rightarrow F_1$ will produce a value of type F_1 when applied to a value of type F.

The above examples have illustrated correspondence between first-order logic connectives and programming languages for intuitionistic logic. A key difference

between intuitionistic logic and classical logic is that the law of excluded middle that states the proposition $F \vee \neg F$ holds for all F does not hold in intuitionistic logic but holds in classical logic. This is because in intuitionistic logic to prove this proposition there needs to be a proof of construction of either F or $\neg F$ for any F and it is impossible to know for any F which side of the disjunction holds.

In first-order logic an entailment relation $T \vdash p : A$ is used to show that a proposition A holds in a first-order theory T given the proof p. The Curry-Howard correspondence states that it is possible to represent this entailment in a programming language. In this case, first-order formulae are seen as *types*, and proofs are seen as terms that inhabit those types. If implemented in a programming language, the entailment relation states that the *proof term* p is an *executable program* that satisfies the property A given the theory T, and moreover that this inference is sound, i.e. that $T \vdash p : A$ holds in some formal system. One advantage of this approach is that the implementation of this entailment in the formal system will give the additional guarantees of that system. For example, implementing a theory in Agda guarantees that all functions will terminate.

2.5 Agda Introduction

Inspired by the Curry-Howard correspondence Martin-Löf introduced an intuitionistic theory of types [73]. This theory gave a formalism for constructive mathematics as well as introduced the notion of dependent types which are types that can contain arbitrary values and appear as arguments to the results of ordinary functions. Martin-Löf furthered his intuitionistic theory of types by showing how it can be viewed as a programming language [74]. This work led to the development of Agda [87] which is a dependently-typed programming language based on Martin-Löf's intuitionistic type theory. Thanks to the Curry-Howard correspondence Agda is not only a functional programming language that can compile executable programs, it is also an interactive theorem prover that can prove propositions written in intuitionistic logic. Agda is not the only programming language that is considered an interactive theorem prover. One notable alternative is the interactive theorem prover Coq which is based on the Calculus of Inductive and Coinductive Construc-

tion (CIC) [23]. Just like Agda, Coq allows for the compilation of programs as well as the ability to prove propositions.

One key difference between the two languages is that proofs in Coq have their own tactic language whereas proofs and functions in Agda are all written in the same language. Note that it is possible in Coq to write proofs directly in the programming language and tactics exist on top of this. However, writing proofs directly in CIC is cumbersome in Coq as it does not provide dependent pattern matching in a convenient way. Agda has dependent pattern matching, which is why it is feasible to use the programming language directly for proofs.

The use of interactive theorem provers has been very successful in many verification projects. For example Coq was used in CompCert [67] to create a verified C compiler. Agda has also been used to create executable formalisations such as the formalisation of system F [20] in Agda.

2.5.1 Agda Basics

This section will give an overview of the basics of using Agda. Agda has an expressive type system with the ability to express arbitrary data types. To define a datatype a developer has to give a data declaration that includes the name and type of the datatype as well as its constructors and their types. It is possible to define the natural numbers with the following data declaration:

Example 27 (Natural Numbers in Agda)**.**

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

The natural numbers are an inductive definition in Agda where a natural number is either zero or some successor of a previous natural number. In Agda a colon is used to represent a type in the form name : Type.

Example 28 (Natural Number)**.**

The declaration x : ℕ in Agda states that x is a natural number. The definition of x therefore has to be a construction of a natural number. The following code defines x as the natural number 2 by applying the successor function twice to zero.

```
x : ℕ
x = suc (suc zero)
```

Addition over natural numbers can be defined by a recursive function in Agda.

Example 29 (Addition)**.**

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
suc m + n = suc (m + n)
```

The addition function makes use of pattern matching over the natural number datatype. In Agda all functions are guaranteed to be total. This includes two obligations:

1. pattern matching has to cover all possible cases,
2. functions have to terminate.

In the case of the addition function there are only two cases for a natural number which is that the number is zero or that it is a successor of some other number. Termination is guaranteed if recursive calls have to be getting smaller. For example, if the recursive call in the addition function was (suc m + n) then the function would not type check. Another concept illustrated by the addition function is Agda's support for definitions of *mixfix* operators where underscores _ are used to denote where the arguments are expected to go.

So far this section has introduced functions as well as data declarations. Due to the rich type system of Agda it is also possible to represent propositions as types.

Example 30 (Less Than or Equal Proposition)**.**

In Agda the "less than or equal" operator for natural numbers is defined as:

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n
```

This is an inductive operator where the base case states that zero is less than or equal to all natural numbers. The inductive case states that if a number is less than

or equal to another number then the relation still holds for the successor of both sides.

Given two natural numbers m and n, a value of the type $m \leq n$ is a proof that the natural number m is less than or equal to n. For example, a proof that the number two is less than or equal to three can be given in Agda by:

```
p : 2 ≤ 3
p = s≤s (s≤s z≤n)
```

A developer knows that the above proof is a correct proof that 2 is less than or equal to 3 because it type-checks in Agda. Another feature of Agda is that it has an automatic proof search that will try to construct a value of the goal type. The proof of p in the above example was automatically generated by Agda.

All the examples shown so far can be represented as a record type. Record types can be used to group values together in a single type. Records do not need to have values of the same type. A record is defined using the record declaration along with the field names and the types of the fields.

Example 31 (Records)**.**

```
record leqNums : Set where
  field small : ℕ
        big : ℕ
        prf : small ≤ big
```

The leqNums record has three fields. small and big fields are natural numbers and the prf field is a proof that the number given in the small field is less than or equal to the number given in the big field. It is possible to construct a value of the leqNums by defining values for all fields as shown below.

```
ln : leqNums
ln =
  record {
    small = 2 ;
    big = 3 ;
    prf = s≤s (s≤s z≤n) }
```

Records also have inbuilt projection functions for easy access to values in their fields. For example, the small value can be easily accessed by leqNums.small.

One of the key differences in intuitionistic logic is the interpretation of negation. In classical logic the statement $\neg F$ states that the proposition F is false however in intuitionistic logic it states that it is impossible to construct a proof that F is true.

Example 32 (Negation in Agda)**.**

The proof that it is impossible to show three is less than or equal to two is represented in Agda by:

```
p : ¬ (3 ≤ 2)
p (s≤s (s≤s ()))
```

To show that it is impossible in Agda it is first assumed that there is a construction of the proof $3 \leq 2$ which is inductively case split on until the case $1 \leq 0$ is reached. At this point, it is impossible to continue the splitting and the proof of negation is finished. The notion that it is impossible to continue is represented in Agda by ().

As discussed in Section 2.4, the intuitionistic view of disjunction perfectly maps to the sum type in Agda. Disjunction is represented by the symbol ⊎ in Agda and a proof is given by choosing which side of the disjunction holds, using inj_1 for the left side and inj_2 for the right side, along with a construction of a value of the chosen side's type.

Example 33 (Sum Type)**.**

This example states that either 2 is less than or equal to 3 or that it is impossible to construct a proof that 2 is less than or equal to 3. Since 2 is less than or equal to 3 the left hand side is chosen and a proof is given.

```
p : 2 ≤ 3 ⊎ ¬ (2 ≤ 3)
p = inj₁ (s≤s (s≤s z≤n))
```

Agda is a language that has a module system that handles the way names are used in a program. This works in similar ways to other module systems where

modules can be imported and definitions from modules can be used, hidden and renamed. Modules can also be parametrised which this thesis will use to give define abstract implementations.

Example 34.

The below example first imports a module containing definitions of natural numbers whilst hiding the definition of addition from the module. An example module is then declared that is parametrised by a natural number.

```
open import Data.Nat hiding (_+_)

module Example (num : ℕ) where
```

2.5.2 Dependent Types

One of the key advantages of Agda is that it is a dependently typed programming language. In standard programming languages, there is a clear separation between types and values. In dependently typed languages, types can contain arbitrary values and appear as arguments and results of ordinary functions. This means that types can not only contain values but also that types can contain propositions about these values.

This thesis will introduce a concept of fuel where an action can be applied depending on the level of fuel. The fuel data type is indexed by a natural number which allows for reasoning about the value of the fuel at the type level.

Definition 19 (Fuel)**.**

```
data Fuel : ℕ → Set where
  fuel : (n : ℕ) → Fuel n
```

Example 35 (Updated Decrement Fuel)**.**

Imagine that a decrement function of type $\mathbb{N} \to \mathbb{N}$ that decrements a natural number. One problem with this decrement function is that type does not ensure the behaviour of the function as any natural number could be returned. It is also possible to try to decrement the number zero which cannot be decremented. With

the fuel type it is possible to ensure that both of these issues can be addressed at the type level. This can be done by the following definition:

```
decrementFuel : ∀ {n} → Fuel (suc n) → Fuel n
decrementFuel (fuel (suc n)) = fuel n
```

The decrementFuel function encodes at the type level that the fuel level will go from suc n to n where suc n is the successor of the natural number n.

The decrementFuel function utilises an implicit type denoted by $\{n\}$. Implicit types are types that Agda's type checker attempts to figure out and do not need to be given as arguments to functions. It is obvious what the value of n should be since when the function is run it is given a value of suc n therefore it does not need to be explicitly passed into the function.

Giving more detail at the type level of functions is really helpful in Agda due to its interactive nature. At any point in development a user can leave a hole in the function and query what the goal is. As discussed, Agda has an automated proof search that tries to construct a value that corresponds to the goal type. The more precise this type is the more likely Agda is to construct something that corresponds to the desired definition. The decrementFuel function can only type check if it returns a construction of fuel with the number decremented however the decrement function described would still type check no matter what number was returned.

It is also possible to encode properties as arguments at the type level of functions. This is helpful as a proof of a type level property has to be given in order to execute a function that contains one.

Example 36 (Minus)**.**

One issue with defining a minus function on natural numbers is what to do in the case $m - n$ when n is greater than m. One option is to simply return zero. In Agda it is possible to ensure that this case never happens by including the proof $n \leq m$ in the argument of the function. This prevents the application of the function in the case that $n > m$.

```
minus : (m : ℕ) → (n : ℕ) → n ≤ m → ℕ
minus m zero n≤m = m
minus (suc m) (suc n) (s≤s n≤m) = minus m n n≤m
```

When defining typing relations such as _≤_ it is useful to be able to define procedures that that will automatically calculate whether the relation holds. Typically, programming languages will define a function that will return a boolean value for whether two numbers are equal. The problem with this approach is that the evidence for why the relation holds is lost as the function only returns the boolean value. Agda allows for the definition of the *decidable* data type that defines a procedure for deciding a property that gives a yes or no value just like a boolean as well as proof evidence for the choice.

Definition 20 (Decidable [123])**.**

```
data Dec (A : Set) : Set where
  yes : A → Dec A
  no : ¬ A → Dec A
```

Example 37 (The relation _≤_ is decidable)**.**

```
_≤?_ : (m n : ℕ) → Dec (m ≤ n)
zero ≤? n = yes z≤n
suc m ≤? zero = no (λ ())
suc m ≤? suc n with m ≤? n
... | yes m≤n = yes (s≤s m≤n)
... | no ¬m≤n = no (λ { (s≤s m≤n) → ¬m≤n m≤n})
```

2.6 Summary

This chapter has introduced the variety of preliminaries required to understand the rest of this thesis. On the planning side, this included introductions to the subset of AI planning and PDDL that are used in this thesis. On the programming language side, this included an introduction to the Curry-Howard correspondence and the Agda programming language. This chapter has also defined all topics using a standard notation that will be used throughout this thesis.

Chapter 3

Literature Review

This thesis is inter-disciplinary, and this chapter clarifies the conceptual relation of different methods used in this thesis, relative to the existing literature. This chapter will separately discuss the literature surrounding programming languages and AI planning and then finally discuss their relations.

3.1 Programming Languages

3.1.1 Type-Based Verification

The Curry-Howard correspondence [118, 122] is at the core of this thesis as this thesis not only utilises many research results deriving from the correspondence but also takes inspiration from the correspondence itself when designing its inference systems. A detailed description of the correspondence is given in Section 2.4.

One of the key research results derived from the Curry-Howard correspondence is the creation of interactive theorem provers. The Curry-Howard correspondence shows that propositions in intuitionistic logic correspond to types in programming languages. As discussed in Section 2.5, the Curry-Howard correspondence inspired Martin-Löf to create a theory of types [73] that can be viewed as a programming language [74]. This work not only led to the creation of Agda [87] but also inspired the creation of the calculus of constructions [23] on which the interactive theorem prover, Coq, is based. Interactive theorem provers such as Agda and Coq are programming languages that have broad uses within verification. This is because they allow for the definition of propositions as well as their proofs within the language.

These proofs have high-levels of assurance as they are checked by the language's type checker.

The creation of interactive theorem provers enables typed-based approaches to programming validation which is a field of research focusing on utilising type systems to prove properties of programs. One of the key methodologies is the idea of Proof-Carrying Code [86, 128] which is the idea of splitting program verification into certification and proof validation stages. At the certification stage, a proof is generated that acts as a witness that the certification was performed. This proof is *carried* with the code where it can be checked during runtime. Since proof validation is far quicker than certification, the Proof-Carrying Code methodology allows for the running of certified code with minimal overhead.

There have been many high-profile successes utilising type-based verification such as CompCert [67] where the Coq proof assistant was used to create a verified optimizing C compiler. This was a massive verification effort that aimed at providing optimizations to C code whilst maintaining high levels of assurance. This included formal proofs that CompCert is free from miscompilation issues. Coq has also been used to prove many mathematical theorems such as the Four-colour theorem [42]. The Four-colour theorem is significant because it was the first theorem proven using a computer program. This theorem was heavily doubted due to it being proven using assembly language programs however this scepticism was alleviated when it was finally proven in Coq. Other examples of famous proofs using the Coq interactive theorem prover include the Kepler conjecture and the Feit-Thompson theorem [41, 44].

Agda has also been used for many formalisations. In recent years, research into cryptocurrencies has led to many executable formalisations being defined in Agda. For example, System F [20] has been formalised in Agda due to it corresponding to the underlying Plutus Core language in the Cardano blockchain /citebrunjes2020utxo. This formalisation can extract a Plutus Core interpreter that can be tested against the production Plutus Core interpreter. Similar research has also been done formalising the UTXO model used by Bitcoin [115] and the Extender UTXO model used by Cardano [19].

As software systems scale there is a general need to produce formal specifications

and proofs about software. Type-based verification is an ideal strategy to accomplish this as interactive theorem provers such as Agda and Coq have many of the tools necessary to accomplish this [7]. This is particularly true when formalising programs that are executable as these languages have the ability to not only prove properties about programs but also compile and execute them. The idea of proof-carrying code and the use of interactive theorem provers is a core part of this thesis. This is because this thesis will not only utilise Agda to verify plans but also to compile and execute them.

3.1.2 Semantics

The study of semantics for computer programs and programming languages is also closely related to this thesis. This is because in order to formalise programs some form of semantics must be given. Key foundational work in this field was done by Hoare and Floyd [34, 54] which focused on providing logical foundations for describing the meaning of computer programs using an axiomatic approach. This work allows for formal proofs to be written about computer programs such as proofs of correctness and termination. Hoare logic is the canonical example of axiomatic semantics for describing computer programs.

There are many approaches to describing the semantics of computer programs such as operational semantics [100] and denotational semantics [113]. Operational semantics focuses on the operations that a computer program can perform. These operations can be seen as describing transition steps from one configuration to another. Typically, operational semantics are given either by structural operational (small-step) semantics [100] or natural (big-step) semantics [60]. Structural operational semantics describe the transitions between configurations as individual steps of computation whereas natural semantics describe the entire transition from a configuration to a final value. Denotational semantics describe computer programs as denotations typically representing mathematical objects.

This thesis utilises various formal semantics to describe AI plan validation. This thesis takes particular inspiration from Hoare triples $\{P\} \rightsquigarrow \{Q\} \mid c$ that state that from a state P, the execution of a computer program c results in a state $\{Q\}$. This thesis adapts Hoare Triples to a planning context by using the triple $\{P\} \rightsquigarrow \{Q\} \mid f$

to state that from an initial state P, the application of a plan f results in a state satisfying the goal state Q. The inference rules given in Chapter 4 can be seen as describing the individual computational steps taken to reach the goal state from the initial state, therefore this thesis gives structural operational semantics for plans.

3.1.3 Resource Logic

This thesis takes inspiration from resource logics [105]. One popular area of research within the field is separation logic [89, 109] which is an extension of Hoare logic that was created for verifying and reasoning about imperative programs with shared mutable data structures. Separation logic has been used extensively in program verification [11, 12, 83, 92, 102]. This thesis takes inspiration from two papers written by Berdine et al. [11, 12]. In the first paper [12], a symbolic execution system was described that provided operational rules for programs as well as proof rules for entailment. In the second paper [11], the system presented in the first paper [12] was mechanised. This thesis follows this style of approach where inference rules are first defined and then later mechanised.

There is also work within the field of separation logic that utilises the Curry-Howard correspondence. Both Hoare logic and Separation logic have been given a Curry-Howard interpretation [81, 83]. Several papers explore the computational and practical benefits of this interpretation. For example, Polikarpova and Sergey [102] took a Curry-Howard approach to Separation logic to improve program synthesis by framing it as a proof search problem. In a similar way to the specifications defined in this thesis, they define a synthesis goal $\Gamma \vdash P \rightsquigarrow Q$, which is solved by a program c if the assertion $\Gamma \vdash P \rightsquigarrow Q \mid c$ can be derived in their system. This is particularly interesting with respect to AI planning as a planning problem can be seen as a program synthesis problem. In Chapter 5 this thesis introduces PCPR logic which is a Curry-Howard approach to AI planning that is in part inspired by existing applications of the Curry-Howard approach in separation logic.

3.1.4 Automated Theorem Proving

Within the programming language field there is also a large amount of research into automated theorem proving. This includes stand-alone automated theorem

provers as well as various tools designed to automatically generate a proof within an interactive theorem prover.

Satisfiability (SAT) solvers [40] and SMT solvers [10] are one example of automated theorem proving that is used in the programming language and AI planning fields. A SAT solver is a computer program designed to solve the boolean satisfiability problem. This problem is: given some boolean statement, try to find an example in which the statement will evaluate as true. One of the restrictions of SAT solvers is that all problems are restricted to boolean theories. Many problems require first-order reasoning which led to the creation of SMT solvers that generalise SAT problems by adding first-order background theories such as equality, lists, integers and real numbers. By adding these background theories an SMT solver is able to do bounded checks for satisfiability over infinite domains.

SMT solvers are used in functional programming languages. The Z3 solver [79] was used to add refinement types to Haskell [121]. The addition of refinement types allows for the encoding of invariants over Haskell functions that are ensured by SMT solvers. For example, it is possible to encode the invariant over the minus function as described in Example 36. SMT also has uses in interactive theorem provers. SMTCoq [30] is a plug-in for Coq that dispatches Coq goals to external SAT and SMT solvers and then translates and verifies the certificates provided by the solvers in Coq. This allows for the use of SAT and SMT solvers in Coq proofs which can vastly speed up compatible proof problems. Recently there has also been work on allowing Agda to interface with SMT solvers [66].

There are also many automated theorem proving methods that are created directly for use with interactive theorem provers such as Hammers [13] which have found great success in automatically proving theorems in interactive theorem provers based on higher order logic such as Isabelle. This thesis does not utilise any external theorem provers to solve proofs in Agda. On the other hand, in Chapter 5 this thesis utilises tactics that are enabled by reflection in Agda [125] to automatically generate equality proofs when translating PDDL domains and problems. Chapter 5 also automatically generates proofs for plans using the inference systems defined in this thesis.

3.2 AI Planning

3.2.1 Logic for AI Planning

Just as there are many different semantics for representing programming languages, there are also many different logics created to represent AI planning problems. This section will discuss some of the approaches taken to represent AI planning, specifically those deriving from the situation calculus and STRIPS.

In 1963 McCarthy published a paper [75] describing the situation calculus that is a formalism designed to represent AI planning problems. The core components of situation calculus are: situations, actions and fluents. Actions are used to change the state of the world. Fluents describe the world state and can be given by boolean relations or functional relations containing a range of values. In McCarthy's situation calculus a situation describes "the complete state of the universe at an instant of time". As discussed in Section 2.2 it is obviously impossible to describe the complete state of the universe therefore the frame problem needs to be addressed.

The frame problem specifically is concerned with representing the non-effects of actions. For example, a person would intuitively understand that picking up some block a that is on the table would have no effect on some other block b that is on the table. In McCathy's situation calculus the situation resulting from performing the action of picking up block a would no longer contain the information that block b is on the table. McCarthy initially proposed the introduction of a frame axiom [76] to deal with this problem however this approach was shown to not work [45, 70].

To address the frame problem in situation calculus, in the 1990s, Reiter introduced a new formalisation of situation calculus [107, 108] that describes situations as finite sequences of actions performed on the initial situation. In this formalisation of the situation, the effects of actions are described using *successor state axioms* that describe all possible changes to fluents given a situation. Any fluent that is not affected by performing the action on a situation remains unchanged in the new situation. Reiter's formalisation of the situation calculus is considered the standard.

The Stanford Research Institute Problem Solver (STRIPS) [32] is a theorem prover for AI planning problems created in 1971. As discussed in Section 2.2, STRIPS addressed the frame problem by defining the effects of actions as additions

and deletions from a state. In this case, all predicates in a state that are unaffected by the action remain the same as in situation calculus. Whilst STRIPS had its own definitions for the planning problem, it was ultimately a theorem prover, not a calculus or logic therefore further work was done in 1987 defining formal semantics for STRIPS [69].

Situation calculus is a first-order logic where situations themselves are first-order objects that can be quantified over. This makes situation calculus more expressive than STRIPS, in contrast, planning in situation calculus is much more computationally expensive than in STRIPS. This led to the creation of a language somewhere in between STRIPS and situation calculus called the Action Description Language (ADL) [95, 96]. Some key differences between STRIPS and ADL are:

- ADL can contain positive and negative literals whereas STRIPS can only contain positive literals,
- ADL allows for quantification within the goal state whereas STRIPS can only contain grounded literals,
- ADL allows for conditional effects.

The work on ADL inspired the creation of the Planning Domain Definition Language [37] discussed in Section 2.3. PDDL is based on the STRIPS formalism of AI Planning but contains many of the extensions that were proposed in ADL including negative preconditions, typing and conditional effects.

This thesis can be seen as part of this larger work of defining logic to represent planning problems. It contains definitions of formal semantics for AI planning, two approaches to deal with the frame problem as well as three inference systems for the validation of AI plans. All of the work presented in this thesis is inspired by the work given above, specifically the STRIPS and PDDL approach.

3.2.2 PDDL Versions

PDDL is a very expressive language with many extensions. PDDL 1.2 usually operates under a closed world assumption and expresses domains using the STRIPS assumption where the effects of an action are applied by adding and deleting predicates to a given world.

PDDL 1.2 also allows for the expression of types with type hierarchy, equalities

over objects, existential and universal quantification over preconditions and conditional effects. Conditional effects are effects that will only be applied when a list of preconditions hold true. In PDDL 2.1 there is also a definition of *numeric fluents* that allow for the reasoning about numbers such as comparing and adding numbers. PDDL 2.1 also introduces negative preconditions and durative actions. Durative actions add the concept of time to actions. Finally, PDDL 3 adds strong and soft constraints that can be applied across a planning problem. Strong constraints can allow for the statement of certain implications to hold across every state during the execution of a plan. Soft constraints, also known as preferences, introduce soft goals that a user would prefer a planner to satisfy but are not necessary to satisfy for a valid plan. Many PDDL extensions such as conditionals introduce branching into planning. Whilst loops are not officially supported there has been work done to encode loops by using actions [99]. There also exist many unofficial extensions and variants to PDDL such as PDDL+ [36] that further extend the expressivity of PDDL.

3.2.3 Embedding Planning in Other Logics

This thesis embeds the AI planning problem into the dependently-typed programming language Agda. This is inspired by previous work done in embedding planning into logic programming languages and Linear Logic.

There is a long history of modelling AI planning in Linear logic, that dates back to the 90s [59], and was investigated in detail in the 2000s, see e.g. [21, 119]. In fact, AI planning is used as one of the iconic use-cases of Linear logic [101]. The main idea behind using Linear logic for AI planning is treating action descriptions as linear implications:

$$\alpha : \forall x.P \multimap Q,$$

where P and Q are given by tensor products of atoms: $R_1(t_1) \otimes \ldots \otimes R_n(t_n)$. It is possible to incorporate information about polarities inside the predicates, as follows: $R_1(t_1, z_1) \otimes \ldots \otimes R_n(t_n, z_n)$. Then, the linear implication and the tensor products model the resource semantics of PDDL rather elegantly. Curry-Howard semantics of Linear logic also attracted attention of logicians first in the 90s [6], and then in

the 2000s in connection with research into Linear Logical Frameworks [18, 112].

In 1997 GOLOG [68] was created which is a logic programming language for AI planning based on Reiter's situation calculus [108]. This allows for the reasoning of AI planning problems in situation calculus with an explicit representation of the world. GOLOG can be used as a tool to not only reason about and check plans but also to search for plans. An issue with searching for plans in GOLOG is that it is far slower than searching for equivalent plans using ADL planners. This is because GOLOG is primarily designed as a programming language so does not have the optimisations that dedicated planners have. For example, planning can be simply implemented in GOLOG by asking its solver to nondeterministically choose a sequence of actions until a goal is met. To improve the speed of planning in GOLOG work was done to integrate GOLOG with ADL planners [22]. In order to accomplish this, the paper showed how to convert ADL planning problems into the situation calculus. This was possible because the situation calculus is more expressive than ADL.

This thesis embeds the planning problem into the richer dependently-typed language Agda in order to reason about plans. The work done in Linear Logic and in particular, GOLOG supports this idea of lightweight verification for plans taken by this thesis. This is because this thesis uses a richer language to verify and reason about plans whilst leaving the planning to planners that can find plans more efficiently.

3.2.4 Automated Theorem Provers as Planners

Whilst planners themselves are automated theorem provers, this section will talk about the use of SAT and SMT solvers in AI planning. Some early research in AI planning [63, 64] framed the planning problem as a SAT problem, which allowed for the use of SAT solvers to solve planning problems. Many AI planning problems are first-order, therefore SMT solvers are useful for modern planning. For example, there has been research [14, 17] that encodes the PDDL$^+$ language [36] as an SMT problem which enables SMT solvers to be used to solve PDDL$^+$ planning problems. SMT solvers are also used in planning at large, not just PDDL planning domains and problems [111, 116].

The use of SAT and SMT solvers is one example of the overlap between the AI planning and programming language communities. The work presented in this thesis produces inference systems that automatically verifies plans produced by planners. This can be seen as part of the larger automated theorem proving work done by both communities.

3.2.5 Verification of AI Planning

This thesis uses formal methods to verify AI plans. There are many approaches to AI plan verification including verifying domain models, planning algorithms as well as plans themselves as shown in Figure 1.1.

The verification of domain models [72, 98] seeks to validate whether domain descriptions accurately capture (expert) knowledge about the world. This can be done by performing test based verification of input and output specifications to check the domain performs as expected. Alternatively, some approaches [72] verify common sanity properties that should hold across many domains. One example of this is ensuring that applying an action cannot lead to an inconsistent state where a fact is both true and false at the same time.

Formalisation of planning algorithms [2] has shown that even well-understood algorithms can produce incorrect plans. Modern AI planners are complex software artefacts, and the existing attempts [97, 110] to verify them focus on creating verified algorithms however bugs are often introduced in their implementation [2]. Due to the complexity of planning problems, many planners will opt for implementations where efficiency is the primary concern which further complicates the formal verification of planners. To the author's best knowledge, no mainstream planner has been fully verified yet.

AI plan verification seeks to verify plans produced by planners against some domain model. These tools check properties such as precondition satisfaction, termination and goal satisfaction to ensure that a plan is valid. For example, PDDL has a validator [57] that performs these checks and suggests repairs. In order to accomplish this VAL defined and implemented a semantic interpretation of PDDL in C++. This includes definitions of plan applicability, plan application and satisfaction as given in Section 2.2. In VAL a plan is valid if according to its semantics

a plan is applicable, and the result of the application of the plan results in a state where the goal is satisfied. This more practical and *lightweight* approach to verification is broadly in line with other lightweight verification trends in the literature [33]. However, at the same time, it is rather disjointed from the growing body of research into *type-based verification* [67, 82, 86] or *resource logics* [15, 105] that offer more principled, formal and rigorous approaches, as well as richer languages for expressing the verification properties.

3.3 Programming Languages and AI Verification

3.3.1 Formal Verification in AI

Formal verification techniques have already been applied to the field of AI at large. One particularly popular area of research in this field is the verification of neural networks. One example of this is the use of SMT solvers to verify properties of neural networks [61, 62, 104]. Verification of binarised deep neural networks has also been presented as a SAT problem [84]. There has also been work [9] using Coq that certifies the generalisation of learning procedures that are used in machine learning.

The trend of using formal methods for AI verification can also be seen as part of Explainable AI research [127]. Explainable AI is a research area that aims to make clear what an AI does. Formal methods can be used to elucidate the properties of various AI. For example, the Marabou framework [62] answers users' queries about a given network by translating queries into an SMT problem. This thesis talks about Explainable AI with respect to planning in Section 6.7.

3.3.2 Discussion - Formal Verification in AI Planning

The AI planning and programming languages fields contain vast bodies of research that are largely disjoint but cover many similar topics. Both fields are concerned with the definition of various semantics and logics, verification and automated theorem proving however there has been little work bringing the two fields together. Most of the verification techniques already employed in AI planning sit in between

what this thesis considers to be lightweight verification and testing as illustrated in Figure 1.1. In recent years, there have been multiple papers attempting to bridge the gap between these two fields. In particular, the work done by Abdulaziz in creating a formally verified planner [2] and plan validator [3]. The thesis continues this trend by taking an alternative type-based approach to lightweight plan verification inspired by the Curry-Howard correspondence.

This chapter has laid out evidence to illustrate that the task of integrating the programming language and AI planning domains is an interesting problem and deserves further attention from both communities.

Chapter 4

Planning Problems as Types STRIPS/Operational Approach

4.1 Introduction

This chapter will formalise an AI plan validator in Agda for the STRIPS subset of AI planning problems that can be expressed in PDDL. This is one of the main contributions of this thesis as the formalisation will allow for the validation of PDDL problems with formal guarantees. This will be done by first introducing a new formal language for representing PDDL planning problems. This chapter will then introduce a new logic called Proof-Carrying Plan logic (PCP logic) where the planning problem is defined as an entailment relation. This will be followed by the introduction of a model of computation for plans that describe the execution of a plan on a world. The entailment relation will then be proven sound against the model of computation.

The second contribution of this chapter is a demonstration that AI planning is a natural domain for the Curry-Howard implementation of declarative reasoning. This will be accomplished by defining the entailment relation describing planning problems as a typing relation in Agda. A plan will be defined as an executable function (program) that inhabits the types given by the formulae describing planning problems in the typing relation. By exploiting the Curry-Howard correspondence, type checking not only validates a plan but also automatically ensures that the execution of the plan corresponds precisely to the specification of the planning problem.

4.2 Example: Proof-Carrying PDDL

Examples in this chapter will be given using the *blocks world* domain and problem defined in Figure 2.2. Given the domain and problem definitions, one solution an automated planner may produce is the plan:

```
plan1 = pickup_from_table b; putdown_on_stack b c;
  pickup_from_table a; putdown_on_stack a b.
```

To be able to produce this plan, an automated planner has to have a planning model that internally simulates the execution of the plan. The automated planner will return a plan that according to its planning model is applicable and when applied to the initial state will return a world that satisfies the goal state. Assuming the correctness of the planning model the planner will produce an applicable plan with the guarantee that the result of executing the plan on the initial state will return a world satisfying the goal state. A plan validator can then verify that the plan is applicable and satisfies the goal state according to its internal model about plan execution. For valid plans, a plan validator typically just returns a confirmation that the plan is valid with no reference to the internal computation that the validator has taken to confirm this. The computation that produced the guarantees at both the planner and plan validation stage are implicit and therefore obfuscated from the user. It would be useful to design a formal system where a validated plan has checkable evidence that makes it clear why a specific plan is valid. This will make it transparent why a particular plan is valid. In a formal system, this checkable evidence can also be used to give formal guarantees about the execution of a plan according to its planning model.

This chapter will define a formal system where an executable function plan_1 is generated from the planning domain and problem, such that plan_1 corresponds to the actions of `plan1` and has a type $\mathit{onTable}\ a \ \wedge\ \mathit{onTable}\ b \ \wedge\ \mathit{onTable}\ c \ \wedge\ \mathit{clear}\ a \ \wedge\ \mathit{clear}\ b \ \wedge\ \mathit{clear}\ c \ \wedge\ \mathit{handEmpty} \ \rightsquigarrow\ \mathit{on}\ a\ b \ \wedge\ \mathit{on}\ b\ c$. This example was introduced in Figure 2.2. If this judgement type checks then a verified program plan_1, that can later be compiled and executed, will be obtained. Guarantees about the execution of the plan will be ensured by including a model of execution and proving properties about it in the formal system. This chapter will show that this

task is far from trivial. Although the Curry-Howard correspondence shows that, in principle, (intuitionistic) first-order proofs have a computational meaning, the problem cannot be solved by just formulating arbitrary proofs. Firstly, there is a need to formulate a generic and automatable approach to translate PDDL domains and problems into the dependently-typed setting. In addition, a calculus has to be devised in such a way as to ensure that the programs that inhabit the types give the actual executable plans in the STRIPS sense.

This chapter will first define and prove the formal system for plan validation in mathematical notation independently of the concrete implementation. After this, the formal system will be formalised in Agda.

4.3 Planning Problems as Types

This section will define the formal language for PCP logic that can represent the AI planning problems that can be expressed in the STRIPS subset of PDDL. This language will be an extension to PDDL as it will contain language for talking not only about the syntactic representation of domains and problems but also plans and entailment. Section 2.3.3 defines the declarative semantics for the entailment of PDDL goal formulae. This section will introduce operational semantics that show how to work with the PDDL formulae directly in a type-system, yielding proof obligations that will be fulfilled by plan execution.

4.3.1 Formal Language

Figure 4.1 represents a syntax for representing the STRIPS subset of AI planning problems. This section will define the syntax in Figure 4.1 and relate it to the standard first-order language defined in Figure 2.1.

Predicates, variables, constants, terms, atomic formulae and substitution are defined in a standard first-order logic style as in Section 2.1. It should be noted that for any concrete planning problem the predicates, variables and constants are all finite sets despite the definitions in Figure 4.1. In this language constants will be used to represent PDDL objects and atomic formulae will be used to represent PDDL predicates.

Atomic Formulae	$\mathit{Atom} \ni A$	$::= R\ (t_1, \ldots\ t_n)$
Actions	$\mathit{Act} \ni a$	$::= \alpha\ (t_1, \ldots\ t_n)$
Plans	$\mathit{Plan} \ni f$	$::=$ halt $\mid\ \alpha; f$
Polarities	$\mathit{Polarity} \ni z$	$::= +\ \mid\ -$
States	$\mathit{State} \ni S$	$::= []\ \mid\ [A \mapsto z :: S]$
(Planning) Context	$\Gamma \ni \gamma$	$::= P(\overline{x}) \rightsquigarrow Q(\overline{x})\ :\ \alpha(\overline{x})$
Specification	$\mathit{Specification} \ni \mathit{Spec}$	$::= \Gamma \vdash P \rightsquigarrow G\ :\ f$
Preconditions	$\mathit{Preconditions} \ni P$	$::= S$
Effects	$\mathit{Effects} \ni Q$	$::= S$
Goals	$\mathit{Goals} \ni G$	$::= S$

Figure 4.1: *Syntax of PCP logic*

The PDDL formulae given in Definition 12 can be used to represent a variety of things in PDDL such as preconditions and effects of actions as well as goals. The formal language defined in this chapter introduces a notion of *state* that will be used instead of PDDL formulae. States are introduced because they are easier to work with in programming languages as they can be defined by the common list data type as shown in Figure 4.4. The definition of state removes the need to define the logical connectives $\wedge$ and $\neg$ in the formal language.

In a state, polarities $+$ and $-$ are used to denote the presence or absence of a certain atomic fact in a world. Given a polarity z and an atomic formula A, $A \mapsto z$ is a *singleton formula map*. A *state* can be given by an empty state, a singleton formula map or a conjunction of such maps (denoted by ::). This definition of state is uncommon compared to standard literature in AI planning where the word state is used to denote the representation of a world. From this point in this thesis, the word state will be used specifically to refer to the definition of state given in Figure 4.1. When referring to a state that represents a world specifically this thesis will use the term world state.

Note that both PDDL formulae and States are not only used to represent the

Representation	World State
World	{(`onTable b`) (`clear a`) (`clear b`) (`holding a`)}
PDDL Fomula	`clear b` $\wedge$ `holding a` $\wedge \neg$ `handEmpty`
State	$[(clear\ b) \mapsto + :: (holding\ a) \mapsto + :: handEmpty \mapsto -]$

Table 4.1: Comparison of Worlds, PDDL formulae and States.

world state but also Goals and the preconditions and effects of actions. When used to represent a world state a state is simply the list of true and false facts. In preconditions and goals, the state is a condition on a world state that identifies the *sets* of world states that satisfy their requirements. When used as an effect, states can be seen as descriptions of updates on world states. This overloading use of the state definition is fine for the STRIPS subset of PDDL that this thesis supports however the definitions of preconditions, effects and goals would have to have separate definitions in order to support many of the extensions of PDDL.

Including the *world* definition given in Section 2.3.3, this thesis has now introduced three ways to represent the world state. A comparison of the representations is given in Table 4.1. This comparison showcases that the world representation does not explicitly say that the predicate `handEmpty` is false as this is implicit in the world definition. Note that PDDL formulae and States have a one to one mapping and in theory can be used interchangeably to represent world states, preconditions, effects and goals however worlds can only be used to represent world states.

Let A be set of action names $\{\alpha, \alpha_1, \alpha_2, ...\}$. An action is an action name applied to a list of terms. For example, *pickup_from_table a* is an action. For any concrete planning problem, the set of action names is finite.

A *context* Γ contains descriptions of actions in the form $P(\overline{x}) \rightsquigarrow Q(\overline{x}) \ : \ \alpha(\overline{x})$ where $P(\overline{x}) \rightsquigarrow Q(\overline{x})$ denotes a transformation from a state $P(\overline{x})$ to a state $Q(\overline{x})$ and $\alpha(\overline{x})$ is an action. Just as in Definition 15, $P(\overline{x})$ describes the preconditions of an action and $Q(\overline{x})$ describes the effects of an action. Note that the formalisation given in this chapter makes two distinct choices:

1. An action is viewed as a function that inhabits the type $P(\overline{x}) \rightsquigarrow Q(\overline{x})$.
2. $P(\overline{x})$ and $Q(\overline{x})$ are represented as states, not PDDL formulae.

It is assumed that there is a context definition for all actions and that all variables in P and Q are universally bound in α. The combination of the definition of actions along with the context definition is used to describe the action definitions given in

a PDDL domain. The action set defines the names and parameters of actions and the context definition defines the preconditions and effects for actions. A PDDL Domain Σ is defined by the combined definition of predicates, actions and context.

Remark on Notation 2

To simplify the notation in this thesis, the use of notation "$(\overline{x})$" from atomic formulae, such as $R(\overline{x})$, is extended to states (e.g. $Q(\overline{x})$) and actions (e.g. $\alpha(\overline{x})$). In all these cases, the presence of $\overline{x}$ signifies the presence of free variables $\overline{x}$ in the states, actions, and constraints, respectively. This thesis will drop $\overline{x}$ and will write just Q and α to emphasise that the state or action do not contain any variables, i.e. Q and α are ground. *This thesis will also use a standard simplification of list notation where $[A \mapsto z] \equiv [A \mapsto z :: []]$.*

Given a halting state *halt*, *plans* are defined inductively as finite sequences of action names ending with *halt*. The halting state *halt* indicates the end of a plan or an empty plan and is therefore equivalent to the empty [] plan in Definition 15.

PCP logic represents the planning problem as an entailment relation referred to as a *specification*. A specification is a sequent of the form:

$$\Gamma \vdash P \rightsquigarrow G \ : \ f$$

A specification states that given a context Γ, f is a plan that gives a provable transformation from (ground) state P to (ground) state G. In the Curry-Howard interpretation of this logic, a plan f is viewed as a function that inhabits type $P \rightsquigarrow G$. The entailment rules for this specification should embody the iterative nature of plans as defined in Definition 17 where the entailment of a plan indicates the sequential entailments of all actions within the plan.

In a traditional PDDL problem the initial world state would be represented as a world that only contains a list of true atomic formulae instead of a state. This chapter opts to use states as it is possible to then represent the initial state, goal state, preconditions and effects of actions all using the same data type. This chapter will show that states correspond to PDDL formulae therefore the whole planning problem can also be defined relative to the declarative semantics. It will also be possible to define the typing relation that defines the planning problem using only a simple subtyping relation for state comparison. Using states will also allow for more flexibility when defining the model of computation for plans. One key

difference is that states do not contain all possible information about the world as it is possible that a state does not necessarily contain information about all negated atomic formulae. Chapter 6 will explore an inference system that uses worlds instead of states to describe the initial state.

4.3.2 Operational Semantics, States and Types

It is useful to establish a correspondence between states and PDDL formulae as it will allow for the formal Agda system to reason about states that are easier to work with. This can be done by defining operational semantics. Matching the declarative-style semantics of Figure 2.3, operational semantics are given by a normalisation function that acts directly on formulae and computes states. The term "normalisation" is used here, as given a formula, it produces a unique state.

Definition 21 (Normalisation of PDDL Formulae to States)**.** The function $\downarrow_z$ *normalises* a PDDL formula to a state:

$$(F \wedge F_1) \downarrow_z S = F_1 \downarrow_z F \downarrow_z S$$

$$\neg A^g \downarrow_z S = A^g \downarrow_{-z} S$$

$$A^g \downarrow_z S = A^g \mapsto z :: S$$

We write $F \downarrow_z$ to mean $(F \downarrow_z [])$.

Example 38 (Normalisation of a Formula to a State)**.**

$$(\mathit{handEmpty} \wedge \neg \mathit{onTable}\ a) \downarrow_+ = [\mathit{handEmpty} \mapsto +\ ::\ \mathit{onTable}\ a \mapsto -]$$

Note that, in order to bring the disjunction into this language in any future extensions, normalisation function for minus could be amended, to allow for non-determinism. This section will now prove that normalisation is sound relative to the declarative semantics. By a small abuse of notation, this thesis will use $\in$ to denote list membership, as well as set membership.

Definition 22 (Well-Formed World)**.**

A world w_S is a *well-formed world* for a given state S, denoted $w \in \langle S \rangle$, if the world w_S contains all A^g's such that $(A^g \mapsto +) \in S$ and contains no A^g's such that $(A^g \mapsto -) \in S$.

Unlike normalisation, w_S is generally not uniquely defined, and the notation $\langle w_S \rangle$ will be used to refer to the (necessarily finite) set of all w_S.

Example 39 (Well-Formed Worlds)**.**
If $S = (\mathit{handEmpty} \wedge \neg \mathit{onTable}\ a) \downarrow_+$, then w_S may be given by e.g. $w_1 = \{\mathit{handEmpty}\}$, or $w_2 = \{\mathit{handEmpty}, \mathit{onTable}\ b\}$, or any other world containing *handEmpty* but not $\mathit{onTable}\ a$. The given formula will be satisfied by any such w_S.

Normalisation is sound and complete (note that all of the following proofs are formalised in Agda [51]):

Lemma 1 ($\in \langle\rangle$ strengthening)
Given a world w, formulas F,F_1 and a polarity z if $w \in \langle w_{F\downarrow_z} \rangle$ and $w \in \langle w_{F_1\downarrow_z} \rangle$ then $w \in \langle w_{F_1\downarrow_z F\downarrow_z} \rangle$.

Lemma 2 ($\in \langle\rangle$ weakening)
Given a world w, formulas F,F_1 and polarities z,z_1 if $w \in \langle w_{F_1\downarrow_{z_1} F\downarrow_z} \rangle$ then $w \in \langle w_{F\downarrow_z} \rangle$.

Lemma 3 ($\in \langle\rangle$ exchange)
Given a world w, formulas F,F_1 and polarities z,z_1 if $w \in \langle w_{F\downarrow_z F_1\downarrow_{z_1}} \rangle$ then $w \in \langle w_{F_1\downarrow_{z_1} F\downarrow_z} \rangle$.

Theorem 1 (Soundness and completeness of normalisation)
Given a formula F and a world w, it holds that $w \models_z F$ iff $w \in \langle w_{F\downarrow_z} \rangle$.

Proof. ($\Rightarrow$) is proven by induction on the derivation of $w \models_z F$.

Case ($w \models_+ A_g$)
By assumption $A_g \in w$ therefore $w \in \langle A_g \downarrow_+ \rangle$.

Case ($w \models_- A_g$)
By assumption $A_g \notin w$ therefore $w \in \langle A_g \downarrow_- \rangle$.

Case ($w \models_+ \neg A_g$)
By assumption $A_g \notin w$ therefore $w \in \langle \neg A_g \downarrow_+ \rangle$.

Case $(w \models_- \neg A_g)$

By assumption $A_g \in w$ *therefore* $w \in \langle \neg A_g \downarrow_- \rangle$.

Case $(w \models_z F \wedge F_1)$

The goal $w \in \langle w_{F\downarrow_z F_1 \downarrow_{z_1}} \rangle$ *can be obtained by the application of Lemma 1 if* $w \in \langle w_{F\downarrow_z} \rangle$ *and* $w \in \langle w_{F_1\downarrow_z} \rangle$. *By assumption* $w \models_z F$ *and* $w \models_z F_1$ *therefore by induction on the assumptions* $w \in \langle w_{F\downarrow_z} \rangle$ *and* $w \in \langle w_{F_1\downarrow_z} \rangle$ *are obtained as required.*

□

Proof. ($\Leftarrow$) follows by induction on the structure of F.

Case (A_g)

By assumption $w \in \langle A_g \downarrow_z \rangle$. *If* $z = +$ *then* $A_g \in w$ *is known from the assumption therefore* $w \models_+ A$ *is obtained by application of the Somewhere rule. If* $z = -$ *then* $A_g \notin w$ *is known from the assumption therefore* $w \models_- A_g$ *is obtained by application of the Nowhere rule.*

Case $(\neg A_g)$

By assumption $w \in \langle \neg A_g \downarrow_z \rangle$. *If* $z = +$ *then* $A_g \notin w$ *is known from the assumption therefore* $w \models_- A_g$ *is obtained by the application of the Nowhere rule. By application of the Flip rule on* $w \models_- A_g$, $w \models_+ \neg A_g$ *is obtained as required. If* $z = -$ *then* $A_g \in w$ *is known from the assumption therefore* $w \models_+ A_g$ *is obtained by the application of the Somewhere rule. By application of the Flip rule on* $w \models_+ A_g$, $w \models_- \neg A_g$ *is obtained as required.*

Case $(F \wedge F_1)$

By assumption $w \in \langle F \wedge F_1 \downarrow_z \rangle$ *which normalises to* $w \in \langle w_{F_1\downarrow_z F\downarrow_z} \rangle$. *The goal* $w \models_z F \wedge F_1$ *can be obtained by the application of the Both rule if* $w \models_z F$ *and* $w \models_z F_1$ *can be obtained. By the application Lemma 2 on the assumption* $w \in \langle F \downarrow_z \rangle$ *is obtained.* $w \models_z F$ *is obtained by induction on* $w \in \langle F \downarrow_z \rangle$. *By application of Lemma 3 on the assumption* $w \in \langle w_{F\downarrow_z F_1\downarrow_z} \rangle$ *is obtained. By application of Lemma 2 on* $w \in \langle w_{F\downarrow_z F_1\downarrow_z} \rangle$, $w \in \langle F_1 \downarrow_z \rangle$ *is obtained.* $w \models_z F_1$ *is obtained by induction on* $w \in \langle F_1 \downarrow_z \rangle$.

□

Theorem 1 will allow the formal system to work with states at the type level,

while keeping the link to the standard PDDL formula syntax and declarative semantics.

This chapter will now move to its main goal: to realise the Curry-Howard intuition and define a framework in which plans will inhabit normalised formulae seen as types. This chapter aims to show that, proving that a certain (possibly composite) plan f satisfies pre- and post-conditions given by the formulae F_1 and F_2 will be equivalent to typing the judgement

$$\Gamma \vdash F_1 \downarrow_+ \rightsquigarrow F_2 \downarrow_+ : \ f$$

We will say $F_1 \downarrow_+$ is the initial state of the plan f, and $F_2 \downarrow_+$ is its goal state. The next section will introduce inference rules that define derivations of these judgements.

4.4 Plans as Proof Terms

4.4.1 Inference Rules for Planning Problems

To define the inference rules for planning problems this chapter has to address the frame problem. These inference rules can be seen as typing rules when implemented in a programming language. This chapter will take a STRIPS like approach to solving this problem where actions describe transformations on a world state rather than the resultant world state itself. On top of this, this chapter will introduce a notion of precondition satisfaction by introducing a subtyping relation.

The subtyping relation will adhere to the STRIPS assumption that the application of a ground action $P \rightsquigarrow Q \ : \ \alpha$ should not produce exactly Q, but an extension of P by Q. For example, picking up `b` from the table does not affect the fact that `c` is still on the table.

To define action application, this chapter will introduce an override operator $P \sqcup Q$:

$$\text{NilSub}\ \frac{}{S <: []} \qquad \text{ASub}\ \frac{S' <: S \quad A \mapsto z \in S'}{S' <: A \mapsto z :: S}$$

Figure 4.2: *Subtyping order on states.*

Definition 23 (Override Operator)**.**

$$P \sqcup [] = P$$

$$P \sqcup [A \mapsto z * Q] = [A \mapsto z :: P \backslash \{A \mapsto + :: A \mapsto -\}] \sqcup Q$$

The override operator adds all formula maps from one state to the other. If a mapping for a formula that is to be added already exists, then that formula is removed before adding the new formula map. Unlike STRIPS which applies actions to a world that only contains information about true atomic formulae, this chapter defines action application on states that explicitly record negated predicates. This change means that negated predicates are added to the state rather than removed.

Example 40 (Override Operator)**.**
$[(\mathit{handEmpty} \mapsto +) :: (\mathit{onTable}\ a \mapsto +) :: (\mathit{clear}\ a \mapsto +)]\ \sqcup$
$[(\mathit{handEmpty} \mapsto -) :: (\mathit{onTable}\ a \mapsto -) :: (\mathit{holding}\ a \mapsto +)]$
$= [(\mathit{holding}\ a \mapsto +) :: (\mathit{onTable}\ a \mapsto -) :: (\mathit{handEmpty} \mapsto -) :: (\mathit{clear}\ a \mapsto +)]$

It is also necessary to be able to apply an action $P \rightsquigarrow Q\ :\ \alpha$ in a state P' that is weaker (has more atomic formulae) than P. For example, if b is known to be on the table, knowing that c is also on the table should not preclude picking up b. This state-weakening action corresponds to subtyping $P' <: P$ defined in Figure 4.2. The notation $P' <: P$, denotes that P' is a *subtype* of P. This agrees with the usual convention that a subtype is given by a stronger assertion. This relation is called subtyping to refer to the fact that states can also be seen as types. Subtyping is also a satisfaction relation as if P' is a *subtype* of P then P' satisfies P.

Example 41 (Subtyping)**.** Given: $Q \equiv (\mathit{onTable}\ a \mapsto -)\ ::\ (\mathit{onTable}\ b \mapsto +)\ ::\ (\mathit{clear}\ a, b \mapsto +)\ ::\ (\mathit{holding}\ a \mapsto +)\ ::\ (\mathit{handEmpty} \mapsto -)$ and
$Q' \equiv (\mathit{onTable}\ b \mapsto +)\ ::\ (\mathit{onTable}\ a \mapsto -)\ ::\ (\mathit{clear}\ a, b \mapsto +)\ ::\ (\mathit{holding}\ a \mapsto +)$, it is obtained $Q <: Q'$.

$$\text{Halt}\ \frac{S <: S'}{\Gamma \vdash S \rightsquigarrow S' \ : \ \text{halt}} \qquad \text{Seq}\ \frac{P' <: P(\overline{x})[\sigma] \quad P(\overline{x}) \rightsquigarrow Q(\overline{x}) \ : \ \alpha(\overline{x}) \ \in \ \Gamma \quad \Gamma \vdash P' \sqcup Q(\overline{x})[\sigma] \rightsquigarrow S \ : \ f}{\Gamma \vdash : P' \rightsquigarrow S \ : \ \alpha(\overline{x})[\sigma]; f}$$

Figure 4.3: PCP logic rules.

The rules of Figure 4.3 define how a program $P \rightsquigarrow G \ : \ f$ can be typed given some planning domain Γ. A well-typed plan $\Gamma \vdash P \rightsquigarrow G \ : \ f$ "transports" an initial state P to a goal state G. To exemplify these rules, this section will now refer again to the blocks world problem with the pre-condition $F_1 = \mathit{onTable}\ a \wedge \mathit{onTable}\ b \wedge \mathit{onTable}\ c \wedge \mathit{clear}\ a \wedge \mathit{clear}\ b \wedge \mathit{clear}\ c \wedge \mathit{handEmpty}$ and the post-condition $F_2 = \mathit{on}\ a\ b \wedge \mathit{on}\ b\ c$. Suppose that the PDDL planner proposes `plan1`, as given in Section 4.2. Let plan_1 be the corresponding version in the precise mathematical notation of Figure 4.1.

$$\text{plan}_1 = \texttt{(pickup_from_table b)}; \texttt{(putdown_on_stack b c)}; \texttt{(pickup_from_table a)}; \texttt{(putdown_on_stack a b)}; \text{halt}$$

Given the initial world state F_1, goal state F_2 if $\Gamma_1 \vdash F_1 \downarrow_+ \rightsquigarrow F_2 \downarrow_+ \mid \text{plan}_1$ yields a typing derivation by Figure 4.3, then this typing derivation verifies that plan_1 correctly implements the given planning problem in the planning domain Γ_1. In this chapter the derivation $\Gamma_1 \vdash F_1 \downarrow_+ \rightsquigarrow F_2 \downarrow_+ \mid \text{plan}_1$ will refer to the PDDL problem domain given in Figure 2.2. In PCP logic, the definition of the context for the example domain is given by the following definition of Γ_1:

$\Gamma_1 = \{$

$$\begin{bmatrix} handEmpty & \mapsto + & :: \\ (onTable\ x) & \mapsto + & :: \\ (clear\ x) & \mapsto + & \end{bmatrix} \rightsquigarrow \begin{bmatrix} handEmpty & \mapsto - & :: \\ (onTable\ x) & \mapsto - & :: \\ (holding\ x) & \mapsto + & \end{bmatrix} \mid pickup_from_table\ x$$

$$\ldots \mid putdown_on_table\ x$$

$$\ldots \mid pickup_from_stack\ x\ y$$

$$\begin{bmatrix} (holding\ x) & \mapsto + & :: \\ (clear\ y) & \mapsto + & \end{bmatrix} \rightsquigarrow \begin{bmatrix} (holding\ x) & \mapsto - & :: \\ (clear\ y) & \mapsto - & :: \\ (on\ x\ y) & \mapsto + & :: \\ handEmpty & \mapsto + & \end{bmatrix} \mid putdown_on_stack\ x\ y\}$$

This chapter will now perform the typing derivation for:

$$\Gamma_1 \vdash F_1 \downarrow_+ \rightsquigarrow F_2 \downarrow_+ \mid \text{plan}_1$$

Given $F_1 \downarrow_+$, then the first action that can be applied by the Seq rule is *pickup_from_table b*. The application of Seq demands that the initial state of the action *pickup_from_table b* in Γ_1 is a subtype of $F_1 \downarrow_+$. A subtyping derivation provides such a proof, selecting the required piece of evidence from $F_1 \downarrow_+$, i.e.

$$\begin{bmatrix} (onTable\ a, b, c) & \mapsto + & :: \\ (clear\ a, b, c) & \mapsto + & :: \\ handEmpty & \mapsto + & \end{bmatrix} <: \begin{bmatrix} handEmpty & \mapsto + & :: \\ (onTable\ b) & \mapsto + & :: \\ (clear\ b) & \mapsto + & \end{bmatrix}$$

At this point of the derivation $\text{plan}_1 = (pickup_from_table\ b);\ f'$ is now verified. To complete the proof and compute an action for f', it must be shown that the remainder of the plan is typeable. The application of Seq, produced a new state P_1 as well as an obligation to prove $f' : P_1 \rightsquigarrow F_2 \downarrow_+$.

$$P_1 = F_1 \downarrow_+ \sqcup \begin{bmatrix} handEmpty & \mapsto - & :: \\ (onTable\ b) & \mapsto - & :: \\ (holding\ b) & \mapsto + & \end{bmatrix} = \begin{bmatrix} (onTable\ a) & \mapsto + & :: \\ (onTable\ b) & \mapsto - & :: \\ (onTable\ c) & \mapsto + & :: \\ (clear\ a, b, c) & \mapsto + & :: \\ (handEmpty) & \mapsto - & :: \\ (holding\ b) & \mapsto + & \end{bmatrix}$$

The proof proceeds by trying to apply the Seq rule to next action from plan_1: *putdown_on_stack b c*. Again P_1 is readily shown to be a subtype of the preconditions of *putdown_on_stack b c*. The proof continues in this way for each action in plan_1, the final state is:

$$P_3 = \begin{bmatrix} (onTable\ a, b) & \mapsto - & :: \\ (onTable\ c) & \mapsto + & :: \\ (clear\ a) & \mapsto + & :: \\ (clear\ b, c) & \mapsto - & :: \\ (on\ b\ c) & \mapsto + & :: \\ (handEmpty) & \mapsto + & :: \\ (holding\ a, b) & \mapsto - & :: \\ (on\ a\ b) & \mapsto + & \end{bmatrix}$$

However, this is not the same state as the goal state G_0. The Halt rule was introduced to resolve such cases as it eliminates all unnecessary evidence from the current state by proof of subtyping i.e. $\Gamma_1 \vdash P_3 \rightsquigarrow G_0 \ : \ \text{halt}$. Clearly $P_3 <: [(on\ a\ b) \mapsto + \ :: (on\ b\ c) \mapsto +]$ as required. After this application of Halt, $\Gamma_1 \vdash \text{plan}_1 : F_1 \downarrow_+ \rightsquigarrow F_2 \downarrow_+$ is now verified.

4.4.2 Computational Characterisation of Plans: Soundness of Plan Execution

The proof of $\Gamma_1 \vdash F_1 \downarrow_+ \rightsquigarrow F_2 \downarrow_+ \mid \text{plan}_1$ provides evidence that the *execution* of plan_1 on a world satisfying F_1 produces a new world satisfying F_2. Generally, the inference of $\Gamma \vdash P \rightsquigarrow G \ : \ f$, with $f = \alpha_1; \ldots; \alpha_j; \text{halt}$ corresponds to successively applying actions $\alpha_1 \ldots \alpha_j$ to states $P, P_1, \ldots P_j$ in a sequence of state transitions, satisfying $P_j <: G$. This section will now prove that the plan f thus inferred indeed has a computational meaning, i.e. can be *evaluated*, and that the result of its evaluation is sound. To state this, an *evaluation function* $[\![.]\!]$ that will interpret actions on worlds needs to be defined. Recall that every state S maps to a world w_S, as shown in Example 4.3.2. This thesis will use the notation δ for an arbitrary mapping (an *action handler*) that maps each ground action $P \rightsquigarrow Q \ : \ \alpha$ to insertions and deletions on the world w_P according to α's action on P. The evaluation function $[\![]\!]^\delta \ w$ that *evaluates* a plan to a world (according to a given world w and action handler δ) can then be defined as:

Definition 24 (Evaluation Function)**.**

$$[\![\text{halt}]\!]^\delta \ w = w$$

$$[\![\alpha; f]\!]^\delta \ w = [\![f]\!]^\delta \ (\delta \ \alpha \ w)$$

This section will now proceed to define the notion of a well-formed handler, that will be used to prove soundness of the PCP logic.

Definition 25 (Well-Formed Handler)**.** An action handler δ is *well-formed* if, given:

- a context Γ with $P'(\overline{x}) \rightsquigarrow Q(\overline{x}) \ : \ \alpha(\overline{x}) \ \in \ \Gamma$,
- a state P, such that $P(x) <: P'(\overline{x})[\sigma]$ for some ground substitution σ,
- a world $w \in \langle w_P \rangle$,

δ satisfies the following property: $(\delta \ (\alpha(\overline{x})[\sigma]) \ w) \in \langle w_{P \sqcup Q(\overline{x})[\sigma]} \rangle$.

Canonical Handler. In order for this thesis to be constructive in its further claims, and to provide a practical solution to the quest for a well-formed handler, a well-formed *canonical handler* will now be defined for a given planning domain.

The first step is to define a function δ_α that updates a world given an effect:

Definition 26 (The update world function δ_α)**.**

$$\delta_\alpha\ [\ ]\ w = w$$

$$\delta_\alpha\ [A \mapsto + :: Q]\ w = \delta_\alpha\ Q\ (w \cup \{A\})$$

$$\delta_\alpha\ [A \mapsto - :: Q]\ w = \delta_\alpha\ Q\ (w \backslash \{A\})$$

This function updates a world given an effect by adding all of the positively mapped predicates to the world and removing all the negatively mapped predicates. When this function is applied to an empty world it simply creates the smallest world that satisfies the effects. When δ_α is applied to a non-empty world, it is equivalent to applying the effects of an action on the world. The canonical handler can then be defined when given a context Γ and a ground action $\alpha(\overline{x})[\sigma]$ by applying δ_α to $Q(\overline{x})[\sigma]$ for each $P(\overline{x}) \rightsquigarrow Q(\overline{x})\ :\ \alpha(\overline{x})$ in Γ. This gives a canonical mapping from actions and worlds into worlds, as required. The resulting canonical action handler canonical-δ Γ (defined formally in Figure 4.6) is well-formed, as long as the states to which it is applied are consistent, i.e. in the following sense:

Definition 27 (Implicit consistency assumption)**.** *Implicit consistency assumption:* for every state S, if $A \mapsto z \in S$ then $A \mapsto -z \notin S$.

The implicit consistency assumption is needed for the following lemmas about the update world function:

Lemma 4 (Update world inserts all positive formulae)
Given a state S and a world w if $(A \mapsto +) \in S$ then $A \in \delta_\alpha\ S\ w$.

Lemma 5 (Update world deletes all negative formulae)
Given a state S and a world w if $(A \mapsto -) \in S$ then $A \notin \delta_\alpha\ S\ w$.

By definition of the update world function, it is known that a positive mapping is added to a world and a negative mapping is deleted from a world. It is however possible in an inconsistent state that after a formula has either been added to or deleted from the given world that the formula appears again in state S with a

different mapping causing the lemmas to not hold. It is therefore necessary to know that the state is consistent to prove the lemmas. This lemma is simply proven by induction on S and discharging the possibility of inconsistent states with the implicit consistency assumption.

Lemma 6 (Union property)

Given a formula map $(A \mapsto z)$, a state P and a state Q, if $A \mapsto z \in P \sqcup Q$ then:

- $(A \mapsto z) \in P$ *and* $(A \mapsto \neg z) \notin Q$, *or*
- $(A \mapsto z) \in Q$.

Lemma 7 (Canonical handler keeps non-negated facts)

Given a predicate A, a world w, and a state S, if $A \in w$ and $A \mapsto - \notin S$ then $A \in \delta_\alpha\ S\ w$

Lemma 8 (Canonical handler does not add additional facts)

Given a predicate A, a world w, and a state S, if $A \notin w$ and $A \mapsto + \notin S$ then $A \notin \delta_\alpha\ S\ w$

Proposition 1

Given a context Γ, the canonical handler canonical-δ Γ *is well-formed with respect to Γ.*

Proof. By assumption there exists a world w, a ground action α with a transformation $P \rightsquigarrow Q$ and $w \in \langle P \rangle$. The proof needs to consider two cases:

Case $(A \in P \sqcup Q)$

The goal of this case is to show that $A \in \delta_\alpha\ S\ w$. By application of the union lemma is it obtained $(A \mapsto +) \in P$ and $(A \mapsto -) \notin Q$, or $(A \mapsto +) \in Q$.

If $(A \mapsto +) \in P$ and $(A \mapsto -) \notin Q$ is obtained then it is known that $A \in w$ from the assumption $w \in \langle P \rangle$. The proof is then completed by the application of Lemma 7.

If $(A \mapsto z) \in Q$ the goal is obtained by the application of Lemma 4.

Case $(A \notin P \sqcup Q)$

The goal of this case is to show that $A \notin \delta_\alpha\ S\ w$. By application of the union lemma is it obtained $(A \mapsto -) \in P$ and $(A \mapsto -) \notin Q$, or $(A \mapsto -) \in Q$.

If $(A \mapsto -) \in P$ and $(A \mapsto -) \notin Q$ is obtained then it is known that $A \notin w$

from the assumption $w \in \langle P \rangle$. The proof is then completed by the application of Lemma 8.

If $(A \mapsto z) \notin Q$ then the goal is obtained by the application of Lemma 5.

□

The next two theorems show that executing a well-typed plan f by the evaluation function $[\![f]\!]^{\delta}\ w$ is *sound*, for any well-formed handler δ.

Theorem 2 (Soundness of evaluation for normalized formulae)
Suppose $\Gamma \vdash P \rightsquigarrow G\ :\ f$. Then for any $w \in \langle w_P \rangle$, and any well-formed handler δ, it follows that $[\![f]\!]^{\delta}\ w \in \langle w_G \rangle$.

Proof. The proof proceeds by structural induction on the typing derivation $\Gamma \vdash P \rightsquigarrow G\ :\ f$.

Case (Halt)
By assumption $w \in \langle w_P \rangle$ and thus because $P <: G$, it follows $w \in \langle w_G \rangle$. Since $[\![\text{halt}]\!]^{\sigma}\ w = w$, $[\![\text{halt}]\!]^{\sigma}\ w \in \langle w_G \rangle$ is obtained as required.

Case (Seq)
Note that by assumption: $f = \alpha(\overline{x})[\sigma]; f'$ and therefore $P'(\overline{x}) \rightsquigarrow Q(\overline{x})\ :\ \alpha(\overline{x})$ is in Γ and $\Gamma \vdash P \sqcup Q(\overline{x})[\sigma] \rightsquigarrow G\ :\ f'$. Then by induction every $w' \in \langle w_{P \sqcup Q(\overline{x})[\sigma]} \rangle$ gives $([\![f]\!]^{\sigma}\ w') \in \langle w_G \rangle$ for any well-formed δ. However, by the well-formedness of δ and because $w \in \langle w_P \rangle$, it obtained that $(\sigma\ \alpha\ w) \in \langle w_{P \sqcup Q(\overline{x})[\sigma]} \rangle$. Thus $[\![f']\!]^{\sigma}\ (\sigma\ \alpha\ w) \in \langle w_G \rangle$ and therefore $[\![f]\!]^{\sigma}\ w \in \langle w_G \rangle$.

□

Theorem 3 (Soundness of Evaluation)
Suppose $\Gamma \vdash F_1 \downarrow_{+} \rightsquigarrow F_2 \downarrow_{+} \mid f$ then for any w such that $w \models_{+} F_1$, and any well-formed δ it follows $[\![f]\!]^{\delta}\ w \models_{+} F_2$.

Proof. By assumption $w \models_{+} F_1$ and by the completeness of normalisation (Theorem 1), $w \in \langle w_{F_1 \downarrow_{+}} \rangle$ is obtained. By the application of Theorem 2 using $w \models_{+} F_1$ and $w \in \langle w_{F_1 \downarrow_{+}} \rangle$, $[\![f]\!]^{\delta}\ w \in \langle w_{F_2 \downarrow_{+}} \rangle$ is obtained. Thus by the soundness of normalisation (Theorem 1), $[\![f]\!]^{\delta}\ w \models_{+} F_2$ is obtained. □

Thus the derivation of a type for a plan f induces a proof that the execution of a plan in world w is correct. Although neither of the above theorems depends on the implicit consistency assumption for its proofs, the existence of a well-formed and canonical handler is predicated upon the consistency assumption. Given a derivation of a plan it is now possible to compute the world generated for any plan. For the blocks world example, a user can directly evaluate $[\![\text{plan}_1]\!]^{\delta}\ w$ by plugging in:
– in place of w – the world resulting from computing $\delta_\alpha(F_1 \downarrow_+ [\ ])\ \emptyset$. (To see this, recall that F_1 is the formula that described the initial state in all examples of the previous section, and $F_1 \downarrow_+ [\ ]$ is the state resulting from normalising F_1.)
– and in place of δ – the canonical handler for Γ_1. (Recall that Γ_1 is the context that defined the given planning domain in the previous section.)

4.5 Agda Formalisation

The Agda formalisation of the plan validation system described in this chapter will allow users to fully utilise the theoretical applications of this work. PDDL domains and problems will be able to be implemented in Agda and plans verified by the typing derivation. If a derivation for a plan type checks in the Agda system then it is known that the plan is valid. The Agda implementation will also allow users to harness the computational properties of the plans defined in this chapter. The definition of the canonical handler will allow users to execute plans on worlds. On top of this, this section will show that it is possible to extract the execution of plans into executable Haskell or binary. Chapter 6 will show that it is possible to fully automate the verification, execution and extraction processes.

4.5.1 Approach to Formalisation

A large amount of the Agda formalisation of PCP logic can be done through inductive data type declarations that exactly map to the definitions given in this thesis. For example, the declarative semantics and typing relation are simply defined as inductive data types in Figure 4.4 and Figure 4.5. Two notable exceptions are worlds and states which are defined as lists to take advantage of the extensive library of list operations that Agda has. Plans are defined as a list of actions where the empty

Declarative and Operational Semantics

```
data Form : Set where
  _∧_ : Form → Form → Form
  ¬_ : Predicate → Form
  atom : Predicate → Form

FormulaMap : Set
FormulaMap = (Polarity × Predicate)

World : Set
World = List Predicate

_↓[_]_ : Form → Polarity → State → State
F ∧ F₁ ↓[ z ] S = F₁ ↓[ z ] F ↓[ z ] S
¬ x ↓[ z ] S = (neg z , x) :: S
atom x ↓[ z ] S = (z , x) :: S

data Plan : Set where
  _::_ : Action → Plan → Plan
  halt : Plan

data Polarity : Set where
  + - : Polarity

State : Set
State = List FormulaMap

data _⊨[_]_ : World → Polarity → Form → Set where
  flip : ∀{w t A} → w ⊨[ neg t ] (atom A) → w ⊨[ t ] ¬ A
  both : ∀{w t P Q} → w ⊨[ t ] P → w ⊨[ t ] Q → w ⊨[ t ] P ∧ Q
  somewhere : ∀{w a} → a ∈ w → w ⊨[ + ] atom a
  nowhere : ∀{w a} → a ∉ w → w ⊨[ - ] atom a
```

Figure 4.4: Agda implementation of Declarative and Operational Semantics of PCP logic defined in Section 2.3.3 and Section 4.3.

Typing Relation

```
del : Predicate → State → State
del x [] = []
del x ((t' , x') :: P) with x ≟p x'
del x ((t' , x') :: P) | yes p = del x P
del x ((t' , x') :: P) | no ¬p = (t' , x') :: del x P

_⊔N_ : State → State → State
P ⊔N [] = P
P ⊔N ((z , q) :: Q) = (z , q) :: del q P ⊔N Q

data _⊢_⇝_:_ : Context → State → State → Plan → Set where
  halt : ∀{Γ currentState goalState} → currentState <: goalState
                → Γ ⊢ currentState ⇝ goalState : halt
  seq : ∀{α currentState goalState Γ f}
      → currentState <: preconditions (Γ α)
      → Γ ⊢ currentState ⊔N effects (Γ α) ⇝ goalState : f
      → Γ ⊢ currentState ⇝ goalState : (α :: f)
```

Figure 4.5: Agda implementation of the typing relation defined in Section 4.4.1.

Computational Characteristics of Plans

```
ActionHandler : Set
ActionHandler = Action → World → World

⟦_⟧ : Plan → ActionHandler → World → World
⟦ (α :: f) ⟧ δ w = ⟦ f ⟧ δ (δ α w)
⟦ halt ⟧ δ w = w

δα : State → World → World
δα [] w = w
δα ((+ , x) :: N) w = x :: δα N w
δα ((- , x) :: N) w = remove x (δα N w)

canonical-δ : Context → ActionHandler
canonical-δ Γ α = δα (effects (Γ α))
```

Figure 4.6: Agda implementation of the computational characteristics of plans defined in Section 4.4.2.

list is renamed to halt to improve readability. Functions are also directly translated to Agda as functions such as the normalisation function in Definition 4.3.2. The canonical handler and the evaluation function are also defined using functions in Agda as shown in Figure 4.6. Functions implemented in Agda are guaranteed to terminate therefore the evaluation function and canonical handler are guaranteed to terminate.

Remark on Notation 3

The use of the word Predicate in the Agda definitions is overloaded where the data declaration of Predicate really describes how to construct atomic formulae. This naming convention for Predicate was adopted to better match PDDL.

The Agda implementation needs to have an abstract definition of a domain in order to reason about planning domains and problems in general. For example, the definition of the typing relation given in Figure 4.4 is defined in a generic Agda module Plan that takes in a domain as a parameter. This will allow the typing derivation to be used for any domain that can be expressed in PCP logic. The module declaration is defined as:

```
module Plans.Plan (domain : Domain) where
```

The abstract definition of a domain is given as a record type that requires a

Abstract Domain

```
record Domain : Set₁ where
  Action    : Set
  Predicate : Set
  Γ         : Context
  _≟p_      : DecidableEquality Predicate
```

```
record ActionDescription : Set where
  field
    preconditions : Preconditions
    effects       : Effects
```

```
Context : Set
Context = Action → ActionDescription
```

Concrete Blocks World Domain

```
data Object : Set where
  a b c : Object

Γ : Context
Γ (pickup_from_table x) = record {
  preconditions =
    (+ , handempty) ::
    (+ , ontable x) ::
    (+ , clear x) :: [] ;
  effects =
    (- , handempty) ::
    (- , ontable x) ::
    (+ , holding x) :: [] }
Γ (putdown_on_table x) = ...
Γ (pickup_from_stack x y) = ...
Γ (putdown_on_stack x y) = ...
```

```
data Predicate : Set where
  clear : Object → Predicate
  on : Object → Object → Predicate
  ontable : Object → Predicate
  holding : Object → Predicate
  handempty : Predicate

data Action : Set where
  pickup_from_table :
    Object → Action
  putdown_on_table :
    Object → Action
  pickup_from_stack :
    Object → Object → Action
  putdown_on_stack :
    Object → Object → Action
```

Figure 4.7: *Agda definition of an abstract domain and the blocks world domain.*

concrete definition of actions, predicates and a context to be defined. Figure 4.7 shows the abstract definition of a domain as well as a concrete example of the blocks world domain that corresponds to the PDDL definition in Figure 2.2. A domain definition also requires a proof of decidable equality for the Predicate data type. A proof of decidable equality for predicates is given by defining an evidence-based procedure for comparing the equality of predicates. This is needed throughout formalisation, for example, it is needed in the del function in Figure 4.5 that deletes all instances of a predicate from a state. One caveat with proving decidable equality for predicates it that it is also necessary to prove decidable equality over objects for the proof. This thesis will address this issue by including objects in the domain

declaration unlike in PDDL which includes the object definition in the problem file. Including the object definition in the domain definition has no effect on plan validity and corresponds to the traditional definition of a domain in a planning problem given in Definition 6.

The approach taken to the Agda formalisation has a focus on simplicity in both the implementation and readability of the code. This was accomplished by reusing code from the standard library wherever possible as well as utilising predefined theory within Agda. One example of this is illustrated in Figure 4.7 where the handling of variables for all definitions is left to Agda. Simplicity is also one of the primary reasons why states are represented as lists of pairs rather than using AVL trees. It would, however, be worthwhile to implement states as AVL trees in future work to see if there is a notable performance increase. Representing states as functions from Predicate → Polarity was not chosen as it overly complicates the definitions that use states. For example, functions complicate definitions where the state has to be printed. Functions are also unnecessary as states generally only need to encode mappings for a small portion of the possible predicates.

4.5.2 Verifying and Executing Plans in Agda

The Agda implementation of PCP logic defined in this section allows for the validation and execution of plans for PDDL problems. By instantiating the Plan module with a domain it is possible to validate a plan by defining the initial state, goal state and using the typing relation. Figure 4.8 shows the code for the validation and execution of the running blocks world example defined in Figure 2.2. The initial and goal state of the problem is given by the PDDL formulae F_1 and F_2 respectively. The example plan, $\texttt{plan}_1$, used in this chapter is defined by $plan_1$. The derivation given in Section 4.4.1 is then defined by the Derivation function. If it is possible to define a program that inhabits the type of the derivation then the plan is validated and it is ensured that plan_1 can be soundly executed as a function. Agda ensures that a program inhabits the derivation type via type checking. The Derivation function type checks in Agda therefore it is known the plan is valid for the given problem.

The function formula-eval executes the validated plan on the initial state using the canonical handler. It should be noted that the initial state is converted into a

```
F₁ : Form
F₁ = atom (onTable a) ∧ atom (onTable b) ∧ atom (onTable c) ∧
  atom (clear a) ∧ atom (clear b) ∧ atom (clear c) ∧ atom handEmpty

F₂ : Form
F₂ = atom (on a b) ∧ atom (on b c)

plan₁ : Plan
plan₁ = pickup_from_table b :: putdown_on_stack b c :: pickup_from_table a
        :: putdown_on_stack a b :: halt

Derivation : Γ ⊢ (F₁ ↓+) ⇝ (F₂ ↓+) : plan₁
Derivation =
  seq
    (λ { (here refl) → here refl
         ; (there (here refl)) → there (there (there (there (there (here refl)))))
         ; (there (there (here refl))) → there (there (here refl))})
  (seq (λ { (here refl) → there (there (there (here refl)))
          ; (there (here refl)) → there (there (there (there (here refl))))})
  (seq (λ { (here refl) → there (there (there (here refl)))
          ; (there (here refl))
            → (there (there (there (there (there (there (there (there (here refl)))))))))
          ; (there (there (here refl)))
            → there (there (there (there (there (there (here refl))))))})
  (seq (λ { (here refl) → there (there (there (here refl)))
          ; (there (here refl))
            → (there (there (there (there (there (there (there (here refl)))))))})
  (halt λ { (here refl)
            → there (there (there (there (there (there (there (there (here refl))))))))
          ; (there (here refl)) → there (there (here refl))})))))

formula-eval : World
formula-eval = ⟦ plan₁ ⟧ (canonical-δ Γ) (δα (F₁ ↓[ + ] []) [])
```

Figure 4.8: *Agda type checking the derivation of* $\Gamma_1 \vdash F_1 \downarrow_{+} \rightsquigarrow F_2 \downarrow_{+} : \ plan_1$.

world by using the $\delta\alpha$ function as described. The evaluation of formula-eval results in the world $w' = \{$ `handEmpty`, `on a b`, `on b c`, `clear a`, `onTable c`$\}$. That is, the world that corresponds to the state P_3 of the previous section.

It is also possible to go one step further, and use Agda's code extraction library and compile the verified plans into executable Haskell programs or executable binary. The process is fully automated by existing Agda libraries, and subsequent execution of the binary takes just milliseconds. The generated code will run a plan on the given world using the evaluation function and return a string of the resultant world. The world produced by the execution of the extracted code is also guaranteed to satisfy the goal of the planning problem if it has been proven by the typing derivation. The binary file can in principle be run on any independent platform, e.g. on a robot. Example code that can be compiled and run is given in the repository [51].

4.6 Approach to State Consistency

As Proposition 1 has shown, the existence of a canonical and well-formed handler depends crucially on the implicit consistency assumption. At the same time, the proofs of Theorems 2 or 3 do not depend on the consistency assumption. The introduction of the inconsistency assumption makes it possible in principle to construct planning domains and problems that violate the assumption but are accepted by the well-typedness relation of Figure 4.3. If domains are added that contain ill-formed actions that can produce an inconsistent state then $\bot$ will become provable. It should be impossible to prove $\bot$ in Agda as it has no constructor therefore the formal system is not sound when the consistency assumption is violated.

Example 42 (Ill-Formed Action)**.**

An ill-formed action can be formed by an action that produces an inconsistent state with no preconditions. Appendix A shows a derivation of $\bot$ using this action.

$$[] \rightsquigarrow \begin{bmatrix} handEmpty & \mapsto - & :: \\ handEmpty & \mapsto + & \end{bmatrix} \mid naughty$$

If it is assumed that this formal system could handle domains that contain ill-

formed actions then the implicit consistency assumption needs to be removed. But without the assumption, it is no longer possible to define a well-formed and canonical handler and therefore the ability to evaluate plans is lost. This situation is of course illustrative of the rigour and transparency that a constructive approach brings to verification. It is necessary to ensure that any practical deployment of the presented prototype needs to enforce the consistency assumption. This thesis will address this issue in two different ways. In Chapter 5 a new formal inference system will be implemented which is guaranteed to be consistent however this system has scalability issues. Chapter 6 will modify the inference system presented in this chapter so that it is guaranteed to be consistent whilst also addressing scalability concerns.

4.7 Discussion

This chapter has given a formalisation of a plan validator in Agda for the STRIPS subset of AI planning problems that can be expressed in PDDL. This formalisation addresses Objective 1, Objective 3 and Objective 4. In line with the Curry-Howard approach to first-order logic, this chapter formulated an inference system that treats planning domains as types, and generated plans as functions that inhabit these types. Type checking then ensures the soundness of these executable functions relative to the specifications given as types. This approach addresses Objective 2 and allows for the ability to generate executable code directly from plans verified in Agda.

The Agda system presented in this chapter was defined in a modular style that is compatible with any domain and problem that is definable in the STRIPS subset of PDDL. As discussed in Section 4.5.2, to validate a different planning problem it is only necessary to define the domain and problem from the relevant PDDL files and simply instantiate the relevant modules with the domain as was shown in Figure 4.7. One issue with this approach is that manually translating and proving derivations for PDDL problems into Agda is a time consuming task and becomes unfeasible with larger problems. It is therefore necessary to automate the translation and verification process for plans, which is a problem that is addressed in the subsequent chapters.

4.8 Related Work

Verification of AI languages and applications is an active research field. In planning languages, two major trends exist. Firstly, PDDL is used to verify autonomous systems and applications, see e.g. [106]; and it has been successfully integrated within other similar languages, such as GOLOG [68], with the purpose of verifying plans written in the situation calculus [22, 129]. Secondly, planning domains have been verified using model checkers [65], other automated provers such as Event-B [35], or planning support tools such as VAL [57]. The method presented in this chapter is complementary to these two trends. Its main difference lies in taking the perspective of *intrinsic*, rather than *external* verification. That is, the correctness of the generated plans is verified not by an external tool, such as a model checker, but is performed intrinsically within the code that implements the plans. At the same time, the code that implements the plans is inseparable from the language in which planning domains are specified. Furthermore, the executable binary files automatically extracted from Agda bear the verification guarantees provided by Agda proofs. This chapter's novel Curry-Howard approach to plan validation fully embodies the correspondence and all of its benefits. This is because provisioning types for plans not only equips planners with certificates of correctness for inspection, but also provides a direct link to an implementation's type theory.

Comparison to other validation methods. The work presented in this chapter provides a strong level of verification for plans as well as a certificate of correctness. There are two major drawbacks to the approach taken in this chapter. The first drawback is that the work presented in this chapter does not automate the validation of plans or the translation process from PDDL. This makes the system infeasible to use in practice in comparison to state-of-the-art validators [3, 57]. Methods for mitigating the drawback will be introduced in Chapter 5. The other drawback is that the system presented in this chapter only validates a small portion of PDDL in comparison to Val [57]. The subset of PDDL that PCP logic can validate still encapsulates a large number of planning problems and is in line with the subset of PDDL that other formally verified validators [3] work with. Because PDDL is such a vast and expressive language planners also only work with a specific subset of PDDL.

Formalisation of Planning in Other Theorem Provers. Much of the work proposed in this chapter could be replicated in another dependently-typed prover, such as Coq, Idris or Lean. The main reasons that Agda was chosen over other interactive theorem provers is due to its extensive standard library and easy-to-use interactive mode of development. There has been an impressive line of work by Abdulaziz and co-authors on formal verification of plan validators in Isabelle/HOL, see e.g. [1, 3, 4]. One advantage of using Isabelle/HOL for PDDL formalisations is the availability of extensive mathematical libraries that can support proof development. Specifically in [3] Abdulaziz et al. create a formally verified validator for the STRIPS subset of planning problems. Their approach to validation differs from the work presented in this chapter because they validate the plan validation algorithm itself which is later extracted and can be executed on a variety of planning problems. The advantage of this approach is that the produced executable validator has comparable validation speeds to the state-of-the-art validator Val. The disadvantage of their approach is that they lose all the formal guarantees that are gained by implementing the validator in a dependently-typed programming language. Many sanity checks that had to be proven in their systems such as substitution and termination are checked and guaranteed automatically by Agda's typechecker. Another advantage of our approach is that by defining plan validation as a typing relation, validated plans have checkable evidence for why a plan has been validated. The typing relation can also be used to give guarantees about extracted plans.

Chapter 5

Resource Logic for AI Planning

5.1 Introduction

Chapter 4 introduced an inference system for the AI planning problems in a STRIPS style where actions are defined by adding and removing atomic predicates from a state. One issue with the STRIPS approach is that an implicit consistency assumption is needed in order to evaluate plans. When the work from Chapter 4 was presented at the Symposium on Practical Aspects of Declarative Languages in 2019, audience members recognized the link between the problem of local reasoning with AI and resource logic. Frame axioms were historically a common solution for local reasoning in AI planning [25, 48]. Due to this audience feedback and the success of research done in resource semantics [93, 105] this thesis will re-investigate the use of a frame rule in AI planning.

This chapter will present a novel resource logic for validating plans, the *Proof Carrying Plans Resource (PCPR) logic*, that embeds consistency as an integral part of the logic. PCPR logic is inspired by resource semantics, and by the Curry-Howard view on Separation logic as given e.g. in [81, 102]. The key logic features are:

- Hoare triples to describe plans and states
- The frame rule for local resource-aware reasoning
- The Curry-Howard view on states as types, and state transformations as functions

As shown in Chapter 4, the final feature ensures that the plans that are verified in the logic are also executable functions.

```
(define (problem blocksworld)
  (:domain blocksworld)
  (:objects a b)
  (:init (onTable a)
         (onTable b)
         (clear a)
         (clear b)
         (handEmpty))
  (:goal (and (on a b) (onTable b))))
```

```
(:action putdown_on_stack
  :parameters
     (?x ?y)
  :precondition
     (and
        (not (= ?x ?y))
        (holding ?x)
        (clear ?y))
  :effect
     (and
        (not (holding ?x))
        (not (clear ?y))
        (on ?x ?y)
        (handEmpty))))
```

Figure 5.1: *Blocks world planning problem and updated* `putdown_on_stack` *action with added constraint.*

One major departure in PCPR logic is the use of a frame rule for local reasoning instead of using the standard STRIPS approach. The use of a frame rule will allow for the definition of an inference system at a higher level of abstraction than Chapter 4. This higher level of abstraction allows for the definition of clear and intuitive formal semantics that helps to clarify the computational properties of plans.

For example, the embedding of consistency into the rules of the logic will lead to a discussion in Section 5.4 about the potential effects of inconsistent actions.

From the theoretical perspective, this chapter will introduce an alternative formalisation of plan validation and will prove it sound relative to the declarative semantics. This chapter will also add support for PDDL constraints into the logic as well as expose additional properties about various aspects concerning both the PCP logic and PCPR logic inference systems. From the practical perspective, the logic and soundness proofs will be fully formalised in Agda [51]. Moreover, in Section 5.6 the full Curry-Howard, or functional, value of the Agda implementation will be showcased by supplementing the Agda formalisation with automation that allows the verification of PDDL problems automatically. This will include the introduction of a parser that converts PDDL domain and problem specifications into their equivalent Agda code as well as a solver that automatically produces Agda derivations for plans.

5.1.1 Results of this chapter by means of an example

This chapter will use the Blocks World domain for its running example however the domain will be updated to include constraints on actions and a different Blocks World problem will be used. An example updated `pickup_from_table` action and the new problem definition is defined in Figure 5.1. Given the new problem definition a PDDL planner could solve the problem with the plan:

$$f_{ab} = ((pickup_from_table\ a)\ ;\ (putdown_on_stack\ a\ b)).$$

This chapter will formalise a proof system which can prove the correctness of this plan whilst using the frame rule for local reasoning. This will be done just as in Chapter 4 by forming a typing relation with judgements of the form:

$$\Gamma \vdash \{P\} \rightsquigarrow \{G\} \mid f$$

This judgement is inspired by Hoare logic. If the context is removed then it is obvious that the judgement is naturally inspired by Hoare triples [54] e.g. $\{P\} \rightsquigarrow \{G\} \mid f$. The need for a context that describes a set of axioms describing the domain of a planning problem is however a minor departure from Hoare logic.

There are two additional problems that need to be solved to define the context in this chapter. The first problem is how to represent the addition of constraints to the domain. Figure 5.1 shows the `pickup_from_table` with the additional inequality constraint $x \neq y$ in its preconditions. This constraint ensures that when the action is given a ground substitution, the objects `x` and `y` do not equal each other. This chapter will follow the resource logic tradition [12, 102] and separate state descriptions from constraints on states. To do this, this chapter will further refine domain descriptions to have the syntax $\phi; \{P\} \rightsquigarrow \{Q\} \mid action$, where ϕ defines constraints on states. For technical reasons, this chapter will formalise ϕ to be a list (rather than a conjunction) of constraints, and will use equality ($=$) and inequality ($\neq$) constraints instead of using negation explicitly. The inference systems presented in this thesis only support a fragment of PDDL therefore states can only contain predicates. Since predicates can only contain objects it is only possible to define equality constraints. In PDDL [37] it is possible for states to contain numeric fluents

$$[]; \begin{Bmatrix} handEmpty \mapsto + \\ * \ onTable\ x \mapsto + \\ * \ clear\ x \mapsto + \end{Bmatrix} \rightsquigarrow \begin{Bmatrix} handEmpty \mapsto - \\ * \ onTable\ x \mapsto - \\ * \ holding\ x \mapsto + \\ * \ clear\ x \mapsto + \end{Bmatrix} \mid \alpha_1\ x$$

where $\alpha_1 \equiv pickup_from_table$

$$[x \neq y]; \begin{Bmatrix} holding\ x \mapsto + \\ * \ clear\ y \mapsto + \end{Bmatrix} \rightsquigarrow \begin{Bmatrix} holding\ x \mapsto - \\ * \ clear\ y \mapsto - \\ * \ on\ x\ y \mapsto + \\ * \ handEmpty \mapsto + \end{Bmatrix} \mid \alpha_2\ x\ y$$

where $\alpha_2 \equiv putdown_on_stack$

Figure 5.2: Context Γ_{BW} that defines Blocks World PDDL domain with constraints.

which assign numeric values to zero or more objects. If states could contain numeric fluents then numeric constraints such as $\leq$ could be added.

The second problem occurs with the representation of actions when using the frame axiom approach to local reasoning rather than the STRIPS approach. The PDDL domains that are being used in this thesis represent actions as effects that contain additions and deletions to a world. Chapter 4 took a STRIPS style approach to local reasoning by defining inference rules using an override operator that added and deleted predicates from a state. This chapter will instead represent actions as preconditions and postconditions where the result of an action is simply the postcondition. This has many implications which will be discussed in Section 5.4. Figure 5.2 defines the context Γ_{BW} that gives an example definition of the modified Blocks World domain in PCPR logic. Preconditions and postconditions will be represented by states that represent a first-order domain therefore this chapter will assume that all $\phi(\overline{x}); \{Pre\} \rightsquigarrow \{Post\} \mid action$ in Γ are implicitly universally quantified as follows: $\forall \overline{x}.\phi(\overline{x}); \{Pre(\overline{x})\} \rightsquigarrow \{Post(\overline{x})\} \mid action(\overline{x})$.

One final challenge when defining the rules of PCPR logic is that it must not allow inconsistent states where a formula is mapped to $+$ and $-$ simultaneously. This chapter will ensure this by introducing a notion of a *valid* state where a state is valid if it is consistent, i.e. contains no conflicting formula maps. For example $\{(onTable\ a \mapsto +)\ *\ (onTable\ b \mapsto +)\}$ is a valid state but $\{(onTable\ a \mapsto +)\ *\ (onTable\ a \mapsto -)\}$ is not. Conjunctions in a valid state are represented by the $*$ operator as opposed to the standard list operator :: used in Chapter 4.

Given the above requirements, the last step is to formulate the rules for PCPR logic. These rules will be defined with simplicity in mind: an "ApplyAction" is

needed to be able to choose specific action definitions from the context, a composition rule is needed to compose the actions, similar to the composition rule of Hoare logic [54], and finally a frame rule [25, 48] is needed to have local reasoning on states. Additionally, the system will need two structural rules, weakening and shrink. This chapter will give definitions for these rules and will show that these rules are sound relative to the declarative semantics of PDDL.

This chapter will then formalise the PCPR logic and the soundness proof in Agda [51]. In a similar manner to Chapter 4 it will be shown that the Agda system can prove the correctness of PDDL plans. For example, given $P_{ab} \equiv \{(onTable\ a \mapsto +) * (onTable\ b \mapsto +) * (clear\ a \mapsto +) * (clear\ b \mapsto +) * (handEmpty \mapsto +)\}$ and $Q_{ab} \equiv \{(on\ a\ b \mapsto +) * (on\ Table\ b \mapsto +)\}$ as in Figure 5.1, $\Gamma_{BW} \vdash P_{ab} \rightsquigarrow G_{ab} |\ f_{ab}$, i.e. it is possible to certify that the plan f_{ab} is indeed valid in Agda.

Finally, this chapter will introduce the implementation of automation for both PCPR logic and PCP logic. This has huge practical value as without automation it is too time consuming and difficult to use the systems presented in this thesis to verify plans. This automation will be showcased by automatically verifying plans from multiple benchmark domains via type checking. The automation introduced in this chapter will also automate the extraction of verified plans to Haskell or binary files.

5.2 The PCPR Logic

This section defines the formal language and rules of PCPR logic with references to the PCP logic of Chapter 4.

5.2.1 Syntax of PCPR Logic

PCPR syntax is defined in Figure 5.3. PCPR syntax is a modification of the PCP logic syntax defined in Figure 4.1 therefore only the changes to syntax will be discussed in detail.

First-order formulas and constraints. First-order formulas are defined in the exact same way as PCP logic with the addition of *constraints* that are two specific kinds of atomic formulae that feature equality and inequality as predicate

Atomic Formulae	$Atom \ni A$	$::= R\ (t_1, \ldots\ t_n)$
Constraint	$Constraint \ni e$	$::= t = t_1 \mid t \neq t_1$
Constraint List	$CList \ni \phi, \psi$	$::= \ [] \mid e :: \phi$
Actions	$Act \ni a$	$::= \alpha\ (t_1, \ldots\ t_n)$
Plan	$Plan \ni f, f_1, f_2$	$::= shrink \mid a \mid f; f_1$
Polarities	$Polarity \ni z$	$::= \ + \mid -$
State	$State \ni S$	$::= emp \mid A \mapsto z \mid S * S$
(Planning) Context	$\Gamma \ni \gamma$	$::= \phi(\overline{x}); \{P(\overline{x})\} \rightsquigarrow \{Q(\overline{x})\} \mid \alpha\ \overline{x}$
Specification	$Specification \ni G$	$::= \Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$
Preconditions	$Preconditions \ni P$	$::= S$
Effects	$Effects \ni Q$	$::= S$
Goals	$Goals \ni G$	$::= S$

Figure 5.3: *The syntax of PCPR logic. The bold text indicates modifications to the syntax in comparison to PCP logic.*

symbols. Constraints can be used to reason about the equality of constants in the preconditions of an action as shown in Figure 5.1.

States in PCPR logic can be represented as a list of formula maps, just as in PCP logic, however, a new syntax for states is given to indicate that the states in this system are valid states. A *state* can be given by an empty state, a formula map or a conjunction of such maps (denoted by $*$). This chapter will only work with valid states.

Definition 28 (Valid State)**.** A state $(A \mapsto z * S)$ is *valid* if A does not occur in S and S is a valid state.

Actions are still action names applied to lists of terms however the context has been updated to add constraints. Constraints are added to the context as a *Contraint List* that is simply a list of constraints. A list is needed as there may be any number of constraints in the preconditions of actions (including zero).

The syntax of a plan has been updated. A plan is a sequence of actions represented as a binary tree with actions or *shrink* as leaves. The constructor *shrink* can therefore be used in a plan instead of an action. The function of *shrink* is similar to *halt* in Chapter 4 however it is no longer forced to be at the end of a plan as discussed later in the chapter. In practical use of the system, many plans will simply be represented in the exact same way as Chapter 4 as a sequence of actions with shrink at the end of the plan.

A plan specification in PCPR logic also represents the planning problem as a specification of the form (compare with Figure 4.1):

$$\Gamma \vdash \{P\} \rightsquigarrow \{G\} \mid f$$

In this chapter both contexts and the specification contain curly braces around states and use | instead of : to indicate the difference between PCP logic and PCPR logic.

Remark on Notation 4

In all examples in this chapter, the following shorthand notation is used:

$$R\ t \mapsto z * R\ t_1 \mapsto z \equiv R\ t, t_1 \mapsto z$$

For example, $(onTable\ a, b \mapsto +)$ *will be written instead of* $(onTable\ a \mapsto +) * (onTable\ b \mapsto +)$*. To emphasise that a formula map binds stronger than* $*$*, parentheses will be put around formula maps in all examples. But this chapter will omit the parentheses in the formal grammar, to keep the notation simple.*

5.2.2 Subtyping (order on states)

PCPR logic has the same subtyping relation and override operator as PCPlogic as defined in Figure 4.2 and Definition 4.4.1. In this chapter, subtyping is useful to compare states or to decide whether states are equal. Two states P and Q are considered equal if $P <: Q$ *and* $Q <: P$. Subtyping is a natural way to compare states as the comparison is order independent. This section will now define lemmas about subtyping and the override operator that will further elucidate their properties and will be useful in proving soundness later.

Subtyping is both reflexive and transitive, i.e. it is a *pre-order*.

Lemma 9 (Subtyping is Preorder)
Given states P, Q, S, it is obtained:

- *(reflexivity) $P <: P$;*
- *(transitivity) $P <: Q$ and $Q <: S$ implies $P <: S$.*

The following lemmas summarise the properties of the subtyping relation and the override operator.

Lemma 10 (Order of Subtyping)
Given an atom A and states P and Q, if $A \notin Q$ and $Q <: P$ then $A \notin P$.

Lemma 11 (Monotonicity of Subtype Expansion)
Given states P and Q and a formula map $A \mapsto z$, if $Q <: P$ then $A \mapsto z * Q <: P$.

Lemma 12 (Post-condition Override)
$(P \sqcup Q) <: Q$ holds for all states P and Q.

Lemma 13 (Monotonicity of Override)
Given a polarity z, an atom A, states P and Q, if $A \notin Q$ then $A \mapsto z \in (A \mapsto z * P) \sqcup Q$.

5.2.3 Evaluation of Constraint Lists

This section will define an evaluation function for constraint lists. This function takes a list of constraints and recurses through them checking that they are true. If a constraint is not true, $\perp$ is returned; otherwise, the empty list case will be reached and $\top$ will be returned. The notation $t \equiv t_1$ denotes syntactic equivalence between terms.

Definition 29 (Evaluation Function for Constraints)**.**

$$eval\ [] = \top$$

$$eval\ (t = t_1 :: \phi) = \boldsymbol{if}\ t \equiv t_1\ \boldsymbol{then}\ eval\ \phi\ \boldsymbol{else}\ \perp$$

$$eval\ (t \neq t_1 :: \phi) = \boldsymbol{if}\ t \equiv t_1\ \boldsymbol{then}\ \perp\ \boldsymbol{else}\ eval\ \phi$$

$$\text{Frame}\ \frac{\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid \alpha}{\Gamma \vdash \{P * A \mapsto z\} \rightsquigarrow \{Q * A \mapsto z\} \mid \alpha}$$
Where $A \notin P$ and $A \notin Q$

$$\text{Weakening}\ \frac{P' <: P \quad \Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f}{\Gamma \vdash \{P'\} \rightsquigarrow \{Q\} \mid f}$$

$$\text{Shrink}\ \frac{P <: Q}{\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid \mathit{shrink}}$$

$$\text{ApplyAction}\ \frac{\phi(\overline{x}); \{P(\overline{x})\} \rightsquigarrow \{Q(\overline{x})\} \mid \alpha(\overline{x}) \in \Gamma}{\Gamma \vdash \{P(\overline{x})[\sigma]\} \rightsquigarrow \{Q(\overline{x})[\sigma]\} \mid \alpha(\overline{x})[\sigma]}$$
Where $\phi(\overline{x})[\sigma]$ evaluates to $\top$

$$\text{Composition}\ \frac{\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f \quad \Gamma \vdash \{Q\} \rightsquigarrow \{S\} \mid f_1}{\Gamma \vdash \{P\} \rightsquigarrow \{S\} \mid f; f_1}$$

Figure 5.4: Rules of the PCP logic. The rules operate on valid states.

Example 43 (Evaluation Function for Constraints)**.** We have *eval* $[a = a, b = b] = \top$ because both a and b are equal to themselves however *eval* $[a = a, b = c] = \bot$ because b is not equal to c.

5.2.4 PCPR Logic Rules

Figure 5.4 gives the rules of the PCPR logic. This section will discuss and illustrate each rule in order, using the running Blocks World example. This chapter defined an example Blocks World context Γ_{BW}, in Figure 5.2. Assume that this is the context for all below examples.

ApplyAction checks that an action is in the context and then constructs the resultant state given by a ground substitution on that action.

Example 44 (Apply Action)**.** The grounded action *pickup_from_table* a can be applied by this rule because *pickup_from_table* is included in Γ_{BW} (cf. the first action in Figure 5.2) and has no constraints. Note that the constraint list is automatically satisfied when empty. Grounding the *pickup_from_table* action with the constant a results in the precondition and postcondition states:

$P^a \equiv \{(\mathit{handempty} \mapsto +) * (\mathit{onTable}\ a \mapsto +) * (\mathit{clear}\ a \mapsto +)\}$ and

$Q^a \equiv \{(\mathit{handEmpty} \mapsto -) * (\mathit{onTable}\ a \mapsto -) * (\mathit{holding}\ a \mapsto +) * (\mathit{clear}\ a \mapsto +)\}$.

The result of applying the ApplyAction rule on the grounded action *pickup_from_table a* is therefore:

$$\frac{(1) \in \Gamma_{BW}}{\Gamma_{BW} \vdash \{P^a\} \rightsquigarrow \{Q^a\} \qquad \mid \mathit{pickup_from_table}\ a}$$

where (1) refers to the first action in Γ_{BW}.

This is the only rule that allows for the access of planning domain definitions. Note also that this is the only rule that checks whether constraints on states are satisfied. This is possible thanks to essentially propositional reasoning implemented in planning, thus it is sufficient to check the constraints only once.

The **Composition** rule says that given an entailment $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$ it can be composed together with another entailment $\Gamma \vdash \{Q\} \rightsquigarrow \{S\} \mid f_1$ to produce $\Gamma \vdash \{P\} \rightsquigarrow \{S\} \mid f; f_1$.

Example 45 (Composition)**.** Assume that their exists an entailment $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$ where
$P \equiv (\mathit{onTable}\ a, b \mapsto +)\ *\ (\mathit{clear}\ a, b \mapsto +)\ *\ (\mathit{handEmpty} \mapsto +)$
$Q \equiv (\mathit{onTable}\ a \mapsto -)\ *\ (\mathit{onTable}\ b \mapsto +)\ *\ (\mathit{clear}\ a, b \mapsto +)\ *$
$(\mathit{handEmpty} \mapsto -)\ *\ (\mathit{holding}\ a \mapsto +)$
$f \equiv \mathit{pickup_from_table}\ a.$
Given the entailment $\Gamma \vdash \{Q\} \rightsquigarrow \{S\} \mid f_1$ where
$S \equiv (\mathit{onTable}\ a \mapsto -)\ *\ (\mathit{onTable}\ b \mapsto +)\ *\ (\mathit{clear}\ a \mapsto +)\ *\ (\mathit{clear}\ b \mapsto -)\ *\ (\mathit{handEmpty} \mapsto +)\ *\ (\mathit{holding}\ a \mapsto -)\ *\ (\mathit{on}\ a\ b \mapsto +)$
$f_1 \equiv \mathit{putdown_on_stack}\ a\ b$
it is possible to apply the composition rule because $Q \equiv Q$. The result of the application of the composition rule for these entailments is:

$$\frac{\Gamma_{BW} \vdash \{P\} \rightsquigarrow \{Q\} \mid f \qquad \Gamma_{BW} \vdash \{Q\} \rightsquigarrow \{S\} \mid f_1}{\Gamma_{BW} \vdash \{P\} \rightsquigarrow \{S\} \mid f; f_1}$$

The **Frame** rule allows the addition of formula maps to both states in an entail-

ment, provided the atom of the formula map does not already have a mapping in either state.

Example 46 (Frame)**.** Continuing the derivation in Example 44 it is possible to use the frame rule to add *onTable b* to the states P^a and Q^a because *onTable b* does not exist in either of the states. This results in the following application of the frame rule:

$$\frac{\Gamma_{BW} \vdash \{P^a\} \rightsquigarrow \{Q^a\} \mid \mathit{pickup_from_table}\ a}{\begin{array}{c}\Gamma_{BW} \vdash \{P^a * (\mathit{onTable}\ b \mapsto +)\} \rightsquigarrow \\ \{Q^a * (\mathit{onTable}\ b \mapsto +)\} \mid \mathit{pickup_from_table}\ a\end{array}}\ \textsc{Frame}$$

In PCPR logic, the frame rule is more restrictive than can be seen in other logics such as the Separation logic [12, 109], as it can only be used at an action level but not at a plan level. The following example shows the problem with the consistency of derivations, if the frame rule is applied to arbitrary judgements of the form $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$.

Example 47 (Problems with the Frame rule for complex plans)**.** Imagine that their exists the context Γ with the following action descriptions:
$\{\mathit{clear}\ x \mapsto +\} \rightsquigarrow \{(\mathit{clear}\ x \mapsto -)\ *\ (\mathit{clear}\ x_1 \mapsto +)\} \mid \alpha\ x\ x_1$
$\{\mathit{clear}\ x \mapsto -\} \rightsquigarrow \{\mathit{clear}\ x \mapsto +\} \mid \alpha\ x.$
Given this context the ApplyAction rule can be used to produce the entailments:
$\Gamma \vdash \{\mathit{clear}\ a \mapsto +\} \rightsquigarrow \{(\mathit{clear}\ a \mapsto -)\ *\ (\mathit{clear}\ b \mapsto +)\} \mid \alpha\ a\ b$
$\Gamma \vdash \{\mathit{clear}\ a \mapsto -\} \rightsquigarrow \{\mathit{clear}\ a \mapsto +\} \mid \alpha\ a.$
Since $\{(\mathit{clear}\ a \mapsto -)\ *\ (\mathit{clear}\ b \mapsto +)\} <: \{\mathit{clear}\ a \mapsto -\}$ the shrink rule can be applied to produce the entailment: $\Gamma \vdash \{(\mathit{clear}\ a \mapsto -)\ *\ (\mathit{clear}\ b \mapsto +)\} <: \{\mathit{clear}\ a \mapsto -\} \mid \mathit{shrink}$. The composition rule can now be applied twice to generate the entailment:
$\Gamma \vdash \{\mathit{clear}\ a \mapsto +\} \rightsquigarrow \{\mathit{clear}\ a \mapsto +\} \mid \alpha\ a\ b;\ \mathit{shrink};\ \alpha'\ a$ After the second application of the composition rule, the information the $\mathit{clear}\ b \mapsto +$ has been lost. If the Frame rule was not bound to single actions it would now be possible to use the Frame rule to incorrectly add $\mathit{clear}\ b \mapsto -$ to the entailment since *clear b* is not

mapped on either side of the entailment.

To be able to apply this rule on judgements involving complex plans instead of single actions, it needs to be ensured that the framed atom is not mapped in any state at any level in the plan derivation. This could be done by amending the restrictions on the frame rule or by amending the shrink rule to prevent loss of information. However, this is not done in preference of keeping the rules of inference simple.

Weakening is applied when a formula map that is wanted in the precondition P already exists in the (previously obtained) post-condition Q.

Example 48 (Weakening)**.** This example will use the *putdown_on_stack* action defined in Γ_{BW}. In Blocks World it is implied that *handEmpty* is false when *holding* any block is true and vice versa. This leads the *putdown_on_stack* action's preconditions to only contain the precondition that *holding* a block has to be true in the preconditions despite adding the information that *handEmpty* is true in the post-conditions. It is possible to use the Weakening rule to gain back the information that *handEmpty* is false in the preconditions as shown below.

$$\frac{\begin{array}{c}\Gamma_{BW} \vdash \{(\mathit{clear}\ b \mapsto +)\ *\ (\mathit{holding}\ a \mapsto +)\} \\ \rightsquigarrow \{(\mathit{clear}\ b \mapsto -)\ *\ (\mathit{holding}\ a \mapsto -)\ *\ (\mathit{on}\ a\ b \mapsto +)\ * \\ (\mathit{handEmpty} \mapsto +)\} \mid \alpha\end{array}}{\begin{array}{c}\Gamma_{BW} \vdash \{(\mathit{clear}\ b \mapsto +)\ *\ (\mathit{holding}\ a \mapsto +)\ * \\ (\mathit{handEmpty} \mapsto -)\} \rightsquigarrow \{(\mathit{clear}\ b \mapsto -)\ * \\ (\mathit{holding}\ a \mapsto -)\ *\ (\mathit{on}\ a\ b \mapsto +)\ *\ (\mathit{handEmpty} \mapsto +)\} \mid \alpha\end{array}}$$

The **Shrink** rule is a constructor that allows for the creation of an entailment given that the precondition state P is a subtype of the postcondition state Q. It is mainly used for the shrinking and reordering of the post-condition state. Shrink can appear anywhere in a plan but the main use of this rule is when there exists a goal state that is smaller than the obtained post-condition state just like the Halt rule in Chapter 4. Without this rule, it would be impossible to create an entailment for an empty plan.

Example 49 (Shrink)**.** For this example assume that there exists an entailment with the post state Q defined by:
$Q \equiv \{(clear\ b \mapsto -)\ *\ (holding\ a \mapsto -)\ *(on\ a\ b \mapsto +)\ *\ (handEmpty \mapsto +)\}$.
It would be possible to prove a goal state $\{(on\ a\ b \mapsto +)\}$ by using the Shrink and composition rule. The shrink rule can be used to create the entailment:

$$\frac{\begin{array}{c}(clear\ b \mapsto -)\ *\ (holding\ a \mapsto -)\ *(on\ a\ b \mapsto +)\ * \\ (handEmpty \mapsto +) <: (on\ a\ b \mapsto +)\end{array}}{\begin{array}{c}\Gamma \vdash \{(clear\ b \mapsto -)\ *\ (holding\ a \mapsto -)\ *(on\ a\ b \mapsto +)\ * \\ (handEmpty \mapsto +)\} \rightsquigarrow \{(on\ a\ b \mapsto +)\} \mid shrink\end{array}}$$

The composition rule can now be applied given the entailments to prove the desired goal state:

$$\frac{\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f \qquad \Gamma \vdash \{Q\} \rightsquigarrow \{(on\ a\ b \mapsto +)\} \mid shrink}{\Gamma \vdash \{P\} \rightsquigarrow \{(on\ a\ b \mapsto +)\} \mid f; shrink}$$

Frame and Weakening are structural rules, i.e. they do not change the computational properties of plans and do not change the plans syntactically. This section will be finished by stating two lemmas that explain subtyping for plans derived by structural rules. Note that all actions have unique definitions in any given context Γ. The proofs of these lemmas are given in Agda [51].

Lemma 14 (Property of structural rules (left))
If there is a derivation for $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid \alpha$ *by the rules of Figure 5.4 then it is known that:*
$\{P'(\overline{x})\} \rightsquigarrow \{Q'(\overline{x})\} \mid \alpha(\overline{x}) \in \Gamma$ *and* $P <: P'(\overline{x})[\sigma]$.

Lemma 15 (Property of structural rules (right))
If there is a derivation $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid \alpha$ *by the rules of Figure 5.4 then it known that:*
$\{P'(\overline{x})\} \rightsquigarrow \{Q'(\overline{x})\} \mid \alpha(\overline{x}) \in \Gamma$ *and* $Q <: Q'(\overline{x})[\sigma]$.

Given a planning context Γ, it will be said that a plan f is *well-typed* (for $\{P\} \rightsquigarrow \{Q\}$), if there is a derivation of $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$ by the rules of Figure 5.4.

5.3 Soundness of the PCPR Logic

This section will show that a derivation of $\Gamma \vdash \{F \downarrow_z\} \rightsquigarrow \{F_1 \downarrow_z\} \mid f$ using the rules given in Figure 5.4 guarantees that the *evaluation* of the plan f on a world that satisfies F produces a new world satisfying F_1. In order to give this proof this section will use the declarative semantics defined in Section 2.3.3, the operational semantics defined in Section 4.3.2 and the definitions of an action handler and canonical handler given in Section 4.4.2. This section will now introduce additional technical machinery as well as additional lemmas needed for the soundness proof.

Well-formed worlds have the following property :

Lemma 16 (Subtyping and Well-Formed Worlds)
If we have states P and Q, $Q <: P$ and $w \in \langle w_Q \rangle$ then $w \in \langle w_P \rangle$.

It is necessary to define a new evaluation function for the new definition of a plan given in this chapter.

Definition 30 (Evaluation Function)**.**

$$[\![\mathit{shrink}]\!]^{\delta}\ w = w$$

$$[\![a]\!]^{\delta}\ w = \delta\ a\ w$$

$$[\![f; f_1]\!]^{\delta}\ w = [\![f_1]\!]^{\delta}([\![f]\!]^{\delta}\ w)$$

The evaluation function has three cases. The shrink case just returns the world itself, as there is no computational meaning for a shrink action in evaluation. For a single action, evaluation applies the action handler to the world. For a complex plan, evaluation recurses to sub-plans.

The following property of action handlers will be used in the soundness proof:

Lemma 17 (Action Handler Strengthening)

If $(\delta\ \alpha\ w) \in \langle w_Q \rangle$ *and* $(\delta\ \alpha\ w) \in \langle w_{A \mapsto z} \rangle$ *then* $(\delta\ \alpha\ w) \in \langle w_{A \mapsto z * Q} \rangle$.

The definition of a well-formed handler needs to be updated to include constraints.

Definition 31 (Well-Formed PCPR Handler)**.** An action handler δ is *well-formed* if, given:

- a context Γ with $\phi(\overline{x}); \{P'(\overline{x})\} \rightsquigarrow \{Q(\overline{x})\} \mid \alpha(\overline{x}) \in \Gamma$,
- a state P, such that $P(x) <: P'(\overline{x})[\sigma]$ for some ground substitution σ and $\phi(\overline{x})[\sigma]$ evaluates to $\top$,
- a world $w \in \langle w_P \rangle$,

δ satisfies the following property: $(\delta\ (\alpha(\overline{x})[\sigma])\ w) \in \langle w_{P \sqcup Q(\overline{x})[\sigma]} \rangle$.

All states in PCPR logic have to be valid therefore the well-formed handler also only works with valid states. The proof for Proposition 1 is prefaced on the existence of lemmas that need the implicit consistency assumption. In PCPR logic Lemma 4 and Lemma 5 hold without the implicit consistency assumption since it is known that all states are valid. Proposition 1 therefore holds in PCPR logic without the need for the implicit consistency assumption. The proof remains the same because the same canonical handler is used and the addition of constraints to the well-formed handler definition does not affect the proof.

The next two theorems show that executing a well-typed plan f by the evaluation function $[\![f]\!]^\delta\ w$ is *sound*, for any well-formed handler δ.

Theorem 4 (Soundness of evaluation for normalized formulae)
Suppose $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$. *Then for any* $w \in \langle w_P \rangle$, *and any well-formed handler* δ, *it follows that* $[\![f]\!]^\delta\ w \in \langle w_Q \rangle$.

Proof. The proof proceeds by structural induction on the typing derivation $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$. In each of the below cases, take P, $w \in \langle w_P \rangle$, and assume $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$ was proven by application of a given rule in Figure 5.4, and each case will aim to show that $[\![f]\!]^\delta\ w \in \langle w_Q \rangle$.

Base Case 1 (ApplyAction)
Suppose there exists a proof for $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$ *by means of the rule ApplyAction. The rules premise requires that some* $\phi(\overline{x}); \{P'(\overline{x})\} \rightsquigarrow \{Q'(\overline{x})\} \mid \alpha(\overline{x}) \in \Gamma$, *and*

moreover there exists σ *s.t.* $P'(\overline{x})[\sigma] \equiv P$, $Q'(\overline{x})[\sigma] \equiv Q$, $\alpha(\overline{x})[\sigma] \equiv f$ *and* $\phi(\overline{x})[\sigma]$ *evaluates to* $\top$.

Because δ *is well-formed and* $w \in \langle w_P \rangle$, *it is obtained that:* $(\delta\ f\ w) \in \langle w_{P \sqcup Q} \rangle$. *Note that* $P <: P'(\overline{x})[\sigma]$ *because* $P \equiv P'(\overline{x})[\sigma]$ *by the conditions of the rule, and* $P <: P$ *by reflexivity of subtyping relation.*

It remains to show that $(\delta\ f\ w) \in \langle w_{P \sqcup Q} \rangle$ *implies that* $(\delta\ f\ w) \in \langle w_Q \rangle$. *It is known that* $(P \sqcup Q) <: Q$ *from Lemma 12 and can therefore it is possible to deduce* $(\delta\ a\ w) \in \langle w_Q \rangle$ *by applying Lemma 16.*

Inductive Case 1 (Weakening)

Taking P, $w \in \langle w_P \rangle$ *as before, assume* $\Gamma \vdash \{P\} \rightsquigarrow \{Q\}|\ f$ *was proven by applying Weakening. By inductive hypothesis it is known that there is a proof of* $\Gamma \vdash \{P'\} \rightsquigarrow \{Q\}|\ f$, *such that* $P <: P'$ *and* $[\![f]\!]^\delta\ w' \in \langle w_Q \rangle$ *if* $w' \in \langle w_{P'} \rangle$ *for some* w'. *By Lemma 16, it is known that* $w \in \langle w_P \rangle$ *implies* $w \in \langle w_{P'} \rangle$. *And so it is obtained that* $[\![f]\!]^\delta\ w \in \langle w_Q \rangle$ *as required.*

Inductive Case 2 (Shrink)

Assume that $\Gamma \vdash \{P\} \rightsquigarrow \{Q\}|$ *shrink is obtained by application of Shrink. From the assumption it is known that* $w \in \langle w_P \rangle$ *and* $P <: Q$. *Lemma 16 is applied to get* $w \in \langle w_Q \rangle$, *as required.*

Inductive Case 3 (Composition)

Assume that $\Gamma \vdash \{P\} \rightsquigarrow \{Q\}|\ f$ *by application of Composition. By inductive hypothesis is is known that, for some* f_1 *and* f_2 *such that* $f \equiv f_1; f_2$,

- *there is a proof of* $\Gamma \vdash \{P\} \rightsquigarrow \{Q'\}|\ f_1$ *and* $[\![f_1]\!]^\delta\ w \in \langle w_{Q'} \rangle$ *if* $w \in \langle w_P \rangle$;
- *there is a proof of* $\Gamma \vdash \{Q'\} \rightsquigarrow \{Q\}|\ f_2$ *and* $[\![f_2]\!]^\delta\ w' \in \langle w_Q \rangle$ *if* $w' \in \langle w_{Q'} \rangle$;

Because $w \in \langle w_P \rangle$ *is part of the assumptions, it is obtained that* $[\![f_1]\!]^\delta\ w \in \langle w_{Q'} \rangle$. *Thus a suitable* $w' \equiv [\![f_1]\!]^\delta\ w$ *has been found. It is then obtained that* $[\![f_2]\!]^\delta([\![f_1]\!]^\delta\ w) \in \langle w_Q \rangle$. *Finally, by definition of the evaluation function, it is known that* $[\![f_1; f_2]\!]^\delta\ w = [\![f_2]\!]^\delta([\![f_1]\!]^\delta\ w)$. *And so it is obtained that* $[\![f_1; f_2]\!]^\delta\ w \in \langle w_Q \rangle$.

Inductive Case 4 (Frame)

Assume that $\Gamma \vdash \{P\} \rightsquigarrow \{Q\}|\ f$ *by application of the Frame rule, that is,* $f \equiv \alpha$, $P \equiv (P' * A \mapsto z)$, $Q \equiv (Q' * A \mapsto z)$ *(for some* α, P', Q', A *and* z*), moreover* $w \in \langle w_{A \mapsto z * P'} \rangle$, $A \notin P'$, $A \notin Q'$. *By the inductive hypothesis, it is known that there*

is a proof of $\Gamma \vdash \{P'\} \rightsquigarrow \{Q'\}|\ \alpha$ *and* $[\![\alpha]\!]^{\delta}\ w' \in \langle w_{Q'} \rangle$ *if* $w' \in \langle w_{P'} \rangle$*, for any* w'*.*

By Lemma 11 and the fact that $P' <: P'$*, it is obtained that* $A \mapsto z * P' <: P'$*. The application of Lemma 16, and the assumption* $w \in \langle w_P \rangle$ *is used to assert that* $w \in \langle w_{P'} \rangle$*, and therefore it is obtained that* $[\![\alpha]\!]^{\delta}\ w \in \langle w_{Q'} \rangle$*. It remains to show that* $[\![\alpha]\!]^{\delta}\ w \in \langle w_Q \rangle$*.*

By the definition of evaluation function, $[\![\alpha]\!]^{\delta}\ w = \delta\ \alpha\ w$*. Lemma 17 allows for the combination of two results: 1.* $(\delta\ a\ w) \in \langle w_{Q'} \rangle$ *and 2.* $(\delta\ \alpha\ w) \in \langle w_{A \mapsto z} \rangle$ *to produce the goal* $(\delta\ \alpha\ w) \in \langle w_{A \mapsto z * Q'} \rangle$ *which gives* $(\delta\ \alpha\ w) \in \langle w_Q \rangle$ *and therefore* $[\![\alpha]\!]^{\delta}\ w \in \langle w_Q \rangle$*, as required.*

It only remains to show that $(\delta\ \alpha\ w) \in \langle w_{A \mapsto z} \rangle$*. To prove this, this proof will make use of the fact that* δ *is a well-formed handler, and consider* $(\delta\ \alpha\ w)$*. Recall that*

- $w \in \langle w_{A \mapsto z * P'} \rangle$*, and,*
- *by inductive hypothesis, there is a derivation for* $\Gamma \vdash \{P'\} \rightsquigarrow \{Q'\}|\ \alpha$*. Therefore, there is* $\phi(\overline{x}); \{P''(\overline{x})\} \rightsquigarrow \{Q''(\overline{x})\}\ |\ \alpha(\overline{x})\ \in\ \Gamma$ *by Lemma 14.*
- *Also by Lemma 14, we have* $P' <: P''(\overline{x})[\sigma]$*, for some* σ*.*
- *It is known that* $\phi(\overline{x})[\sigma]$ *must evaluate to* $\top$*, or there would be no derivation for* $\Gamma \vdash \{P'\} \rightsquigarrow \{Q'\}|\ \alpha$*.*

Given these four conditions, a well-formed handler must satisfy the property: $(\delta\ \alpha\ w) \in \langle w_{(A \mapsto z * P') \sqcup Q''(\overline{x})[\sigma]} \rangle$*. The Lemma 16 can be applied to show that* $(\delta\ \alpha\ w) \in \langle w_{A \mapsto z} \rangle$*, if is is possible to show that* $(A \mapsto z * P') \sqcup Q''(\overline{x})[\sigma] <: (A \mapsto z)$*. Using Lemma 13 it can be established that* $A \mapsto z \in (A \mapsto z * P') \sqcup Q''(\overline{x})[\sigma]$ *if* $A \notin Q''(\overline{x})[\sigma]$*. To show* $A \notin Q''(\overline{x})[\sigma]$*, the proof uses Lemma 10, Lemma 15 (which gives* $Q' <: Q''(\overline{x})[\sigma]$*) and the assumption that* $A \notin Q'$*. From* $A \mapsto z \in (A \mapsto z * P') \sqcup Q''(\overline{x})[\sigma]$ *it is obtained that* $(A \mapsto z * P') \sqcup Q''(\overline{x})[\sigma] <: (A \mapsto z)$ *by using the subtyping derivation rules.*

□

The soundness of evaluation is proven in the exact same way as Theorem 3 however the proof will be given for completeness.

Theorem 5 (Soundness of Evaluation)

Suppose $\Gamma \vdash \{F_1 \downarrow_+\} \rightsquigarrow \{F_2 \downarrow_+\}|\ f$ *then for any* w *such that* $w \models_+ F_1$*, and any*

well-formed δ *it follows* $[\![f]\!]^{\delta}\ w \models_{+} F_2$.

Proof. By assumption $w \models_{+} F_1$ and by the completeness of normalisation (Theorem 1), $w \in \langle w_{F_1\downarrow_{++}} \rangle$ is obtained. Then from Theorem 4, $[\![f]\!]^{\delta}\ w \in \langle w_{F_2\downarrow_{++}} \rangle$ is obtained. Thus by the soundness of normalisation (Theorem 1), $[\![f]\!]^{\delta}\ w \models_{+} F_2$ is obtained. □

Thus if f is well-typed, it is guaranteed that the execution of f in world w is correct.

5.4 Lessons Learnt: Effects and States

The implementation and verification of PCPR Logic helped to uncover some interesting properties of PDDL. This section will cover two interesting examples discovered when formalising action definitions in PCPR logic.

The standard STRIPS subset of PDDL defines actions as a definition containing preconditions and effects. In the restricted version of PDDL used in this thesis there is not any difference between what can be expressed in preconditions and effects therefore they are both represented as states. If the systems in this thesis worked with a less restricted set of PDDL then preconditions and effects would have different representations. The normal semantic interpretation of an effect is that it is given by a list containing additions and deletions. This was reflected in the rules of PCP logic however in PCPR logic the effect list is no longer seen as a list of additions and deletions but instead as the resultant state of action application. The first problem discovered was that states are lists, and so they come with the notion of ordering, however, effects do not have an explicit ordering. The second problem was that an effect does not quite represent the resultant state of an action therefore they have to be converted to keep the list of all unaffected formula maps intact. These simple observations have surprisingly powerful consequences.

Example 50 (Ordering and consistency)**.** Consider the following *move* action that moves a vehicle from one location to another:

$$[];\left\{\begin{array}{c} isVehicle\ v \mapsto + \\ *\ isLocation\ loc1 \mapsto + \\ *\ isLocation\ loc2 \mapsto + \\ *\ isAt\ v\ loc1 \mapsto + \end{array}\right\} \rightsquigarrow \left\{\begin{array}{c} isAt\ v\ loc1 \mapsto + \\ *\ isAt\ v\ loc2 \mapsto - \end{array}\right\}$$

If the *move* action is instantiated with $v \mapsto car, (loc1, loc2) \mapsto museum$ then the resultant precondition and postcondition states would be inconsistent.

$$\left\{\begin{array}{c} isVehicle\ car \mapsto + \\ *\ isLocation\ museum \mapsto + \\ *\ isLocation\ museum \mapsto + \\ *\ isAt\ car\ museum \mapsto + \end{array}\right\} \rightsquigarrow \left\{\begin{array}{c} isAt\ car\ museum \mapsto + \\ *\ isAt\ car\ museum \mapsto - \end{array}\right\}$$

Inconsistent states are not explicitly forbidden by PDDL therefore this action could be executed. In practice, a planner or verifier will have to make a decision about what to do with actions that contain inconsistent effects. If taken as a standard effect, the result of this action will depend on the order in which the effect formulas are executed. To show this, two execution methods will be considered. The first method will simply apply the effects in the order that they are given and the second will take the more common STRIPS approach where all negatively mapped preconditions are removed first then the additions are made.

Both of these methods will be applied to an initial world:

$$w = \{isVehicle\ car, isLocation\ museum, isAt\ car\ museum\}$$

Method 1

The first step would add *isAt car museum* to the world therefore the world will remain the same as it already contained this information.

Step 1: $\{isVehicle\ car, isLocation\ museum, isAt\ car\ museum\}$

The second step will remove *isAt car museum* from the world resulting in the final state

Final State: $\{isVehicle\ car, isLocation\ museum\}$

Method 2

The first step would remove *isAt car museum* from the world.

Step 1: $\{isVehicle\ car, isLocation\ museum\}$

The second step would add *isAt car museum* to the world resulting in the final state:

Final State: $\{isVehicle\ car, isLocation\ museum, isAt\ car\ museum\}$

Different execution methods can therefore produce different results which means that different planners and verifiers can also produce different results for the same domain depending on their implementation. PDDL specifications [37] do not specify any particular ordering on effect formulas, therefore planners and verifiers have to make this decision themselves. A third option a planner or verifier could take is to throw an error if an inconsistent action is reached. This is the approach that PCPR logic takes as it is impossible to produce a typing derivation for a plan containing inconsistent actions. This thesis proposes that producing an error is the best option as it will produce consistent results across various planners as the order in which predicates are added and removed is irrelevant for consistent effects.

It could also be argued that it isn't the job of a planner or verifier to deal with inconsistent effects but instead that the domain engineer should prevent the construction of ill-formed actions in the domain. The above action could easily be fixed by adding in the precondition either an equality constraint making it impossible to try to apply the inconsistent effects in the first place. Another possible solution would be to represent effects as functions where only one mapping is allowed per predicate. Whilst this may be an assumption some planners make, this thesis proposes that it is useful to have these additional checks in the verifier as it can help identify these ill-formed domains.

Example 51 (Loss in Translation)**.** During early experiments of PCPR logic, an issue was encountered where many good plans were not being successfully validated when they were translated verbatim to PCPR logic. The reason for this is the loss of information between the "precondition" and the "effect" in the PDDL formulation.

The following example illustrates the problem:

Consider the *pickup_from_stack* action from the BlockWorld domain definition:

$$[]; \left\{ \begin{array}{c} on\ x_1\ x_2 \mapsto + \\ *\ clear\ x_1 \mapsto + \\ *\ handEmpty \mapsto + \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{c} on\ x_1\ x_2 \mapsto - \\ *\ handEmpty \mapsto - \\ *\ holding\ x_1 \mapsto + \\ *\ clear\ x_2 \mapsto + \end{array} \right\}$$

Notice that $clear\ x_1 \mapsto +$ is not mentioned in the effect list, because this fact is unaffected by the action. In PCPR logic an action defines the effect list as a resultant state (postcondition) rather than an effect therefore the information about $clear\ x_1 \mapsto +$ will simply be lost. In the PCPR logic, the frame rule can not be used to recover this information, as this formula already occurs in the precondition. As a result, some PDDL plans will fail to type check in the PCPR logic. To fix this problem, all such formula maps are added explicitly to the postconditions:

$$[]; \left\{ \begin{array}{c} on\ x_1\ x_2 \mapsto + \\ *\ clear\ x_1 \mapsto + \\ *\ handEmpty \mapsto + \end{array} \right\} \rightsquigarrow \left\{ \begin{array}{c} on\ x_1\ x_2 \mapsto - \\ *\ handEmpty \mapsto - \\ *\ holding\ x_1 \mapsto + \\ *\ clear\ x_2 \mapsto + \\ *\ clear\ x_1 \mapsto + \end{array} \right\}$$

Automation will be introduced in Section 5.6.1 which does this transformation automatically when producing domain definitions in Agda.

5.5 Agda Formalisation

The PCPR logic and all lemmas and theorems presented in this chapter are formalised in Agda, see [51]. This gives additional assurance over the correctness of the presented approach. In a similar manner to Chapter 4 an Agda module has been implemented for this chapter and also serves as a *standard library* for verifying and executing plans for PDDL problems. A large part of the Agda formalisation of the background theory is defined in Section 4.5. This section will therefore focus on the unique aspects of the formalisation such as the typing relation and validity of state.

The majority of the Agda formalisation works on states that do not necessarily need to be valid therefore it was decided not to redefine states to be valid inherently

Valid State, Typing Relation and Updated Well-Formed Handler

```
ValidState : State → Set
ValidState [] = ⊤
ValidState ((z , A) :: S) = A ∉S S × ValidState S

data _,_¦_ : Γe → (State × State) → f → Set where
  weakening : ∀{Γe P Q fs}
            → (P' : State)
            → P' <: P
            → ValidState P'
            → Γe , P ⇝ Q ¦ fs
            → Γe , P' ⇝ Q ¦ fs
  applyAction : ∀ {Γe f}
            → trueListExp (expressions (Γe f))
            → ValidState (preconditions (Γe f))
            → ValidState (postconditions (Γe f))
            → Γe , (preconditions (Γe f)) ⇝ (postconditions (Γe f)) ¦ act f
  composition : ∀ {Γe P Q R f f'}
            → Γe , P ⇝ Q ¦ f
            → Γe , Q ⇝ R ¦ f'
            → Γe , P ⇝ R ¦ join f f'
  frame : ∀ {Γe P Q f }
            → (z : Polarity)
            → (a : R)
            → a ∉S P
            → a ∉S Q
            → Γe , P ⇝ Q ¦ act f
            → Γe , ((z , a) :: P) ⇝ ((z , a) :: Q) ¦ act f
  shrink : ∀ {Γe P Q}
            → (ValidState P)
            → (ValidState Q)
            → P <: Q
            → Γe , P ⇝ Q ¦ shrink

WfHandlere : Γe → ActionHandler → Set
WfHandlere Γe σ =
  ∀{α P w} → P <: (preconditions (Γe α))
           → w ∈⟨ P ⟩
           → trueListExp (expressions (Γe α))
           → ValidState (postconditions (Γe α))
           → σ α w ∈⟨ P ⊔N (postconditions (Γe α)) ⟩
```

Figure 5.5: Snippet of Agda formalisation for PCPR Logic. Note that the ⇝ symbol is sometimes used instead of the standard product symbol , to better match the syntax given in Figure 5.3.

Proof Of Consistency for PCPR Logic

```
consistency : ∀ {Γ P Q fs} → Γ , (P , Q) ¦ fs → ValidState P × ValidState Q
consistency (weakening _ P<:P₁ ValidP P₁⇝Q) with consistency P₁⇝Q
... | ValidP₁ , ValidQ = ValidP , ValidQ
consistency (applyAction e ValidP ValidQ) = ValidP , ValidQ
consistency (composition P⇝Q Q'⇝R) with consistency Q'⇝R
                                        | consistency P⇝Q
... | ValidQ' , ValidR | ValidP , ValidQ = ValidP , ValidR
consistency (frame z a a∉P a∉Q P⇝Q) with consistency P⇝Q
... | ValidP , ValidQ = (a∉P , ValidP) , (a∉Q , ValidQ)
consistency (shrink ValidP ValidQ P<:Q) = ValidP , ValidQ
```

Figure 5.6: Agda proof that states in the PCPR logic typing relation are valid.

but instead define the notion of valid state as a property. The valid state property is defined in Agda by the ValidState function in Figure 5.4. Note that the operator $A \notin S\ S$ simply denotes that the atom A does not have a mapping in the state S. Just as in Definition 5.2.1 the ValidState function returns that a state is valid if there is a proof for each atom in the state that it does not exist in the rest of the state.

The notion of valid state is needed to ensure that all states in a given typing derivation are consistent therefore additional requirements are added to the PCPR logic rules in the Agda formalisation as shown in Figure 5.4. The ApplyAction rule adds conditions that both the precondition and postcondition states are valid. The Composition rule remains the same as it is composing derivations already containing valid states. The Frame rule knows that the transformation from $P \rightsquigarrow Q$ contains valid states and therefore only adds a condition that the new atom being framed does not already exist in either side. Both the Weakening and Shrink add the condition that the new state is valid. This culminates in a consistency proof given in Figure 5.6 that states that for any given typing derivation $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid f$ that the states P and Q are valid.

To be able to prove that the canonical handler is well-formed both Lemma 4 and Lemma 5 previously required an implicit consistency assumption to be true. As discussed in Section 5.3 the implicit consistency assumption can be removed provided the lemmas are being applied to a valid state. The canonical handler is an action handler that applies the effects of actions to a world therefore to prove the canonical handler to be well-formed it is necessary to ensure that all effects of actions are valid

states. In the Agda formalisation, this is done by adding the condition that all effects have to be valid states in the well-formed handler definition. The updated well-formed handler, given in Figure 5.4, propagates additional proofs of consistency to Theorem 4. The additional consistency proof requirements of Theorem 4 are trivial as shown by the consistency proof, therefore, will not be discussed further in this chapter. The proof of Proposition 1 remains the same except the implicit consistency assumption is no longer needed as it is known that all effects being applied are valid states. Since Proposition 1 no longer uses the implicit consistency assumption it is no longer needed or included in the formalisation. The full formalisation of all proofs is given in the repository [51].

5.6 Automation

The formalisation of PCPR logic in Agda allows for the verification, execution and extraction of AI plans just as shown for PCP logic in Section 4.5.2. The running example presented in Section 5.1.1 of this chapter aimed to derive $\Gamma_{BW} \vdash P_{ab} \rightsquigarrow G_{ab} \mid f_{ab}$. The Agda code for this derivation is given in Figure 5.8. Agda cannot automatically produce the code for this derivation therefore a user has to manually use the rules of PCPR logic in order to produce the derivation. As can be seen in Figure 5.8, producing a derivation for a simple Blocks World problem is a complex and time consuming task. It is therefore intuitive to see that producing derivations for more complex and larger problems is unfeasible. This section will introduce an implementation that automates the validation and extraction process for the STRIPS subset of PDDL domain and problem specifications. The automation aims to be friendly even to non-expert users therefore a user will only have to supply a PDDL domain specification, a PDDL problem specification and a plan to generate the necessary Agda code to verify and extract the plan.

To automate this process it is necessary to automate:

1. The conversion PDDL Domain and Problem specifications to Agda.
2. The PCPR logic proof generation for Agda, given a PDDL plan.
3. The creation of Agda code in order to extract a plan.

The following sections will mainly focus on the overall strategy for the automa-

```
(defun parseProblem (in)
  (cond
   ((eq nil (car in)) (print "done"))
   ((eq 'define (car in)) (parseProblem (cdr in)))
   ((eq :objects (caar in))
      (progn
        (setq objList
          (concatenate 'string (objectConvert (cdar in) "") " : Object"))
        (parseProblem (cdr in))))
   ((eq :init (caar in))
      (progn
        (setq init (stateConvert (cdar in) ""))
        (parseProblem (cdr in))))
   ((eq :goal (caar in))
      (progn
        (setq goal (stateConvert (cdar (cdar in)) ""))
        (parseProblem (cdr in))))
   (t (parseProblem (cdr in)))
  ))
```

Figure 5.7: Parser for PDDL Problem Spefication and Object .

tion and only discuss implementation details when necessary. The full code for the automation can be found in the repository [51].

5.6.1 PDDL to Agda Translation

PDDL uses a Lisp like syntax for the definition of domain and problem specifications therefore this thesis will use Common Lisp to implement the translation of PDDL to Agda. This choice is uncommon in comparison to modern planners and validators such as VAL [57] that are written in C++. This thesis has opted for Common Lisp due to the ease of parsing PDDL in the language which makes the implementation of PDDL to Agda translation simple and clear. The validator INVAL [46] is also written in Common Lisp due to its simplicity for working with the PDDL language. In Common Lisp reading in a PDDL specification will be read in as a list of symbols. A symbol is simply a generic data type representing names like a string however there are many predefined functions in Common Lisp that can be used with symbols such as the `eq` function that compares the equality of symbols. The list of symbols generated from reading in a PDDL specification excludes white space therefore it is simple to define a parser for specifications. A parser for PDDL problems is shown

```
P : State
P = (+ , (ontable a))
      :: (+ , (ontable b))
      :: (+ , (clear a))
      :: (+ , (clear b))
      :: (+ , (handempty)) :: []

Q : State
Q = (+ , (on a b)) :: (+ , (ontable b)) :: []

P' : State
P' = (+ , ontable b)
       :: (+ , clear b)
       :: (+ , handempty)
       :: (+ , ontable a)
       :: (+ , clear a) :: []

plan : f
plan = (join (join (act (pickup_from_table a)) (act (putdown_on_stack a b))) shrink)

Derivation : Γ₁ , P ⇝ Q ¦ plan
Derivation = weakening P (from-yes (decSub P P')) tt (composition
  (weakComp (from-yes
              (decSub ((+ , ontable b)
                       :: (+ , clear b)
                       :: (+ , clear a)
                       :: (- , handempty)
                       :: (- , ontable a)
                       :: (+ , holding a) :: [])
                         ((- , ontable a)
                           :: (+ , ontable b)
                           :: (+ , clear a)
                           :: (+ , holding a)
                           :: (+ , clear b) :: []))))
  ((frame + (ontable b) (λ z → z) (λ z → z)
    (frame + (clear b) (λ z → z) (λ z → z) (applyAction tt tt tt))))
  ((frame - (ontable a) (λ z → z) (λ z → z)
    (frame + (ontable b) (λ z → z) (λ z → z)
    (frame + (clear a) (λ z → z) (λ z → z) (applyAction tt tt tt))))))
  (shrink tt tt (from-yes (decSub ((- , ontable a)
                                   :: (+ , ontable b)
                                   :: (+ , clear a)
                                   :: (- , holding a)
                                   :: (- , clear b)
                                   :: (+ , on a b)
                                   :: (+ , handempty) :: []) Q))))
```

Figure 5.8: Agda typing derivation for Blocks World problem given in in Figure 5.1 where weakComp is a composition of the composition and weakening rules.

in Figure 5.7 that recursively looks for keywords indicating a definition and then sends the definition to the relevant functions for converting it into Agda code.

Example 52 (Translating an Object Definition)**.** In order to translate an object definition the `parseProblem` function recursively compares the current symbol read in to see if it is : `objects` indicating an object definition. Once an object definition is found the object list is passed to the `objectConvert` function that converts an object definition into a string representing the equivalent Agda definition. In the case of Figure 5.1 the object list is (`A B`) and the string produced is " a b : Object".

The general strategy for translating the definitions from PDDL domain and problem specification follows the methodology shown in Example 52. Most of the definitions can be simply translated to a string or a list of strings that represent the definition in PCPR logic syntax. Once all of the definitions are translated to strings they are then written to an Agda file. One notable exception is the translation of PDDL actions into PCPR logic syntax. This is because additional constraints sometimes need to be added to the postconditions of actions as discussed in Example 51. This is performed by comparing the list of preconditions to effects whilst they are still symbols and modifying the effect list to add any missing preconditions before it is translated to a string.

To avoid having to generate the decidability proof for predicates in Lisp, this thesis will make use of the equality tactics included in the Agda Prelude library [88] to automatically derive the proofs.

Example 53 (Agda Translation of Objects)**.** The automated translation of the objects in Blocks World will produce the following Agda code:

```
data Object : Set where
  a b : Object
-- EqObject : Eq Object
unquoteDecl EqObject = deriveEq EqObject (quote Object)
```

The two additional lines under the normal data declaration for objects use the Agda Prelude library to automatically derive the decidability proof for objects. If the library cannot automatically derive the proof then the Agda code will not type check.

It remains to discuss whether the translation of PDDL to Agda produces an Agda file that remains faithful to the original specification. This thesis argues that the conversion is directly equivalent to the specification for the supported STRIPS subset of PDDL. Conversion of *objects* is one-to-one, i.e. the list of *objects* is given as a list of constructors for the datatype *Object* that stores constants. *Predicates* are directly translated one-to-one as they are simply constructors parametrised by a number of objects. The translation converts the preconditions and effects of actions as well as the initial world and goal state to the State type defined in the Agda implementation. As shown by the normalisation function, states are equivalent to PDDL formulae. In Section 2.3 it was shown that PDDL formulae can be used to represent preconditions and effects of actions as well as the goal for a planning problem therefore this translation is equivalent. As discussed in Section 4.3.1 it is unusual to represent the initial world as a state however all positive information in the world is however retained therefore any derivation proven using this conversion remains sound.

Actions are slightly abnormal since the Agda file gives two definitions one for the name and parameters of actions and another describing their preconditions and effects. The combination of the two definitions define actions in the exact same way as PDDL excluding constraints. Constraints define equalities over objects in PDDL which is mapped one-to-one in Agda however in PDDL constraints are defined as a *precondition* whereas *constraints* are defined separately in the context in Agda. The Agda definition of constraints is equivalent as constraints are simply discharged when actions are applied as if they were preconditions.

It is unlikely that there is an error in the implementation of the translation due to the straightforward nature of the implementation through its definition in Common Lisp. Type checking the translation in Agda will give additional guarantees as well as ensure that the translation adheres to the abstract definition of domains.

5.6.2 Automation of Derivation for PCPR Logic

To automatically produce the Agda derivation for a plan that solves a given planning problem a solver was implemented in Common Lisp. A high-level overview of the solver algorithm is shown in Figure 5.9. Critically the solver had to have an in-

Given a PDDL domain D, a PDDL problem description P_D, and a plan f_1, the Agda proof script for f_1 in PCP logic is generated as follows:

1. Parse D, P_D into Lisp syntax:
   (a) Store the objects in D, initial and goal world in P_D as variables in Lisp. The initial world id stored in a variable w representing the current world.
   (b) Convert actions from D into parametrised Lisp functions that generate preconditions and post-conditions.
2. For all actions in the plan f_1:
   (a) Store w in a backup variable so that the Agda subtyping relations can be generated.
   (b) Use the relevant Lisp function (as defined in (1.b)) to generate the preconditions and postconditions of the current action.
   (c) Generate the frame axioms by comparing the preconditions of the action to w where all formula maps in w that are not in the preconditions are framed in.
   (d) Use Lisp functions to apply the action to the world and store the result in the world variable.
3. Use all stored results to generate and write Agda proof to file:
   (a) Start derivation with the Weakening rule to allow for the reordering of the initial state.
   (b) Use the Shrink rule over the rest of the derivation to shrink the result to the goal state.
   (c) The rest of the derivation proceeds by composing all actions in the plan f_1 with the relevant subtyping relations and frame axioms.
4. Type check the generated Agda file to confirm the validity of the proof.

Figure 5.9: Overview of the code that automatically generates PCP proofs in Agda given PDDL Domain and plan. The code is given in [51].

ternal representation of PDDL domains, problems and plans as well as the ability to evaluate these plans to generate the relevant frame axioms. If the solver terminates with an Agda derivation generated then the Lisp solver has internally decided that the plan is valid for the given problem. Type checking the produced derivation will provide the additional formal guarantee that the plan is valid.

Both the translation of PDDL domains and problems and the production of the derivation have now been automated therefore simply combining these and type checking the result in Agda fully automates the validation of PDDL plans. This thesis refers to the resultant tool as a *plan verifier*. It is assumed that plans are received from external tools and then translated into Lisp syntax manually.

A plan is simply a list of symbols.

Decidability for PCP Logic and Subtyping

```
_<:?_ : Decidable _<:_
xs <:? [] = yes ([]<: xs)
xs <:? (y :: ys) with y ∈? xs | xs <:? ys
... | yes y∈xs | yes xs<:ys = yes (atom<: y∈xs xs<:ys)
... | no y∉xs | _  = no λ { (atom<: x x₁) → y∉xs x}
... | _ | no xs≮ys = no λ { (atom<: x x₁) → xs≮ys x₁}

solver : (Γ : Context) (f : Plan) (P Q : State) → Dec (Γ ⊢ P ⇝ Q : f)
solver Γ (α :: f) P Q with P <:? (preconditions (Γ α))
... | no ¬p = no λ { (seq P<:Pre d) → ¬p P<:Pre}
... | yes p with solver Γ f (P ⊔N (effects (Γ α))) Q
... | no ¬p = no (λ { (seq P<:Pre d) → ¬p d})
... | yes p₁ = yes (seq p p₁)
solver Γ halt P Q with P <:? Q
... | no ¬p = no λ { (halt P<:Q) → ¬p P<:Q}
... | yes p = yes (halt p)
```

Figure 5.10: Proof of decidability for PCP logic and subtyping.

Example PCP Logic Derivation Using Decidability

```
Derivation : Γ ⊢ P ⇝ Q : plan
Derivation = from-yes (solver Γ plan P Q)
```

Figure 5.11: Example PCP logic derivation utilising the solver.

Example 54 (Lisp Plan)**.** The Blocks World plan

$$f_{ab} = ((pickup_from_table\ a)\ ;\ (putdown_on_stack\ a\ b)).$$

is represented in Lisp as:

```
(setq plan '((pickup_from_table 'a) (putdown_on_stack 'a 'b))
```

Section 6.5 contains the methodology for this process and actual implementation [51] contains further instructions and examples.

5.6.3 Automation of the Derivation for PCP Logic

It is also possible to automatically produce a derivation for PCP Logic that is defined in Chapter 4. Since the rules of PCP logic are more simple than PCPR logic the automation of the derivation can be implemented in Agda rather than Lisp. The

automation for PCP logic derivation is given by defining Agda function solver that is a proof of decidability for PCP logic. As discussed in Section 2.5.2, a proof of decidability defines a decidable procedure that gives proof evidence for the result. The solver function is defined in Figure 5.10 and is a major improvement over the automation strategy given for PCPR logic. In the PCPR logic, it is known that any derivation that type checks is valid however there is no proof that the solver will produce a derivation for a valid plan. By proving the decidability of PCP logic it is guaranteed that if it is possible to produce a derivation for a given plan then the derivation will be produced. On top of this in the case that a derivation fails there is proof evidence for why the derivation failed and therefore why the plan is not valid.

In order to define the solver function a proof of decidability for subtyping is needed. The proof of decidability for subtyping is given by the function _<:?_ in Figure 5.10. To prove the decidability of PCP logic the solver considers two cases. The first case considered is that there is a plan $\alpha :: f$. In this case the solver checks to see if the current state P is a subtype of the preconditions of the action *alpha*. If the preconditions are not a subtype of P then the derivation cannot be produced. If the preconditions are a subtype of P then the postconditions of α are applied to P and then the solver is recursively applied to the rest of the plan f. If the solver is successful then the derivation can be produced using the *seq* rule. In the case that the solver is not successful the derivation fails. The second case is that the plan is at the halting state *halt*. In this case the solver checks to see that the final state P is a subtype of the goal state Q and if so the halt rule is applied and the derivation is complete. If P is not a subtype of Q then the derivation fails.

To implement a plan verifier for PCP logic it is necessary to automatically create an Agda file that will show that a plan is valid upon type checking. This can be done by using Lisp to write an Agda file including the derivation given in Figure 5.11. This code is rather generic, given the supporting solver infrastructure (of Figure 5.11. The function from-yes assumes that a decidable function will succeed. This means that Agda will check when type checking a file that a decidable function applied to from-yes will actually succeed. The derivation given in Figure 5.11 will therefore only type check when a plan is valid.

PDDL Domain / Problem	Number of Predicates	Number of Actions	Number of Objects	Predicates in Initial State	Predicates in Goal
Blocks World (A)	5	4	6	13	5
Blocks World (B)	5	4	7	9	6
Blocks World (C)	5	4	11	15	10
Logistics (A)	9	6	15	30	4
Logistics (B)	9	6	22	44	7
Logistics (C)	9	6	29	58	11
Satellite (A)	11	5	12	17	3
Satellite (B)	11	5	18	35	8
Satellite (C)	11	5	42	85	14
MPrime	12	4	25	77	1

Table 5.1: Comparison table containing information about the various domains and problems used for evaluation testing. See Appendix B for the PDDL domain and problem definitions.

5.6.4 Evaluation of the Library Performance

This chapter has now described the implementation of a plan verifier for both PCPR logic and PCP logic. Results for this section have been obtained by evaluating the plan verifiers over a few benchmark PDDL domains: BlocksWorld, Logistics, Satellite and Mprime [5] with accompanying definitions in Appendix B. All domains use the STRIPS requirement with Mprime also requiring equality and negative preconditions. For each domain, three problem descriptions of increasing complexity were used indicated by (A),(B) or (C). A comparison of the different domains and problems is given in Table 5.1. MPrime only has one test as it cannot be compared with PCP logic due to the logic not supporting equality constraints. The results of the evaluation are presented in Table 5.2. The results for all of these examples were generated automatically by supplying a plan and the PDDL domain and problem descriptions to the plan verifiers. The plans were all automatically produced through the *Planning.Domains* planner [80].

The evaluation shows that for smaller plans the PCP logic and PCPR logic system have similar type checking times however the PCP logic system scales far better than the PCPR logic system. Specifically, the PCPR logic system seems to explode in type checking time after a certain length of plan as can be seen in the type checking time difference between Logistics (A) and (B). PCPR logic unfortunately does not scale to larger problems as it fails to type check both Logistics (C) and Satellite (C). This is due to the type checking exceeding the maximum heap size

		PCPR Logic		PCP Logic
PDDL Domain / Problem	Plan Length (number of actions)	Lisp Solver Time (seconds)	Type checking Time (seconds)	Type checking Time (seconds)
Blocks World (A)	10	0.017	6.7	6.17
Blocks World (B)	20	0.025	8.22	6.28
Blocks World (C)	34	0.047	13.59	6.59
Logistics (A)	24	0.047	14.47	8.02
Logistics (B)	58	0.142	47.88	9.25
Logistics (C)	74	0.249	*	11.27
Satellite (A)	9	0.019	8.43	7.9
Satellite (B)	21	0.049	15.56	8.59
Satellite (C)	44	0.288	*	16.71
MPrime	11	0.057	26.49	n/a

Table 5.2: Evaluation of the plan verifier. A * indicates a test that failed due to the maximum heap size being exceeded after 800 seconds. All tests were performed on an Intel Core i5-12600K processor with 16GB of RAM. See Appendix B for the PDDL domain and problem definitions.

after 800 seconds. Fortunately, the PCP logic scales well and manages to type check all problems. In particular, there is only a 3 second difference in the type checking time between Logistics (A) and Logistics (C). PCPR logic also has to generate proofs using the lisp solver however this adds a negligible amount of time. Whilst the PCP logic system does scale with increasing plan complexity the type checking time is still far slower than traditional validators. For example, VAL [57] validates the slowest problem Satellite (C) in 0.003 seconds in comparison to the 18 seconds taken by PCP logic to type check the problem.

The type checking time for the PCPR logic system is worrying, however there is plenty of room for improving it. The given type checking times reflect the fact that the Lisp script generates excessively long Agda proofs. This happens because the Lisp script creates a derivation that frames all possible formulas in when generating proofs (see Figure 5.9). For example, the Logistics (A) domain and problem contain a derivation with nearly 900 rule applications. Out of the almost 900 applications, there are 838 Frame rules, 24 ApplyActions, 24 Weakenings, 23 Compositions and 1 shrink. Many of these frame rule applications are redundant and could be removed in order to speed up type checking time.

When comparing the type checking time versus plan length another outlier for

```
showObject : Object → String
showObject a = "a "
showObject b = "b "

showPredicate : Predicate → String
showPredicate (clear x) = "clear " ++ showObject x
showPredicate (on x y) = "on " ++ showObject x ++ showObject y
showPredicate (ontable x) = "ontable " ++ showObject x
showPredicate (holding x) = "holding " ++ showObject x
showPredicate handempty = "handempty "

showWorld : World → String
showWorld [] = "emp"
showWorld (x :: xs) = showPredicate x ++ " * " ++ (showWorld xs)

world-eval : World
world-eval = ⟦ plan ⟧ (canonical-σ Γ₁) (σα P [])

main : IO ⊤
main = putStrLn (showWorld world-eval)
```

Figure 5.12: Automatically Generated Agda Code for Plan Extraction

the PCPR logic system is the MPrime domain. The reason the type checking time is so long for this example is also because of excessive frame rule applications. There are also nearly 900 rule applications in the derivation for the MPrime problem and these applications contain 865 frame rules, 11 ApplyActions, 11 Weakenings, 10 compositions and 1 shrink. Once again most of these frame rule applications are redundant.

Whilst it is possible to reduce the size of the proofs produced by PCPR logic this will only further increase the complexity of the solver generated in Lisp. The simplicity of the current algorithm gives confidence that the solver for PCPR logic will find a proof. This however does not supply the same levels of confidence that the decidability proof of PCP logic brings. Constructing a similar decidability proof for PCPR logic would be far more difficult and has not been completed for this thesis. This leads this thesis to conclude that the automation of plan validation is better left to the PCP logic system.

5.6.5 Extraction of Plans to Executable Code

Just as with PCP logic it is also possible in PCPR logic to extract code that evaluates a plan on the given initial world. The code needed for the extraction of the Blocks World running example is given in Figure 5.12. In order to print out the resultant world to a user, it is necessary to define functions that convert a world to a string. To convert a world to a string there needs to be functions that map strings to both the object and predicate data definitions. This is a time consuming task manually however it is easy to produce these functions automatically using the Lisp parser defined in this chapter. The resultant extracted code will run the main function that evaluates a plan using the world-eval function and returns the resultant world as a string to the user. The extraction is fully automated in the repository [51] and the resultant code runs in milliseconds.

5.6.6 Applicability to PDDL Domains

The inference systems presented in this thesis have been tested on multiple benchmark domains from the downward-benchmark repository [5] as shown in Table 5.2. It is difficult to accurately evaluate how applicable the subset of PDDL formalised in this thesis is to PDDL domains in general as there is no centralised database containing all PDDL domains. In lieu of this, this thesis will use two benchmark repositories called downward-benchmarks and pddl-instances [5, 103] to evaluate the applicability of the presented inference systems. Applicability will be evaluated by considering the percentage of domains in the repositories, that the inference systems presented in this thesis support.

Table 5.3 shows the applicability results. These results were calculated by iterating through all the domains within these repositories and checking if their requirements were a subset of the supported requirements. If the requirements field was empty then the domain was ignored. The results also consider the applicability of the inference systems for the `:typing`, `:action-costs` and `durative-actions` requirements.

The `:typing` requirement is already possible to encode without changing the inference rules. This is demonstrated in Chapter 6 which contains domains that use the typing extension. Automation for `:typing` has been left for future work however

Benchmark Repository	Supported Requirements	Supported plus typing	Supported plus typing and action-costs	Supported plus typing and durative-actions	Total Domains
downard-benchmarks	19%	37%	82%	37%	650
pddl-instances	17%	23%	45%	55%	1836

Table 5.3: Evaluation of the percentage of domains supported by the inference systems presented in this thesis. The supported requirements are : `equality`, : `strips` and : `negative − preconditions`.

this does not prevent the use of typed domains. This is because typing in PDDL has mainly a syntactic meaning so types can be compiled into predicates [49]. It would therefore be possible to use typed PDDL domains automatically as long as they have been compiled into predicates.

The : `action − costs` and : `durative − actions` requirements are considered as they represent the two biggest applicability improvements for the inference systems. The : `action − costs` requirement adds costs to the effects of actions and an additional goal condition to minimise the action cost for the plan. Adding action costs would be a minimal change to the inference rules similar to adding equality constraints however to ensure that the minimal action cost has been reached the inference systems would have to be able to meta-reason over the set of correct plans. This meta-reasoning would be a major extension to the system and may require interfacing with SAT solvers. The : `durative − actions` requirement adds temporal reasoning to the domains. It was also assumed that the : `numeric − fluents` extension would be added along with the : `durative − actions` which allows for numeric function definitions in the domain. Adding the : `durative − actions` requirement would require a significant change to the inference rules of the inference systems in this thesis.

The results in Table 5.3 show that the inference systems are already compatible with a large number of domains. For future work, extending the inference systems to support requirements : `action − costs` or : `durative − actions` have been identified to best improve applicability. In the benchmark domains, there is no overlap between these two requirements, therefore pursuing either of these extensions would be worth further investigation.

5.7 Discussion

This chapter has introduced PCPR logic, a novel resource logic for the verification of AI plans, and proven its soundness. Critically, PCPR logic addresses the issue of inconsistent states that affected PCP logic in Chapter 4. It was possible to accomplish this through the use of resource semantics and the frame rule. The implementation of the frame rule for local reasoning introduced an alternative view on the verification of AI planning which further addresses Objective 1. Since both the rules of PCP logic and PCPR logic are proven sound relative to the computational semantics defined in this thesis it shows that both the STRIPS and the frame approach to local reasoning are equally sound as expected. The last theoretical contribution was to show how the systems presented in this thesis are easily extendable to capture more of the PDDL syntax through the introduction of constraints.

From the practical perspective, this chapter formalised PCPR logic as an Agda library. On top of this, the parsing, verifying and extracting of AI planning problems from PDDL specifications and plans was automated. The automation introduced fully realises the practical significance of the Curry-Howard correspondence as it allowed for the systems to be used on real benchmark domains to verify and extract plans without any need for expert knowledge in Agda. The introduced automation therefore achieves Objective 5 and further addresses Objective 2.

The automated inference systems have been used to test domains however no issues with planners have yet to be found for the benchmark domains that were tested in this thesis. This chapter discusses potential issues with state consistency in Section 5.4 that lead to undefined behaviour. In this case, planners can produce invalid plans that are proven false in PCPR logic. This is because domains that produce inconsistent states are considered ill-formed, therefore some planners assume that they will not receive these domains. By modifying benchmark domains so that they intentionally produce inconsistent states there have therefore been cases where planners produce incorrect plans. The inference systems produced by this thesis will prove that these plans produce inconsistent states and therefore can be used to not only verify plans but also as a sanity check for domains. It would be useful to perform a larger-scale verification effort using the inference systems in this thesis to see if any other issues with planners can be found.

Compared to the PCP logic the PCPR logic is stronger theoretically. This is because PCPR logic embeds consistency directly into the system through its rules. This has two advantages. The first is that it is impossible to derive proofs that contain inconsistent states and the second is that type errors for inconsistency will show exactly where and why there is an inconsistent state. The PCPR logic has also been extended to support PDDL constraints. On the other hand, the evaluation of the automation introduced in this chapter has shown that PCPR logic is far less practical to use for verification than PCP logic. Even though the frame rule seems like a natural way to perform local reasoning, it unfortunately overcomplicates the derivations which leads to two disadvantages. The first is that the automated proofs produced for PCPR logic are far slower to type check and do not scale as well as PCP logic. The second is that to produce the derivations for PCPR logic an external solver was required whereas PCP logic was automated by being shown to be decidable. The decidability of PCP logic guarantees not only that a plan will be definitely verified if it is possible to verify but also that in the case where a plan fails to verify that the PCP logic system will return a proof of why it has failed.

It should be noted that there has not been a conclusive investigation into whether proving the decidability of PCPR logic is possible however it is much more complicated to prove than PCP logic. This can be seen by comparing the simple decidability proof given in Figure 5.10 to the algorithm in Figure 5.9. It can also be shown by considering some of the cases of the decidability proof for PCPR logic. For example, to prove decidability for entailment over an action $\Gamma \vdash \{P\} \rightsquigarrow \{Q\} \mid \alpha$ any possible combination of the *Weakening*, *Frame* and *ApplyAction* rules have to be considered in order to show decidability. In comparison, the similar case in PCP logic only needs to consider the *Seq* rule. Whilst theoretically, it would be useful to further investigate the possibility of proving the decidability of PCPR logic this investigation has been left to future work.

This chapter concludes that PCP logic is more useful from a practical perspective. From the experience gained in developing PCPR logic this thesis will therefore adapt the PCP logic system in Chapter 6 to ensure consistency rather than continue with using PCPR logic. Due to the results from this chapter, Chapter 6 will focus on utilising the benefits of formalising the logic in Agda to introduce additional ex-

pressivity and verification constraints rather than continue to develop more complex logics for verification.

5.8 Related Work

Origins of the Frame Rule. The "frame problem" that inspired the frame rule of Separation logic has origins in AI [25, 48]. One of the early methodologies to solve this problem was to declare "frame axioms" for every action explicitly. This is an inefficient way to deal with this problem as defining these frame axioms becomes infeasible the larger the system gets [25]. Since most actions in AI planning only make small local changes to the world, a more general representation would be more suitable. The logic of Bunched Implications [58, 92] and Separation Logic [93] took inspiration from this older notion of the frame problem, and introduced more abstract formalism, which is now known as a "frame rule", into the resource logics [105]. This family of logics has brought many theoretical and practical advances to modelling of complex systems, and is behind many *lightweight verification* projects [15].

This chapter has shown how the original frame problem from AI maps back to the more abstract ideas of resource logics. This is one of this chapters theoretical contributions.

Curry-Howard Approaches to Resource Logics and Linear Logic.

The PCPR Logic introduces a Curry-Howard approach to AI planning inspired by resource logic. This is in part inspired by existing applications of the Curry-Howard approach in the field as discussed in Section 3.1.3. The computational (Curry-Howard) interpretation of AI plans was not the focus of study in the above mentioned approaches, yet it plays a crucial role in the PCPR logic, from design all the way to implementation, verification and proof extraction (see Section 5.5).

Linear logic has also been used extensively to represent AI plans and has connections to Curry-Howard as discussed in Section 3.2.1. This thesis proposes that many results obtained in this chapter could be replicated in an existing Linear Logic system [18, 112]. Generally speaking, the PCPR logic can be seen as a domain specific language for AI planning. It is simpler and less expressive than Linear logic but makes up for it in simplicity and close correspondence to PDDL syntax. Trans-

formations between PDDL domain and problem descriptions to the PCP logic are straightforward since the syntax is so similar. This enables us to automate the generation of Agda proofs from PDDL plans. Notably, PCPR logic has typing rules for functions that are given directly by PDDL plans. Thus, the outputs of PDDL planners are verified as given. This close correspondence to the plans would be impossible in either of the above Curry-Howard versions of Linear logic, where proof terms tend to be much more complex. Pros and cons of domain specific versus general approaches to verification of AI plans deserves further investigation.

This chapter proposes that the DSL nature of the PCP logic and PCPR logic along with their automation algorithms will pave the way for their wider adoption as a *practical verification tool* for the AI planning community. This is something that previously proposed Linear logic approaches to AI planning did not achieve.

Chapter 6

Dependently Typed Enrichment for AI Plan Verification

6.1 Introduction

In Chapter 4 and Chapter 5 this thesis focused on the design of formal logics to verify plans produced for a subset of PDDL domains and specifications. The evaluation given in Chapter 5 showed that from a practical perspective the simpler PCP logic was the best choice for plan verification. This chapter will explore the benefits that the dependently-typed approach generally, and the Agda implementation of the PCP logic specifically, bring to the modelling and verification of AI plans.

The first contribution of this chapter is a modification of PCP logic called *PCP* Logic* that is the most optimal formalisation for the verification of plans. In order to do this, this chapter will simplify and adapt the PCP logic implementation to not only remove the need for a consistency assumption but also further improve the speed of type checking derivations. The second contribution of this chapter will be to demonstrate how the Agda implementation can enable users to reason about and verify more general and abstract properties of plans, and also provides a more holistic programming language infrastructure for modelling plan execution. To accomplish this, this chapter will introduce additional verification properties into the Agda system that are not part of the original domain. These properties will be referred to as extrinsic properties and will include properties that are inexpressible in PDDL such as higher-order properties.

6.1.1 Verifying Extrinsic Properties

The work presented up until this chapter has focused on the verification of plans in the traditional sense. This includes the verification of properties such as the satisfaction of preconditions and goal states for plans. These kinds of properties are inherent in the domain definition and will therefore also have to be ensured by a planner in order to produce a plan. This thesis will therefore refer to these properties as *intrinsic* properties. Although AI planners are mature tools from the algorithmic and engineering points of view, they have limitations in the type of properties they can ensure. For example, PDDL prohibits the use of higher-order properties and recursion so that planners can have decidable and efficient automated search. This chapter will extend the Agda system to reason about plan properties that the planner itself either *cannot* or *should not* reason about. These properties will be referred to as *extrinsic* properties.

This chapter identifies three main classes of extrinsic properties:

1. **Inexpressible properties** - these are properties that cannot be expressed in the declarative specification language of the planner, for example, because they involve high-order functions or an unbounded state. A good example of such a property is that the plan produced is *fair*. Fairness typically involves universally quantifying over all the agents in the problem and keeping track of and comparing state. As discussed in Section 6.3.3, such global properties are typically impossible to express as pre-conditions of individual actions in the baseline versions of planning languages such as PDDL [78]. In contrast, it is easy to express and reason about them in a dependently-typed language like Agda.
2. **Unavailable properties** - these are properties whose evaluation requires world states that are not available at planning time. A good example of such a property is the fuel consumption of a robotic agent. Although the fuel used per action can be estimated at planning time, in practice the amount of fuel required to carry out an action in the real world may depend on real-time conditions such as weather, temperature or other local conditions. Therefore, even though it cannot be checked at planning time, it is still desirable to verify that during execution the robotic agent never starts an action that it has

insufficient fuel to complete.

3. **Probable properties** - finally these are properties which plans produced by the planner have a high probability of satisfying. These types of properties are, in general, meta-level properties that do not necessarily need to be included in the planning domain and problem. Fairness is also an example of this type of property. For example, suppose that the planner is assigning jobs to workers and it needs to be ensured that the set of assignments does not exhibit gender bias. By default, if the planner does not have access to gender information it would be expected that the vast majority of plans produced are fair. Nonetheless, it is possible that in certain circumstances some other part of the domain may act as a proxy for gender and result in plans that are biased. Such problems are widely known in data science and machine learning [90]. Even if such a property can be added to the planning domain, the time complexity of planning algorithms is typically super-linear in the size of the domain. Therefore this chapter argues that one should avoid encoding it in the problem domain and only verify the property holds of any produced plans.

It should be noted that none of the *extrinsic* properties discussed are novel ideas in planning. For example, the idea of unavailable properties is already studied in semantic attachments [26]. The core contribution of adding these extrinsic properties in Agda is to provide an alternative "lightweight" way of reasoning and verifying these properties. Whilst it is possible to have planners reason about these kinds of extrinsic properties, that is not the focus of this chapter. The focus of this chapter is to provide additional verification value to planners that use the restricted subset of PDDL that PCP logic supports.

This thesis views the distinction between intrinsic and extrinsic properties as mirroring the separation between the *search* and *control* components of AI modelling. The intrinsic properties are incorporated into PDDL domains and inform the PDDL search algorithms. In contrast, the extrinsic properties are imposed on the controller and the Agda framework provides a methodology for guaranteeing that the controller does not violate them during the execution of the plan. This chapter proposes that this guarantee will be used in one of two ways: firstly, prior to execu-

tion, a plan can be run in a simulated environment to check for extrinsic property violations. Secondly, during the live execution of a plan it can be used as a form of run-time checking that provides a *formally verified* guarantee that the controller will never perform an action that violates the extrinsic properties. Note that this chapter will not address what the controller should do in response to an imminent violation of an extrinsic property however it will discuss how potential methods to provide feedback about the violation to the planner in Section 6.6.

6.1.2 The Technical Approach

The first contribution of this chapter is the introduction of PCP* Logic that is a modified version of PCP logic that eliminates the consistency issues of the original formalisation. This will be accomplished by modifying the typing derivation as well as showing how these modifications affect the soundness proofs. An evaluation of the type checking time of PCP* logic will also be given.

The second and main technical contribution of this chapter is the use of *action handlers* as a means of integrating rich extrinsic properties expressed in the proof and programming environment of Agda in PCP* logic formalisation. In this chapter, action handlers become the central tool for building richer programs and proof infrastructure around the plans produced by AI planners. In particular, this chapter will utilise dependent types to enrich the handlers with additional constraints representing extrinsic properties that should hold during plan execution. As a result, *enriched action handlers* will be obtained in which it is possible to incorporate additional safety, security, fairness or other checks of arbitrary complexity which are then formally verified by Agda. Crucially, these extrinsic properties can be expressed and verified without altering either the native PDDL problem domain or its formal semantics. Notably, the richer properties that this chapter will define and prove are specified at the type level. From this point of view, this chapter will present a non-trivial exercise in dependently-typed programming.

This chapter also serves as a prototype for embedding existing automated reasoning tools within dependently-typed modelling environments. For example, it is possible to perform higher-order reasoning (in Agda's interactive style) on top of the first-order proof search already performed by the AI planner. This substan-

$$\text{Halt}\ \frac{w \in \langle S \rangle}{\Gamma \vdash w \rightsquigarrow S\ :\ \text{halt}} \qquad \text{Seq}\ \frac{\begin{array}{c} w \in \langle P(\overline{x})[\sigma] \rangle \quad P(\overline{x}) \rightsquigarrow Q(\overline{x})\ :\ \alpha(\overline{x})\ \in\ \Gamma \\ \Gamma \vdash (\delta_\alpha\ Q(\overline{x})[\sigma])\ w \rightsquigarrow S\ :\ f \end{array}}{\begin{array}{l} \Gamma \vdash : w \rightsquigarrow S\ :\ \alpha(\overline{x})[\sigma]; f \\ \text{where } Q(\overline{x})[\sigma] \text{ is a valid state} \end{array}}$$

Figure 6.1: PCP* logic rules.

tially extends the modelling power of the AI planners, as in Agda it is possible to encode many properties that PDDL cannot encode. This includes function definitions, universal and existential quantification, action dependencies and higher-order quantification. This chapter will give evidence that this approach could play an important role in verification of complex AI applications.

6.2 PCP* Logic - Validating Intrinsic Properties of a Plan

This section presents PCP* logic, a modification of the PCP logic defined in Chapter 4. The major change in the modified formalisation is that the inference rules will no longer represent the world state as a state but instead as a world. Recall that a world is simply a list of all true atomic predicates and a state is a list containing mappings of predicates to polarities. The intuition for the change in representation of the world state is that a world cannot possibly be inconsistent as it only contains true formula. The inference rules for PCP* logic are given in Figure 6.1 and use the grammar defined in Figure 4.1. This change in representation has major consequences to the definition of the inference rules, well-formed handler and proof of soundness of normalised formula as shown in Figure 6.2. Note that in Figure 6.2 the δ_α function has been renamed to updateWorld.

By defining the world state as a state in the inference rules of PCP logic, the world state, preconditions and effects of actions all had the same type. This allowed for satisfaction to be defined using the subtyping relation in the inference rules. In comparison PCP* logic uses the notion of well-formed worlds, given in Definition 22, to define the satisfaction of preconditions and goal states. In the case of the *Halt* rule the definition is the exact same as Figure 4.3 except that it uses worlds and the well-formed world relation. The *Seq* rule is more interesting as changing the

```
_∈⟨_⟩ : World → State → Set
w ∈⟨ S ⟩ = (∀ a → (+ , a) ∈ S → a ∈ w) ×
           (∀ a → (- , a) ∈ S → a ∉ w)

ValidS : State → Set
ValidS [] = ⊤
ValidS ((z , r) :: S) = r ∉S S × ValidS S

data _⊢_⇝_:_ : Context → World → State → Plan → Set where
  halt : ∀ {Γ currentWorld goalState}
       → currentWorld ∈⟨ goalState ⟩
       → Γ ⊢ currentWorld ⇝ goalState : halt
  seq : ∀ {α currentWorld goalState Γ f}
       → currentWorld ∈⟨ preconditions (Γ α) ⟩
       → ValidS (effects (Γ α))
       → Γ ⊢ updateWorld (effects (Γ α)) currentWorld ⇝ goalState : f
       → Γ ⊢ currentWorld ⇝ goalState : (α :: f)

WfHandler : Context → ActionHandler → Set
WfHandler Γ δ =
  ∀{α w} → w ∈⟨ preconditions (Γ α) ⟩
         → ValidS (effects (Γ α))
         → δ α w ≡ updateWorld (effects (Γ α)) w

sound : ∀{δ w Γ f Q}
      → WfHandler Γ δ
      → Γ ⊢ f : w ⇝ Q
      → execute f δ w ∈⟨ Q ⟩
sound wfh (halt w∈⟨Q⟩) = w∈⟨Q⟩
sound wfh (seq w∈⟨P⟩ prf tr) rewrite wfh w∈⟨P⟩ prf = sound wfh tr

wf-canonical-δ : ∀ Γ → WfHandler Γ (canonical-δ Γ)
wf-canonical-δ Γ w∈⟨P⟩ prf = refl
```

Figure 6.2: PCP* Logic definition changes in Agda

PDDL Domain / Problem	Plan Length (number of actions)	PCPR Logic Type checking Time (seconds)	PCP Logic Type checking Time (seconds)	PCP* Logic Type checking Time (seconds)
Blocks World (A)	10	6.7	6.17	2.39
Blocks World (B)	20	8.22	6.28	2.46
Blocks World (C)	34	13.59	6.59	2.65
Logistics (A)	24	14.47	8.02	4.29
Logistics (B)	58	47.88	9.25	5.42
Logistics (C)	74	*	11.27	7.26
Satellite (A)	9	8.43	7.9	4.06
Satellite (B)	21	15.56	8.59	4.85
Satellite (C)	44	*	16.71	11.99

Table 6.1: Type checking time on benchmark domains for all Agda formalisations presented in this thesis. A $*$ indicates a test that failed due to the maximum heap size being exceeded after 800 seconds. All tests were performed on an Intel Core i5-12600K processor with 16GB of RAM. See Appendix B for the PDDL domain and problem definitions.

representation of the world state and using the well-formed world relation is not enough. The previous *Seq* rule uses the override operator to apply actions however the override operator only works on states. The new *Seq* rule instead uses the δ_α function from Definition 26 to apply the action. This change means that inference rules use the same method of action application as the operational semantics.

The final restriction of the *Seq* rule is that the effects of the applied actions are valid states. It might be assumed that this restriction is necessary in the *Halt* rule as well as on the preconditions of actions, however, this is not necessary. This is because it is impossible for a world to satisfy a state that is inconsistent using the world relation, therefore the *Seq* and *Halt* rules cannot be applied to inconsistent states. Whilst it is impossible to produce an inconsistent world regardless of whether an effect is consistent, it is still necessary to restrict effects to be valid states. As discussed in Section 5.4 the application of inconsistent effects leads to undefined behaviour dependent on the execution order. However, the application of consistent effects will always result in the same world independent of order execution. In Agda the notion of valid state is given by the ValidS function given in Figure 6.2.

A well-formed handler defines the expected behaviour of an action handler in the case when the action handler is applying an action to a world where the action's preconditions are satisfied and effects of the action are consistent. The well-formed

handler defined in PCP* logic forces an action handler to apply an action using the updateWorld function. This is more restrictive than previous definitions of the well-formed handler. Since consistent effects will always result in the same world regardless of order application, this chapter proposes that there is no functional need to be able to define the application of actions differently. It should be noted that action handlers will still be free to independently deal with the cases where an action's preconditions are not met. The modification of the well-formed handler definition makes the proof of soundness of normalisation as well as the proof that the canonical handler is well-formed trivial as shown in Figure 6.2.

It is also possible to prove that the well-formed world relation is decidable and therefore also prove that PCP* logic is decidable. The automation algorithm can therefore be used to generate derivations for PCP* logic. The results are shown in Table 6.1 and show that there is a flat speedup of around 4 seconds for type checking across all examples in comparison to PCP logic. There are two trade-offs to this modification. First is that PCP* logic cannot be extended to work with open world problems as it assumes the world is closed. Second is that the definition of execution for well-formed handlers is restricted. Since this thesis only works with closed world problems, PCP* logic is the best fit as it is simple, fast and ensures consistency.

This concludes the definition of PCP* logic that validates the intrinsic properties of PDDL plans.

6.3 Running Example for Extrinsic Verification

This section will introduce the running example. This includes a PDDL definition of the domain and problem as well as its Agda encoding. The running example has a unique Agda implementation because it encodes a typed domain. This section will therefore reintroduce the formalisation of domains and problems in Agda explaining the design changes. This example will pave the way to Section 6.4 where this chapter will explain how to extend the formalisation to allow the embedding of extrinsic properties.

```
(define (domain taxi)
  (:requirements :strips :typing)
  (:types taxi location person)
  (:predicates
   (taxiIn ?obj1 - taxi
           ?l1 - location)
   (personIn ?obj1 - person
             ?l1 - location))
  (:action drive_passenger
    :parameters
      (?t1 - taxi
       ?p1 - person
       ?l1 - location
       ?l2 - location)
    :precondition
      (and
        (taxiIn ?t1 ?l1)
        (personIn ?p1 ?l1))
    :effect
      (and
        (not (taxiIn ?t1 ?l1))
        (not (personIn ?p1 ?l1))
        (taxiIn ?t1 ?l2)
        (personIn ?p1 ?l2)))
  (:action drive
    :parameters
      (?t1 - taxi
       ?l1 - location
       ?l2 - location)
    :precondition
      (taxiIn ?t1 ?l1)
    :effect
      (and
        (not (taxiIn ?t1 ?l1))
        (taxiIn ?t1 ?l2))))
```

```
(define (problem taxi)
  (:domain taxi)
  (:objects
      taxi1 taxi2 taxi3 - taxi
      person1 person2 person3 - person
      loc1 loc2 loc3 - location)
  (:init  (taxiIn taxi1 loc1)
          (taxiIn taxi2 loc2)
          (taxiIn taxi3 loc3)
          (personIn person1 loc1)
          (personIn person2 loc2)
          (personIn person3 loc3))
  (:goal (and  (taxiIn taxi1 loc2)
               (personIn person1 loc3)
               (personIn person3 loc1))))
```

Figure 6.3: *PDDL Taxi Domain and Problem.*

6.3.1 PDDL to Agda Translation

In order to illustrate extrinsic constraints this chapter defines a new taxi domain in Figure 6.3. The taxi domain is a typed domain with the types: `taxi`, `location` and `person`. There are only two predicates `taxiIn` and `personIn` that are used to represent whether a taxi or person is in a specific location respectively. There are two actions `drive` and `drive_passenger`. The `drive` action drives a taxi from one location to another and the `drive_passenger` action drives a taxi and passenger

from one location to another.

To represent a typed domain the abstract domain definition is extended to include types:

```
record Domain : Set₁ where
  field
    Type      : Set
    Action    : Set
    Predicate : Set
    Γ         : Context
    _≟p_      : DecidableEquality Predicate
```

The Taxi Planning Domain This section will now introduce the concrete code describing the taxi domain. Types are simply defined by creating a data type in Agda with the required types as constructors.

```
data Type : Set where
  taxi location person : Type
```

The introduction of types needs to then be propagated to the definitions of predicates, actions and objects. This will be performed by adding types to objects and then defining predicates and actions using these typed objects.

In PDDL, objects are first introduced implicitly as typed variables within the block that defines predicates, and only later does the planning problem give an explicit set of objects that can be used to instantiate these variables. As discussed in Section 4.5.1 this chapter will combine these two separate notions of objects and variables into a single Object data type whose constructors are indexed by Types.

The most obvious method to define objects would be to simply define them as constructors indexed by types. For example, it would be possible to define the objects with the following definition: It may be expected

```
data Object : Type → Set where
  taxi1 taxi2 taxi3 : Object taxi
  person1 person2 person3 : Object person
  loc1 loc2 loc3    : Object location
```

The above definition of objects mimics the style of definitions given in PDDL however this style of definition will not be used as it does not allow for meta reasoning about objects. For example, in Section 6.4.2 this chapter will define functions over a list of all taxis. The automatic creation of a list containing all taxis can only be defined through meta reasoning over the object data declaration. To enable meta reasoning over the objects data type, instead of defining objects by giving a list of constructors, objects will be defined by a finite number indicating the number of objects of each type. In Agda a finite number is represented as the Fin type. For example, in the object definition below the number of taxis is given by numberOfTaxis and if this number was equal to 3 then it is possible to construct taxis: taxi 0, taxi 1, taxi 2. The following object definition implements the objects given by the problem specification in Figure 6.3:

```
data Object : Type → Set where
  taxi     : Fin numberOfTaxis      → Object taxi
  location : Fin numberOfLocations → Object location
  person   : Fin numberOfPeople     → Object person
```

Now that the number of taxis is known it is possible to produce the list of all taxis by mapping over list of all finite numbers. This is defined in Agda as:

```
allTaxis : List (Object taxi)
allTaxis = Data.List.map taxi (allFin numberOfTaxis)
```

By defining objects using finite numbers instead of constructors the names of the objects given in PDDL are lost. Whilst there is no intrinsic meaning in the names of objects it can help users understand the PDDL domain better. If desired, the names can easily be recovered by defining an additional function that maps the object data type to their original names or by using pattern synonyms.

Predicates are defined as constructors over typed objects, closely mimicking the PDDL syntax in of Figure 6.3.

```
data Predicate : Set where
  taxiIn    : Object taxi     → Object location → Predicate
  personIn : Object person → Object location → Predicate
```

Actions are defined in the same way as in previous chapters however they now give typed parameters instead.

```
data Action : Set where
  drive : Object taxi → Object location
                      → Object location
                      → Action
  drivePassenger : ...
```

The context definition does not change from previous chapters however the incorporation of types into the action definition ensures that the context is defined over typed objects. The context for the drive action is given by:

```
Γ : Context
Γ (drive t1 l1 l2) =
  record {
    preconditions =
      (+ , taxiIn t1 l1) :: [] ;
    effects =
      (- , taxiIn t1 l1) ::
      (+ , taxiIn t1 l2) :: [] }
...
```

The Planning Problem As discussed in Section 6.2 the initial world is now represented using a world data type. The problem file can therefore be translated to the following definitions:

```
initialWorld : World
initialWorld =
  taxiIn (taxi 0) (location 0) ::
  taxiIn (taxi 1) (location 1) ::
  taxiIn (taxi 2) (location 2) ::
  personIn (person 0) (location 0) ::
  personIn (person 1) (location 1) ::
```

```
  personIn (person 2) (location 2) ::
  []


goalState : State
goalState =
  (+ , taxiIn (taxi 0) (location 1)) ::
  (+ , personIn (person 0) (location 2)) ::
  (+ , personIn (person 2) (location 0)) ::
  []
```

6.3.2 Verification and Evaluation of Plans

This section will now discuss how to verify and evaluate plans for typed PDDL planning problems. The running example for this chapter can be solved by the following plan:

```
plan = (drive (taxi 0) (location 0) (location 1)) ::
       (drivePassenger (taxi 2) (person 2) (location 2) (location 0)) ::
       (drivePassenger (taxi 2) (person 0) (location 0) (location 2)) ::
       halt
```

It should be noted that whilst there is full automation for PCP* logic for untyped domains there does not exist automation to translate typed PDDL domains. This is not because it is impossible to automate but rather because it is not the focus of this chapter. Even though the domains have to be manually implemented, it is still possible to use the solver to automatically verify the intrinsic properties of plans using the following code:

```
Derivation : Γ ⊢ plan : initialWorld ⇝ goalState
Derivation = from-yes (solver Γ plan initialWorld goalState)
```

It is now possible to evaluate the plan, as shown in previous chapters, by applying the execute function to the canonical handler and initial world. Note that the evaluation function has been renamed to execute in this chapter.

```
evaluationResult : World
evaluationResult = execute plan (canonical-σ Γ) initialWorld
```

As the execution is run on a validated plan the output of this function will be a world that satisfies the goal state. The resultant world is:

```
Output:
taxiIn (taxi 2) (location 2) ::
personIn (person 0) (location 2) ::
personIn (person 2) (location 0) ::
taxiIn (taxi 0) (location 1) ::
taxiIn (taxi 1) (location 1) ::
personIn (person 1) (location 1) :: []
```

As discussed in previous chapters, the derivation of the plan establishes that the plan, and therefore each action in it, is valid e.g. all preconditions are satisfied and all effects are consistent. As a consequence, it is not required for action handlers to check the validity of each action with respect to the current world before applying it. This simplifies the definition of action handlers. In the following sections, this will also enable this chapter to keep the definitions of enriched handlers that add additional constraints simple.

6.3.3 Expressivity of PDDL

This thesis has focused on the STRIPS subset of PDDL 1.2 under the closed world assumption. Specifically, this thesis has only worked with the subset of PDDL for deterministic planning domains and problems. Determinism implies the use of first-order logic without function symbols (which guarantees finite domains when defining the models). Whilst it is possible to define restricted functions in deterministic planning, the problem with functions, especially with recursive functions, is that they can make domains infinite. For example, it only takes one nullary and one unary function to generate the infinite set of natural numbers.

This chapter will utilise arbitrary functions in its development, as well as open ways to surpass the closed world assumption, by embedding the plans in a wider programming and modelling environment. By embedding these functions in Agda,

determinism is still ensured as Agda's type checker ensures that functions will terminate. Higher-order functions will also be used to express some more sophisticated properties, for example calculating the number of taxis that satisfy a certain property as discussed in Section 6.4.2. Both arbitrary and higher-order functions cannot be defined in PDDL and therefore are the subject of the syntactic (type-driven) extensions that will be introduced in this chapter. These extensions will give an alternative, lightweight and dependently-typed way to introduce some of the extensions that PDDL 2.1 and 3 introduced.

6.4 Verifying Extrinsic Properties

This thesis has introduced components (II) and (III) of its framework:

(I) **Plan generation via a PDDL planner** which takes a PDDL domain and problem definition as in Figure 6.3, and performs an automated search for plan that takes the system from the initial to the goal state.

(II) **Validation of the resulting plan via Agda** which compiles the planning domain, planning problem and plan received from the planner into a compact DSL. The plan is then validated relative to the formalised operational and declarative semantics.

The rest of this chapter will focus on a third, and perhaps the most intriguing and novel component of the framework:

(III) **Dependently-typed verification of extrinsic properties of the execution of the plan via Agda**, in which this chapter formally verifies that during the execution of the plan the controller will never execute an action that violates extrinsic properties. As discussed previously, extrinsic properties are those which are either undesirable or impossible to encode in PDDL at planning time.

To achieve this, this chapter will augment the ActionHandler type with the desired property and then ensure that the execute function has the correct type. Note that this third stage lacks the generality of stage (II) as these higher level properties are necessarily specific to the particular domain being modelled. Nonetheless, this thesis argues that it is a powerful and flexible technique for verifying properties that

cannot be checked at planning time. A notable advantage of our approach is that it is possible to verify that a property holds without altering the semantics of PDDL specification or the shape of the plans produced by the planner.

As discussed in Section 6.1.1, the verified controller could be used directly during execution. Alternatively, it could be run in a simulated environment prior to the execution of the plan by a non-verified controller. If the plan satisfies the extrinsic properties during simulation, then the environment can be monitored during execution and as long as it doesn't deviate from the simulation then the actual execution of the plan will not violate the extrinsic properties either.

6.4.1 Example 1: Fuel Consumption

The first property this chapter will introduce is a fuel property that ensures that the agent never runs out of fuel while executing a plan. Although fuel is often used in an abstract sense in functional programming to limit the number of iterations a function may perform before termination, in planning fuel often has a very real interpretation as it represents a resource (e.g. electrical energy) that an agent uses to perform actions. Typically, before an agent runs out of fuel it must return to its base and recharge.

In many domains, fuel levels cannot be taken into account by the planner in stage (I) because it is unknown what the exact fuel level will be at a given point in the plan. For example, while the plan of driving from location1 to location2 and then from location2 to location3 may be valid at planning time, it can subsequently be invalidated by unexpectedly high fuel consumption during the first leg of the trip (e.g. due to a diversion caused by road-works) that leaves the taxi unable to complete the second leg.

This chapter will now show how such a constraint can be incorporated into the Agda framework. For simplicity and pedagogical purposes, it will be assumed that all taxis share a single fuel source and that applying any action uses 1 unit of fuel. To add this property to an action handler it is first modelled using types and then it is added to the type level of the action handler. In order to reason about natural numbers at the type level this chapter defines a Fuel data type that is indexed by a natural number:

```
data Fuel : Nat → Set where
  fuel : (n : Nat) → Fuel n
```

It is now possible to enrich the definition of an action handler, by encoding the fact that applying an action reduces the fuel level from suc n to n where suc n is the successor of the natural number n.

```
FuelAwareActionHandler = ∀ {n} → Action
  → World × Fuel (suc n)
  → World × Fuel n
```

This encodes at the type level that the agent cannot begin to execute an action without having sufficient energy and that each action uses one unit of fuel.

It is now possible to define an *enriched handler*, by changing the canonical handler's return type to FuelAwareActionHandler:

```
enriched-σ : Context → FuelAwareActionHandler
enriched-σ Γ α = updateWorld' (effects (Γ α ))
```

The auxiliary function updateWorld' above is an enriched version of updateWorld that has to be defined due to the additional type requirements on the execution of actions. It takes care of checking the additional fuel constraint on the type of the action handler is satisfied during the execution of an action:

```
updateWorld' : Effect → World × Fuel (suc n)
  → World × Fuel n
updateWorld' s (w , fuel (suc n)) = updateWorld S w , fuel n
```

One advantage of defining at the type level that the fuel goes from suc n to n is that the updateWorld' function is forced to supply an energy of exactly n in its return type.

To execute plans with the FuelAwareActionHandler and check the constraints during execution, the evaluation function also has to be enriched. The evaluation function must check the fuel level and if it is suc n then the action is applied and if it is zero whilst there are still actions to apply then the plan fails in which case the function will return an error message with the failure. One simple way to implement

this is to introduce a disjunction ⊎ in the return type where the function can either return a world or an error if the execution fails. To do this, this chapter defines a OutOfFuelError data type that is constructed by passing in the current world and the failed action.

```
data OutOfFuelError : Set where
  error : Action → World → OutOfFuelError

executeWithFuel : Plan → FuelAwareActionHandler
  → World × Fuel n
  → World ⊎ OutOfFuelError
executeWithFuel halt      σ (w , _) = inj₁ w
executeWithFuel (α :: f) σ (w , fuel 0) = inj₂ (error α w)
executeWithFuel (α :: f) σ (w , fuel (suc n)) =
  executeWithFuel f σ (σ α (w , fuel (suc n)))
```

It is now possible execute the same example plan that was validated in the previous section, with the enriched (rather than canonical) handler and evaluation function:

```
evaluationResult : World ⊎ OutOfFuelError
evaluationResult = executeWithFuel plan (enriched-σ Γ)
                                   (initialWorld , (fuel 3))
```

The evaluationResult function defined above will produce the same output as given in Section 6.3.2. This is because the plan length is 3 and the amount of fuel given is also 3 therefore allowing for the application of all the actions in the plan. If the fuel level was set to 2 then an error would be returned during the application of the final action of the plan.

This section has used a simple fuel consumption example to explain the general approach of reasoning about meta-properties of already validated plans and demonstrated how enriched handlers allow for the introduction of arbitrary additional constraints at execution time without interfering with either the native (sound) semantics of PDDL, or the shape of the native plans produced by STRIPS. In a realistic system, fuel levels might be better implemented as a monadic evaluation

function, such as State n m A = Fuel n → A × Fuel m, that performs real-time measurements of the current fuel level.

6.4.2 Example 2: Fairness

This chapter will now introduce a more complex constraint, in particular, that the assignment of taxi drivers to trips exhibits no significant gender bias. Unlike the fuel consumption example, the gender information could be made available to the planner at Stage (I). However, it is infeasible and undesirable to do so for the following two reasons. Firstly, any non-trivial fairness property is unlikely to be expressible in standard PDDL syntax. Secondly and perhaps more subtly, statistically speaking it would be expected for there to be no gender bias in the output of the planner in the first place. This thesis argues that it is unnecessary to add extra complications to the planning stage to enforce something that should normally hold true most of the time. It would be possible to encode the notion of gender bias as a planner-level property at Stage (I). In this case, it is still useful to model this property in Agda to check that the planner has adhered to it.

A model gender is needed in order to encode this property in Agda. Gender is defined by the following data declaration:

```
data Gender : Set where
  male female other : Gender
```

A TripCount type will now be defined which is used to store the number of trips each gender has taken so far.

```
TripCount : Set
TripCount = Gender → ℕ
```

It should be noted that all of the code associated with the enriched handler is defined in a separate Agda module. The advantage of this choice is that it will be possible to pass in static functions representing data that will not change during the evaluation of a given domain.

```
driverGender : Object taxi → Gender
margin : Nat
```

In particular two functions driverGender and margin will be passed into the module. The driverGender function maps all taxi drivers to a Gender. The margin function represents a natural number that is used to allow for some leeway for statistical fluctuations when enforcing the fairness constraint. It is natural to include both the mapping of gender to drivers and the margin as a module parameter because their definitions vary depending on the given PDDL problem specification. For example, the driverGender definition depends on the number of taxis in the problem specification.

The total trips taken is simply calculated by adding the trips taken for all genders.

```
totalTripsTaken : TripCount → ℕ
totalTripsTaken f = _+_ (_+_ (f male) (f female)) (f other)
```

The percentage of trips assigned to a given gender is then calculated via the following function:

```
calculateGenderAssignment : Gender → TripCount → ℕ
calculateGenderAssignment g tripCount =
  (tripCount g * 100) /₀ totalTripsTaken tripCount
```

Note that the above definition uses the function $/_0$ for division that returns zero in the case that a division by zero happens to avoid errors.

To calculate a fair percentage of assignments it is necessary to first calculate the number of drivers of each gender. Note that this uses a higher order function filter which, as discussed in Section 6.3.3, is not supported by the PDDL language.

```
noGender : Gender → ℕ
noGender g =
  length (filter (λ t → decGender g (driverGender t)) allTaxis)
```

Using this it is possible to calculate the percentage of drivers of a given gender:

```
percentage : Gender → ℕ
percentage g = (noGender g * 100) /₀ totalDrivers
```

The lowest acceptable threshold that is deemed to be fair, which is controlled by the *margin* parameter, is then calculated as follows:

```
calculatelower bound : Gender → ℕ
calculatelower bound g =
  percentage g − (percentage g /₀ margin)
```

Example 55 (Margin)**.** The margin allows for users to have more leeway when calculating the accepted lower bound value that a gender assignment can be. For this example assume that the percentage of women is 20 percent. If the margin value is set to 0 then the lower bound is 20 percent because dividing by zero equals zero in the formalisation. If the margin was set to 4 then the lower bound would have 25 percent leeway over the actual percentage giving a lower bound value of 15 percent for women.

Is is now possible to express the property that a trip count is unbiased for a particular gender as follows:

```
IsFair : Gender → TripCount → Set
IsFair g f =
  calculateGenderAssignment g f ≥ calculatelower bound g
```

At this point, this section has defined a fairness property for a single gender however it is desired to enrich an action handler so that applying an action is fair for all genders, not just one. This is modelled by adding a IsFairForAll type that is the product of the $IsFair$ type for all genders.

```
IsFairForAll : TripCount → Set
IsFairForAll f = ∀ (g : Gender) → IsFair g f
```

There are still two problems with implementing the action handler just using the $IsFairForAll$ type. The first problem is that it is unreasonable to assume that after the assignment of one or just a few trips that the trips will be fairly assigned. To model this, this chapter defines the following predicate:

```
UnderMinimumTripThreshold : TripCount → Set
UnderMinimumTripThreshold tripCount =
  totalTripsTaken tripCount < totalDrivers * 10
```

The second problem is that there are two actions drive and drivePassenger and only the latter should count as a paying trip for the purpose of fairness. Again this is represented by another predicate:

```
TripAgnostic : Action → Set
TripAgnostic (drivePassenger t p1 l1 l2) = ⊥
TripAgnostic (drive t l1 l2) = ⊤
```

There are now sufficient definitions to describe the fairness property in detail, in which an action is fair if it satisfies any of the three predicates defined above:

```
data ActionPreservesFairness
  (α : Action) (tripCount : TripCount) : Set where
  underThreshold : UnderMinimumTripThreshold tripCount
    → ActionPreservesFairness α tripCount
  fairForAll : IsFairForAll tripCount
    → ActionPreservesFairness α tripCount
  agnostic : TripAgnostic α
    → ActionPreservesFairness α tripCount
```

The type of enriched action handlers that enforce this property can then be defined as follows:

```
GenderAwareActionHandler : Set
GenderAwareActionHandler =
  (α : Action)
  → {tripCount : TripCount}
  → {fair : ActionPreservesFairness α tripCount}
  → World → World
```

One thing to note is that the form of this definition is slightly different from that of the FuelAwareActionHandler defined in the previous section. Instead of adding TripCount as a part of a product with the World, it is added as an implicit argument. This is because, unlike fuel, this thesis has chosen not to enforce any type level relationships between the trip count before and after applying the action. Instead, this thesis will rely on the enriched execute function to update the trip count correctly.

The disadvantage of this approach is that one cannot enforce relationships between actions and the additional enriched state at the type level, however, the advantage of this is that it allows us to use exactly the same form for the enriched handler and canonical handler instances:

```
enriched-σ : Γ → GenderAwareActionHandler
enriched-σ Γ α = updateWorld (effects (Γ α))
```

Another advantage of working in a rich dependently-type language such as Agda is that the execution function can return error messages containing proofs in them explaining exactly why the execution of the function failed. Currently, a failed execution just returns a precise error for why an execution has failed however this thesis proposes these precise errors could be used for plan repair in future work. In this case, the error contains a proof of why the action is not fair:

```
data GenderBiasError : Set where
  notProportional : (a : Action) (f : TripCount)
    → ¬ (ActionPreservesFairness a f) → GenderBiasError
```

The enriched execute function can then be defined to check for fairness and can only execute in action if it can generate a proof that the action will not result in any gender bias:

```
execute' : Plan →
           GenderAwareActionHandler →
           TripCount →
           World →
           World ⊎ GenderBiasError
execute' halt     σ tripCount w = inj₁ w
execute' (a :: f) σ tripCount w with updateTripCount a tripCount
... | updatedTrips with ActionPreservesFairness? a updatedTrips
...     | yes fair = execute' f σ updatedTrips (σ a {fair = fair} w)
...     | no ¬fair = inj₂ (notProportional a updatedTrips ¬fair)
```

To be able to define the above execution function it is necessary to be able to automatically decide whether an action is fair. This is accomplished by proving

decidability for the ActionPreservesFairness data type. This is defined in the Agda formalisation by the ActionPreservesFairness? function. The decidability functions are defined in the accompanying Agda library [51].

6.4.3 Concrete Fairness Example

This section will showcase the fairness constraint using the running example. All examples will use a margin value of 4 and the following driverGender function:

```
driverGender : Object taxi → Gender
driverGender (taxi 0F) = male
driverGender (taxi 1F) = female
driverGender (taxi 2F) = male
```

The execute' function takes in the trip count as a function before applying the plan to the given initial world. It is therefore possible to execute a plan whilst giving data for previous trips taken. This section will give examples using three different trip assignments for the running example. For all examples the output will be generated by the following function definition with the relevant trips taken function passed in:

```
output : World ⊎ GenderBiasError
output = execute' plan (enriched-σ Γ) tripsTaken initialWorld
```

Example 56 (Under Threshold)**.** This example assumes that there were 0 trips taken before the execution. This is defined by the following function:

```
tripsTaken : Gender → ℕ
tripsTaken g = 0
```

There are three taxis therefore the minimum threshold for enforcing the fairness constraint is 30. Since the total number of trips will never exceed 3 the execution function will apply all actions in the plan as expected and return the output world given in Section 6.3.2.

Example 57 (Fair Assignments)**.** This example considers the following $\mathsf{tripsTaken}_2$ function:

$\mathsf{tripsTaken}_2 : \mathsf{Gender} \to \mathbb{N}$

$\mathsf{tripsTaken}_2\ \mathsf{male} = 30$

$\mathsf{tripsTaken}_2\ \mathsf{female} = 11$

$\mathsf{tripsTaken}_2\ \mathsf{other} = 0$

In this case, there have been 41 trips taken which is above the minimum threshold therefore the execute' will function will check to see if the plan is fair for each action. The first action of the plan is a drive action therefore it is trip agnostic so will be applied. The second action is a drive passenger action that uses a male taxi driver. To apply this action the execute function has to have a proof that after the drive passenger action is completed the IsFairForAll property holds. This holds true for this example. The last action of the plan is also a drive passenger action using a male taxi driver. After the application of the last action the IsFairForAll property will hold therefore in this example the execute' function returns the expected world.

Example 58 (Unfair Assignments)**.** This final example will consider the $\mathsf{tripsTaken}_3$ function:

$\mathsf{tripsTaken}_3 : \mathsf{Gender} \to \mathbb{N}$

$\mathsf{tripsTaken}_3\ \mathsf{male} = 30$

$\mathsf{tripsTaken}_3\ \mathsf{female} = 9$

$\mathsf{tripsTaken}_3\ \mathsf{other} = 0$

In this case, there have been 39 trips taken which is above the minimum threshold therefore the execute' will function will check to see if the plan is fair for each action. The first action of the plan is a drive action therefore it is trip agnostic so will be applied. Applying the next action in the plan would produce a trip assignment that is unfair therefore a gender bias error will be given. The proof the error returns is not very readable for users therefore an additional displayErrorMessage function is defined in the Agda formalisation that produces the following output:

```
drivePassenger (taxi 2F) (person 2F) (location 2F) (location 0F) ,
"The gender female is not proportional:
  the assignment 22 is not greater than the lower bound 25."
```

6.4.4 Example 3: Universal Properties

In PDDL domains there are often many assumed implicit universal properties that are enforced by the action definitions rather than explicitly stated. For example, in blocksworld, it is assumed that it is never possible for any grounded *holding* predicate and *handEmpty* to be true at the same time. It is enforced on an implicit meta-level by ensuring that the preconditions and postconditions of all actions will never allow for the creation of such a world. Historically, this type of property could be encoded by using a universal formula in STRIPS [32, 69]. PDDL does not include universal formulae for efficiency reasons however this thesis proposes that it could be useful to explicitly encode these properties for testing purposes. This section will demonstrate how universal properties can be explicitly encoded into the action handler. Action handlers can then be used to sanity-check that domains enforce their universal properties.

To implement the BlocksWorld example objects are defined using the Fin type in order to meta-reason over the set of objects. It is now possible to define the list of all holding predicates with the following code:

```
allObjects : List Object
allObjects = Data.List.map obj (allFin objects)

allHolding : List Predicate
allHolding = Data.List.map holding allObjects
```

It is now possible to define the cantHoldWithHandEmpty function to enforce the desired property. This function first checks if the *handEmpty* predicate is true in the world and if so checks if any possible *holding* predicate is also true:

```
cantHoldWithHandEmpty : World → Set
cantHoldWithHandEmpty w with handempty ∈? w
... | no ¬p = ⊤
... | yes p with any? (λ x → x ∈? w) allHolding
... | no ¬p = ⊤
... | yes p₁ = ⊥
```

To create the action handler that enforces the cantHoldWithHandEmpty property, it is necessary to return a world where this property holds. Since this property

is dependent on a world, the dependent product type will be used to create the correctWorld type. This will then be used to define the BlocksWorld action handler:

```
correctWorld = Σ World λ w → cantHoldWithHandEmpty w

BlocksWorldActionHandler : Set
BlocksWorldActionHandler = (α : Action) → correctWorld → Maybe correctWorld
```

There is no guarantee that the updateWorld function will produce a correctWorld therefore the type definition will change to Maybe correctWorld to reflect this. The decidability of cantHoldWithHandEmpty is trivial and is referred to as cantHoldWithHandEmpty?. The enriched handler is defined by using the decidability of cantHoldWithHandEmpty to check that the updateWorld function returns a correct world:

```
enriched-σ Γ α (w , prf) with cantHoldWithHandEmpty?
                                (updateWorld (effects (Γ α)) w)
... | no ¬p = nothing
... | yes p = just ((updateWorld (effects (Γ α)) w) , p)
```

The execute function is simply defined by recursively applying the enriched handler as long as it returns a correctWorld:

```
execute' : Plan →
           BlocksWorldActionHandler →
           correctWorld →
           Maybe correctWorld
execute' halt σ w = just w
execute' (a :: f) σ w with σ a w
... | nothing = nothing
... | just w = execute' f σ w
```

With this definition of the execution function, it is now possible to check for a given plan and world that no predicate *holding* $?x$ is true whilst the *handEmpty* predicate is also true. Whilst it is known that this property will always hold true for BlocksWorld, this section has demonstrated how to encode a universal property explicitly. By encoding explicit definitions of implicit properties, it is possible to

check that these properties hold over a set of plans and worlds. This will give additional assurance over PDDL domains. The methodology taken in this section can also be made more generic in order to make it simple to encode similar properties.

6.5 Applications of Agda Library

An accompanying Agda library [51] has been defined along with this thesis containing the full formalisation and code presented. It has been designed with the intention that it is friendly to users from the AI planning community. The library can be used in two ways. The first is to verify plans for untyped PDDL domains and problems. The second, as shown in this chapter, is to fully utilise the Agda implementation to define enriched handlers. The methodology for both as well as their applications will be given in this section.

Automatic Verification

The automatic translation and verification of plans for untyped PDDL domains and problems are run by editing a run.lisp program in the relevant Automation Algorithm folder. The general methodology is as follows:

1. Open run.lisp.
2. Change the domainfile and problemfile variables to the path of the desired PDDL domain and problem specifications
3. Set the name of the generated Agda file by changing the outputfile variable.
4. Set the plan file you want to type check to the plan variable.
5. Run the run.lisp file.

This methodology does not require a user to write any Agda code at all. The user simply has to define the relevant variable in lisp and run the run.lisp file. The run.lisp file generate and type check an Agda file where if the file type checks then the plan is known to be valid as discussed in Section 5.6.

Verification with Enriched Handlers

This is the summary of the general methodology to set up, verify and execute a PDDL problem using an enriched handler in the Agda library:

1. Import the Semantics and Plan files from the Plan folder.
2. Create and import a Domain file for your problem.

3. Define an initial world, goal state and a plan.
4. Use the typing derivation to check that the plan is valid for the initial world and goal state provided.
5. Create an enriched handler and evaluation function:
   (a) Model the additional properties as types.
   (b) Show that the additional properties are decidable if necessary.
   (c) Create the relevant error types.
   (d) Define an action handler that includes the additional properties.
   (e) Define an evaluation function for the action handler.
   (f) Define an enriched canonical handler.
6. Import the enriched handler that you want to use.
7. Use the relevant evaluation function to execute your handler on the initial world.

Although the primary purpose of the presented work is to illustrate the benefits of embedding plan validation in a dependently-typed language, there are also many possible future extensions and applications of this work.

In this chapter steps 2, 3 and 4 have been performed manually due to the changes in syntax for typed domains and problems however there is no reason that these steps could not be automated for typed domains. It should be noted that step 4 is still automated via a solver function. These steps have not been automated because the presented work in this chapter is to test the limits of type-driven code development in AI. It would be possible to update the automation algorithms for these steps however it would not be possible to automate step 5 in the same way.

This thesis envisages several applications of this work for AI planning. As discussed in previous chapters it is possible to use the implementation of the plan validator to verify plans using the typing relation as well as extract them.

For enriched handlers, there may be use cases when software and hardware requirements, or indeed legal regulations, do not permit the direct deployment of code extracted from Agda. For example in the autonomous car industry, the set of usable languages is strictly regulated. In such cases, the methodology proposed in this chapter can be used as part of a broader modelling and simulation environment. In fact, this chapter proposes this to be the most promising avenue for applications of

these ideas. The enriched handlers proposed in this chapter enhance exactly this modelling aspect, by opening a way for lightweight and flexible modelling of arbitrary properties of plans separately from (and in addition to) the automated plan search performed by an AI planner such as STRIPS.

6.6 Discussion

This chapter has introduced a final Agda formalisation that addressed the shortcomings of the formalisations presented in the previous chapters. The new formalisation ensured the consistency of the effects of actions whilst maintaining the decidability of the derivation. This allowed for the definition of a system that not only has the fastest type checking time but also removed the need for the inconsistency assumption.

This chapter has also presented a novel methodology of using enriched handlers for embedding AI plans into a richer programming and modelling environment in Agda. The main focus of this was to show that the idea of a verification framework combining automated solvers and planners on the one hand and richer type-driven programming environments on the other hand has its merits, and can be implemented in an interesting, natural and even user-friendly way. This achieves Objective 6. One of the main merits of this approach is that properties can be expressed in the Agda system that are inexpressible in PDDL. The Agda system gives additional guarantees over these properties such as termination of higher-order functions. This chapter therefore concludes that the best extensions to the work presented in this thesis will come from further utilising the benefits of the Agda implementation.

6.7 Related Work

Formalisation of Planning in Other Theorem Provers. The extrinsic verification that this chapter has presented is a novel use case of the Agda implementation that cannot be expressed in other state-of-the-art plan validators [3, 57]. In particular, it is not possible to express and extract the higher-order properties in Isabelle that have been presented in this chapter. This novel approach has shown how Agda

can be used to extend the expressibility of properties in a way that is unique to dependently typed programming languages.

Explainable AI. Extrinsic tools that introduce meta properties over PDDL are already being used in the field of Explainable AI. In [16] a wrapper over PDDL was created so that users can express "contrastive questions" to better understand and explore why a planner has chosen certain actions over others. An example contrastive question could be *"Why did you choose action A rather than B?"*. To accomplish this, users give questions in natural language which are then converted into formal constraints that are then compiled down into PDDL. These additional constraints force the planner to choose different actions which the wrapper will use to generate a contrastive explanation by comparing the original plan to the new plan generated from the additional constraints. The user can then add additional constraints by asking further contrastive questions. This ability to ask further questions is particularly useful as it allows a user to build complex constraints to gain a deeper understanding of a plan.

Plan-Property Dependencies. There is also work [28, 29] that introduces plan-property dependencies which impose boolean functions over plans which allows a user to query why a plan satisfies certain properties over others. These properties are equivalent to soft goals in PDDL [38]. This work explains plans by showing the cost of satisfying certain properties over others by computing the minimal unsolvable goal subsets of a planning problem. An example question in this work could be *"Why does the plan not satisfy the property X?"* and a potential reply could be *"because then we would have to forgo property Y and property Z"*. To be able to do this, they compile plan properties into goal facts and then compute the minimal unsolvable goal subsets to produce plan explanations. This work can also reason about plan properties in linear temporal logic.

In comparison to the work presented in this thesis, both Chapter 4 and Chapter 5 define extrinsic properties in a domain-independent manner. Whilst the verification and execution of plans in our system are domain-independent, the enriched handlers are not necessarily domain independent. For example, the more generic properties of FuelAwareActionHandler could be used in any domain, however the GenderAwareActionHandler is defined specifically for the taxi domain. The benefit

of the approach taken in this chapter is the ability to define complex properties that would be undefinable in either of the previous systems. However, at the current moment, there is no way to compile the properties into PDDL when a plan fails.

Chapter 7

Conclusions

This thesis showcases the practical benefits of modern programming language theory whilst simultaneously improving the state-of-the-art plan validation. Aim 1 was to demonstrate why taking a Curry-Howard approach to plan validation is a useful and natural solution. In order to address this aim, this thesis formalised multiple inference systems where planning problems were represented as types and plans were executable functions that inhabited those types. This approach to plan validation was a novel practical use of the Curry-Howard correspondence which allowed for the validation of plans to provide formal guarantees about their execution. Showing that plans can be represented in this Curry-Howard style is a major contribution as the inference systems in this thesis had to be derived in such a way as to ensure that the programs that inhabit the types give the actual executable plans in the STRIPS sense.

Aim 2 was to explore how dependently-typed programming languages can be used to improve the state-of-the-art plan validation. This thesis accomplished this by implementing formal inference systems for plan validation in the dependently-typed programming language Agda. By using Agda this thesis could prove these inference systems to be sound as well as ensure that any validated plans had checkable evidence. Automation for these systems was also introduced so that their usability was more in line with state-of-the-art plan validators [3, 57]. The biggest advantage of the dependently-typed approach was the gained extensibility of the inference systems. This thesis introduced the concept of enriched handlers that could be used to check additional properties over the execution of plans. By using a

dependently-typed language it was possible to implement useful properties that are inexpressible in PDDL. On top of this, the Agda system gives additional guarantees over these properties such as the termination of higher-order functions.

7.1 Challenges

In order to address both aims, this thesis created formal inference systems in Agda. The creation of these inference systems in Agda achieves Objective 2, Objective 3 and Objective 4. The first challenge in accomplishing this task was that the existing semantics defined in PDDL were not detailed enough to be simply defined verbatim in Agda. In order to address this issue, this thesis defined a new formal language for representing plans in Chapter 4. The implementation of this logic provided new formal foundations for a subset of PDDL which achieves Objective 1 of this thesis. These foundations contained the detail necessary to be implemented and proven sound in Agda.

Another challenge that this thesis faced was discovering how best to write the inference rules for plan validation. This thesis explored two different methodologies for describing the inference rules by creating PCP logic and PCPR logic. PCP logic took the standard STRIPS approach to plan validation whereas PCPR logic took an older approach to plan validation based on a frame rule. One major issue with PCP logic was that it needed an implicit consistency assumption for soundness. This was addressed in PCPR logic where consistency was embedded directly into the rules. The implementation of PCPR logic not only addressed the consistency issue but also provided interesting insights about the interpretation of actions as discussed in Section 5.4. This work further addresses Objective 1. Unfortunately, the proofs produced by PCPR logic were very complicated making it an impractical system to use for plan validation. PCP logic on the other hand had simple proofs and could even be proven to be decidable. This thesis therefore introduced PCP* logic in Chapter 6 that addressed the issues of consistency.

Figuring out the best approach to formalise and automate PCP logic and PCPR logic in Agda was another major challenge that this thesis faced. This thesis has shown in detail its methodology behind defining the formal systems presented in

Agda. In Chapter 4 the Agda code to implement PCP logic was shown and explained. Chapter 5 explored how to best automate these systems including how to use techniques such as decidability to automatically generate proofs. This automation achieves Objective 5. Finally, Chapter 6 demonstrated how to implement verification properties at the type level of functions. This implementation of these verification properties achieves Objective 2. This thesis can therefore be used as a demonstration of the methodology for implementing a declarative system in a dependently-typed programming language.

7.2 Future Work

One limitation of the systems presented in this thesis in comparison to other plan validators is the time it takes to validate plans. Whilst the systems presented in this thesis have been shown to scale to larger problems they are much slower than their counterparts [3, 57]. This was not a focus of the work presented in this thesis and instead, this thesis focused on utilising the dependently-typed approach to its full potential by implementing inexpressible properties. One area of future work is to investigate the best methods to reduce the type-checking time whilst retaining all of the benefits of the dependently-typed approach.

Other limitations can be found in the implementation of enriched handlers. In comparison to Chapter 4 and 5 which requires no Agda knowledge to use, the definition of enriched handlers requires a user to be familiar with dependently-typed programming. This is justified because even though the verification and execution of plans in the presented systems are domain-independent, the enriched handlers are not necessarily domain independent. Because the extrinsic properties (modelled by the enriched handlers) are not part of the PDDL domain or problem, it is not possible to provide the same level of automation for generating them. In future work, it would be beneficial to lower the barrier for entry in creating and using these extrinsic properties. This could be done by the creation of a library of generic properties that could be used in many domains. Another more ambitious approach to creating a more user-friendly infrastructure for enriched handlers would be to define a DSL for enriched handlers. This would mean that a user would only have

to learn how to use the DSL.

A final limitation of the enriched handler approach at the moment is what to do in the case an extrinsic property fails. At the current moment in time, it is only possible to produce an error message along with a proof of why the plan has failed. Chapter 6 attempted to mitigate this limitation by defining extrinsic properties that will most likely be satisfied by a planner without any additional replanning. One area of future work would be to look at if it is possible to do plan repair in Agda. This thesis proposes that this issue could be addressed by compiling down additional constraints to PDDL based on the extrinsic properties of the enriched handler. Since the extrinsic properties can not be easily expressed in PDDL it would be necessary to create compilation strategies based on the errors produced by failed evaluations to force the planner to pick different actions. For example, if we have a plan that fails in our taxi domain because it has disproportionately picked men over women in a plan we could fix this by removing a certain number of male taxi drivers from the planning problem so that the planner no longer has the option to choose them. Another interesting potential way to solve the repair plans would be to interface Agda with an SMT solver that can produce new plans.

7.3 Reflection

This work presented in this thesis has largely accomplished what it set out to do and has achieved all of its objectives, as discussed. One major accomplishment of this thesis is the complete automation of plan validation within Agda. This allows for members of the planning community to be able to use the work presented without knowing any Agda at all. This greatly reduces the barrier needed to utilise this work. At the current stage of this research, there is a large trade-off in time for validation in comparison to modern planners. Whilst it would be great to improve this, and an obvious next step, this thesis argues that the validation is still useful. This thesis envisions users using the inference systems as a final extra validation step to double-check plans that need the increased assurance.

There are many further potential benefits from defining the inference systems in Agda. This thesis has shown a definition of enriched handlers that introduce a novel

way of adding constraints to plans. If time permitted, it would have been great to further explore this idea as there are so many different avenues for improvement. This includes defining new unique properties that can only be expressed with dependent types, plan repair, and the possibility to add some of these constraints to a planner.

7.4 Supporting Code

The supporting code for this thesis is located in the following repository [51]. This code includes all of the inference systems as well as their proofs. All of the automation experiments are also included in the repository.

Bibliography

[1] Abdulaziz, M., and Berger, D. Computing plan-length bounds using lengths of longest paths. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021* (2021), AAAI Press, pp. 11709–11717.

[2] Abdulaziz, M., Gretton, C., and Norrish, M. A verified compositional algorithm for AI planning. In *10th International Conference on Interactive Theorem Proving (ITP 2019)* (2019), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[3] Abdulaziz, M., and Lammich, P. A formally verified validator for classical planning problems and solutions. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)* (2018), IEEE, pp. 474–479.

[4] Abdulaziz, M., and Lammich, P. AI planning languages semantics. *Arch. Formal Proofs* (2020).

[5] aibasel. Downward benchmarks. https://github.com/aibasel/downward-benchmarks, 2020.

[6] Albrecht, D. W., Crossley, J. N., and Jeavons, J. S. New Curry-Howard terms for full linear logic. *Theor. Comput. Sci. 185*, 2 (1997), 217–235.

[7] Appel, A. W., Beringer, L., Chlipala, A., Pierce, B. C., Shao, Z., Weirich, S., and Zdancewic, S. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 375*, 2104 (2017), 20160331.

[8] Artho, C., Barringer, H., Goldberg, A., Havelund, K., Khurshid, S., Lowry, M., Pasareanu, C., Roşu, G., Sen, K., Visser, W., et al. Combining test case generation and a verification. *Theoretical Computer Science 336*, 2-3 (2005), 209–234.

[9] Bagnall, A., and Stewart, G. Certifying the true error: Machine learning in Coq with verified generalization guarantees. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 2662–2669.

[10] Barrett, C., and Tinelli, C. Satisfiability modulo theories. In *Handbook of model checking*. Springer, 2018, pp. 305–343.

[11] Berdine, J., Calcagno, C., and O'hearn, P. W. Smallfoot: Modular automatic assertion checking with separation logic. In *International Symposium on Formal Methods for Components and Objects* (2005), Springer, pp. 115–137.

[12] Berdine, J., Calcagno, C., and O'hearn, P. W. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems* (2005), Springer, pp. 52–68.

[13] Blanchette, J. C., Kaliszyk, C., Paulson, L. C., and Urban, J. Hammering towards QED. *Journal of Formalized Reasoning 9*, 1 (2016), 101–148.

[14] Bryce, D., Gao, S., Musliner, D., and Goldman, R. SMT-based nonlinear PDDL+ planning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence* (2015).

[15] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., and Rodriguez, D. Moving fast with software verification. In *NASA Formal Methods Symposium* (2015), Springer, pp. 3–11.

[16] Cashmore, M., Collins, A., Krarup, B., Krivic, S., Magazzeni, D., and Smith, D. Towards explainable AI planning as a service. 2nd ICAPS

Workshop on Explainable Planning, XAIP 2019 ; Conference date: 12-07-2019 Through 12-07-2019.

[17] Cashmore, M., Fox, M., Long, D., and Magazzeni, D. A compilation of the full PDDL+ language into SMT. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence* (2016).

[18] Cervesato, I., and Pfenning, F. A linear logical framework. *Inf. Comput. 179*, 1 (2002), 19–75.

[19] Chakravarty, M. M., Chapman, J., MacKenzie, K., Melkonian, O., Peyton Jones, M., and Wadler, P. The extended UTxO model. In *International Conference on Financial Cryptography and Data Security* (2020), Springer, pp. 525–539.

[20] Chapman, J., Kireev, R., Nester, C., and Wadler, P. System F in Agda, for fun and profit. In *International Conference on Mathematics of Program Construction* (2019), Springer, pp. 255–297.

[21] Chrpa, L., Surynek, P., and Vyskočil, J. Encoding of planning problems and their optimizations in linear logic. In *Applications of Declarative Programming and Knowledge Management*. Springer, 2007, pp. 54–68.

[22] Classen, J., Eyerich, P., Lakemeyer, G., and Nebel, B. Towards an integration of GOLOG and planning. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence* (2007), M. M. Veloso, Ed., pp. 1846–1851.

[23] Coquand, T., and Huet, G. The calculus of constructions. *Information and Computation 76*, 2-3 (1988), 95–120.

[24] Curry, H. B. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America 20*, 11 (1934), 584.

[25] Dennett, D. C. Cognitive wheels: The frame problem of AI. *Minds, machines and evolution* (1984), 129–151.

[26] Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. Semantic attachments for domain-independent planning systems. In *Nineteenth International Conference on Automated Planning and Scheduling* (2009).

[27] Dutertre, B., and De Moura, L. The yices SMT solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf 2*, 2 (2006), 1–2.

[28] Eifler, R., Cashmore, M., Hoffmann, J., Magazzeni, D., and Steinmetz, M. A new approach to plan-space explanation: Analyzing plan-property dependencies in oversubscription planning. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2020), vol. 34, pp. 9818–9826.

[29] Eifler, R., Steinmetz, M., Torralba, A., and Hoffmann, J. Plan-space explanation via plan-property dependencies: Faster algorithms & more powerful properties. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence* (2021), pp. 4091–4097.

[30] Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., and Barrett, C. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *International Conference on Computer Aided Verification* (2017), Springer, pp. 126–133.

[31] Ernst, G. W., and Newell, A. *GPS: A case study in generality and problem solving.* Academic Pr, 1969.

[32] Fikes, R., and Nilsson, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*, 3/4 (1971), 189–208.

[33] Fisher, K., Launchbury, J., and Richards, R. The HACMS program: using formal methods to eliminate exploitable bugs. *Phil. Trans. Royal Society* (2017).

[34] Floyd, R. W. Assigning meanings to programs. In *Program Verification.* Springer, 1993, pp. 65–81.

[35] Fourati, F., Bhiri, M. T., and Robbana, R. Verification and validation of PDDL descriptions using Event-B formal method. In *2016 5th International Conference on Multimedia Computing and Systems (ICMCS)* (Sept 2016), pp. 770–776.

[36] Fox, M., and Long, D. PDDL+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space* (2002), vol. 4, p. 34.

[37] Fox, M., and Long, D. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence 20* (2003), 61–124.

[38] Gerevini, A., and Long, D. Plan constraints and preferences in PDDL3. Tech. rep., Technical Report 2005-08-07, Department of Electronics for Automation, 2005.

[39] Ghallab, M., Nau, D., and Traverso, P. *Automated Planning: theory and practice*. Elsevier, 2004.

[40] Gomes, C. P., Kautz, H., Sabharwal, A., and Selman, B. Satisfiability solvers. *Foundations of Artificial Intelligence 3* (2008), 89–134.

[41] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Roux, S. L., Mahboubi, A., O'Connor, R., Biha, S. O., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., and Théry, L. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings* (2013), S. Blazy, C. Paulin-Mohring, and D. Pichardie, Eds., vol. 7998 of *Lecture Notes in Computer Science*, Springer, pp. 163–179.

[42] Gonthier, G., et al. Formal proof–the four-color theorem. *Notices of the AMS 55*, 11 (2008), 1382–1393.

[43] Green, C. Theorem proving by resolution as a basis for question-answering systems. *Machine intelligence 4* (1969), 183–205.

[44] Hales, T. C. A proof of the Kepler conjecture. *Annals of mathematics* (2005), 1065–1185.

[45] Hanks, S., and McDermott, D. Nonmonotonic logic and temporal projection. *Artificial intelligence 33*, 3 (1987), 379–412.

[46] Haslum, P. Inval. `https://github.com/patrikhaslum/INVAL`, 2022.

[47] Havelund, K., Groce, A., Holzmann, G., Joshi, R., and Smith, M. Automated testing of planning models. In *International Workshop on Model Checking and Artificial Intelligence* (2008), Springer, pp. 90–105.

[48] Hayes, P. J. The frame problem and related problems in artificial intelligence. In *Readings in Artificial Intelligence*. Elsevier, 1981, pp. 223–230.

[49] Helmert, M. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence 173*, 5-6 (2009), 503–535.

[50] Hendler, J. A., Tate, A., and Drummond, M. AI planning: Systems and techniques. *AI magazine 11*, 2 (1990), 61–61.

[51] Hill, A., Daggitt, M., Farka, F., Komendantskaya, K., and Schwaab, C. DOI of Agda code for thesis. `https://doi.org/10.5281/zenodo.7953083`, 2023.

[52] Hill, A., Komendantskaya, E., Daggitt, M. L., and Petrick, R. P. A. Actions you can handle: dependent types for AI plans. In *TyDe 2021: Proceedings of the 6th ACM SIGPLAN International Workshop on Type-Driven Development, Virtual Event, Korea, 22 August 2021* (2021), H. Ko and D. Orchard, Eds., ACM, pp. 1–13.

[53] Hill, A., Komendantskaya, E., and Petrick, R. P. A. Proof-Carrying Plans: a Resource Logic for AI planning. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020* (2020), ACM, pp. 14:1–14:13.

[54] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (1969), 576–580.

[55] Hopcroft, J. E., Motwani, R., and Ullman, J. D. Introduction to automata theory, languages, and computation. *Acm Sigact News 32*, 1 (2001), 60–65.

[56] Howard, W. A. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism 44* (1980), 479–490.

[57] Howey, R., Long, D., and Fox, M. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *16th IEEE International Conference on Tools with Artificial Intelligence* (2004), pp. 294–301.

[58] Ishtiaq, S. S., and O'Hearn, P. W. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2001), pp. 14–26.

[59] Jacopin, E. Classical AI planning as theorem proving: The case of a fragment of linear logic. In *AAAI Fall Symposium on Automated Deduction in Nonstandard Logics* (1993), AAAI Press Publications Palo Alto, CA, pp. 62–66.

[60] Kahn, G. Natural semantics. In *Annual symposium on theoretical aspects of computer science* (1987), Springer, pp. 22–39.

[61] Katz, G., Barrett, C., Dill, D. L., Julian, K., and Kochenderfer, M. J. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International conference on computer aided verification* (2017), Springer, pp. 97–117.

[62] Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D. L., Kochenderfer, M. J., and Barrett, C. W. The marabou framework for verification and analysis of deep neural networks. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (2019), I. Dillig and S. Tasiran, Eds., vol. 11561 of *Lecture Notes in Computer Science*, Springer, pp. 443–452.

[63] Kautz, H., and Selman, B. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 workshop on planning as combinatorial search* (1998), vol. 58260, sn, pp. 58–60.

[64] Kautz, H. A., and Selman, B. Planning as satisfiability. In *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings* (1992), B. Neumann, Ed., John Wiley and Sons, pp. 359–363.

[65] Khatib, L., Muscettola, N., and Havelund, K. Verification of plan models using UPPAAL. In *International Workshop on Formal Approaches to Agent-Based Systems* (2000), Springer, pp. 114–122.

[66] Kokke, W. Schmitty the solver. `https://github.com/wenkokke/schmitty`, 2022.

[67] Leroy, X., et al. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt 53* (2012).

[68] Levesque, H. J., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming 31*, 1 (1997), 59–83.

[69] Lifschitz, V. On the semantics of STRIPS. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop* (1987), pp. 1–9.

[70] Lin, F. Situation calculus. *Foundations of Artificial Intelligence 3* (2008), 649–669.

[71] Lloyd, J. W. *Foundations of logic programming.* Springer Science & Business Media, 2012.

[72] Long, D., Fox, M., and Howey, R. Planning domains and plans: validation, verification and analysis. In *Proc. Workshop on V&V of Planning and Scheduling Systems* (2009).

[73] Martin-Löf, P. An intuitionistic theory of types. *Twenty-five years of constructive type theory 36* (1972), 127–172.

[74] Martin-Löf, P. Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*, vol. 104. Elsevier, 1982, pp. 153–175.

[75] McCarthy, J. Situations, actions, and causal laws. Tech. rep., Stanford University CA department of Computer Science, 1963.

[76] McCarthy, J. Applications of circumscription to formalizing common-sense knowledge. *Artificial intelligence 28*, 1 (1986), 89–116.

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

[78] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. PDDL – The Planning Domain Definition Language (Version 1. 2). Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[79] Moura, L. d., and Bjørner, N. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer, pp. 337–340.

[80] Muise, C. Planning. domains. *ICAPS system demonstration* (2016), 242–250.

[81] Nanevski, A. Separation logic and concurrency. *Oregon programming languages summer school* (2016).

[82] Nanevski, A., Banerjee, A., and Garg, D. Dependent type theory for verification of information flow and access control policies. *ACM Transactions on Programming Languages and Systems (TOPLAS) 35*, 2 (2013), 1–41.

[83] Nanevski, A., Morrisett, G., and Birkedal, L. Polymorphism and separation in Hoare type theory. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming* (2006), pp. 62–73.

[84] Narodytska, N., Kasiviswanathan, S., Ryzhyk, L., Sagiv, M., and Walsh, T. Verifying properties of binarized deep neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2018), vol. 32.

[85] Nau, D. S. Current trends in automated planning. *AI magazine 28*, 4 (2007), 43–43.

[86] Necula, G. C. Proof-carrying code. In *Encyclopedia of Cryptography and Security, 2nd Ed.* Association for Computing Machinery, 2011, pp. 984–986.

[87] Norell, U. *Towards a practical programming language based on dependent type theory*, vol. 32.

[88] Norell, U. Agda prelude. `https://github.com/UlfNorell/agda-prelude`, 2022.

[89] O'Hearn, P. W., and Pym, D. J. The logic of bunched implications. *Bulletin of Symbolic Logic 5*, 2 (1999), 215–244.

[90] O'neil, C. *Weapons of math destruction: How big data increases inequality and threatens democracy.* Crown, 2016.

[91] Ong, L. Higher-order model checking: An overview. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015* (2015), pp. 1–15.

[92] O'Hearn, P., Reynolds, J., and Yang, H. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic* (2001), Springer, pp. 1–19.

[93] O'hearn, P. W. Resources, concurrency, and local reasoning. *Theoretical computer science 375*, 1-3 (2007), 271–307.

[94] Paulson, L. C. *Isabelle: A generic theorem prover*, vol. 828. Springer Science & Business Media, 1994.

[95] Pednault, E. P. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning* (1989), pp. 324–332.

[96] Pednault, E. P. ADL and the state-transition model of action. *Journal of logic and computation 4*, 5 (1994), 467–512.

[97] Penberthy, J. S., and Weld, D. S. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, USA, October 25-29, 1992* (1992), B. Nebel, C. Rich, and W. R. Swartout, Eds., Morgan Kaufmann, pp. 103–114.

[98] Penix, J., Pecheur, C., and Havelund, K. Using model checking to validate AI planner domain models. In *Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard* (1998).

[99] Petrick, R. P. P2: A baseline approach to planning with control structures and programs. In *Workshop on Generalized Planning: Macros, Loops, Domain Control (ICAPS-09)* (2009), pp. 59–64.

[100] Plotkin, G. D. *A structural approach to operational semantics.* Aarhus university, 1981.

[101] Polakow, J., and Pfenning, F. *Ordered linear logic and applications.* Carnegie Mellon University Pittsburgh, 2001.

[102] Polikarpova, N., and Sergey, I. Structuring the synthesis of heap-manipulating programs. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 72.

[103] Potassco. PDDL benchmark instances. https://github.com/potassco/pddl-instances, 2017.

[104] Pulina, L., and Tacchella, A. Challenging SMT solvers to verify neural networks. *AI Communications 25*, 2 (2012), 117–135.

[105] Pym, D. Resource semantics: logic as a modelling technology. *ACM SIGLOG News 6*, 2 (2019), 5–41.

[106] Raimondi, F., Pecheur, C., and Brat, G. PDVer, a tool to verify PDDL planning domains. In *ICAPS'09 Workshop on Verification and Validation of Planning and Scheduling Systems, September 20, 2009, Thessaloniki, Greece* (2009).

[107] Reiter, R. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In *Artificial and Mathematical Theory of Computation, Papers in Honor of John McCarthy on the occasion of his sixty-fourth birthday* (1991), V. Lifschitz, Ed., Academic Press / Elsevier, pp. 359–380.

[108] Reiter, R. *Knowledge in action: logical foundations for specifying and implementing dynamical systems.* MIT press, 2001.

[109] Reynolds, J. C. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (2002), IEEE, pp. 55–74.

[110] Rizaldi, A., Immler, F., Schürmann, B., and Althoff, M. A formally verified motion planner for autonomous vehicles. In *International Symposium on Automated Technology for Verification and Analysis* (2018), Springer, pp. 75–90.

[111] Scala, E., Ramirez, M., Haslum, P., and Thiébaux, S. Numeric planning with disjunctive global constraints via SMT. In *Twenty-Sixth International Conference on Automated Planning and Scheduling* (2016).

[112] Schack-Nielsen, A., and Schürmann, C. Celf - A logical framework for deductive and concurrent systems (system description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings* (2008), A. Armando, P. Baumgartner, and G. Dowek, Eds., vol. 5195 of *Lecture Notes in Computer Science*, Springer, pp. 320–326.

[113] Schmidt, D. A. *Denotational semantics: a methodology for language development.* William C. Brown Publishers, 1986.

[114] Schwaab, C., Komendantskaya, E., Hill, A., Farka, F., Petrick, R. P. A., Wells, J. B., and Hammond, K. Proof-Carrying Plans. In *Practical Aspects of Declarative Languages - 21th International Symposium, PADL 2019, Lisbon, Portugal, January 14-15, 2019, Proceedings* (2019), J. J.

Alferes and M. Johansson, Eds., vol. 11372 of *Lecture Notes in Computer Science*, Springer, pp. 204–220.

[115] Setzer, A. Modelling bitcoin in agda. *CoRR abs/1804.06398* (2018).

[116] Shoukry, Y., Nuzzo, P., Saha, I., Sangiovanni-Vincentelli, A. L., Seshia, S. A., Pappas, G. J., and Tasbuada, P. Scalable lazy SMT-based motion planning. In *2016 IEEE 55th Conference on Decision and Control (CDC)* (2016), IEEE, pp. 6683–6688.

[117] Smith, M. H., Holzmann, G. J., Cucullu III, G. C., and Smith, B. D. Model checking artificial intelligence based planners: Even the best laid plans must be verified.

[118] Sorensen, M. H., and Urzyczyn, P. *Lectures on the Curry-Howard Isomorphism*, vol. 149 of *Studies in Logic*. Elsevier Science, 2006.

[119] Steedman, M. Plans, affordances, and combinatory grammar. *Linguistics and Philosophy 25*, 5-6 (2002), 723–753.

[120] van Atten, M. The Development of Intuitionistic Logic. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Summer 2022 ed. Metaphysics Research Lab, Stanford University, 2022.

[121] Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D., and Peyton-Jones, S. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming* (2014), pp. 269–282.

[122] Wadler, P. Propositions as types. *Commun. ACM 58*, 12 (2015), 75–84.

[123] Wadler, P., Kokke, W., and Siek, J. G. *Programming Language Foundations in Agda*. July 2020.

[124] Waldinger, R. J. Achieving several goals simultaneously. *Machine Intelligence 8* (1977).

[125] Walt, P. v. d., and Swierstra, W. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages* (2012), Springer, pp. 157–173.

[126] Wilkins, D. E. *Practical planning: extending the classical AI planning paradigm.* Elsevier, 2014.

[127] Xu, F., Uszkoreit, H., Du, Y., Fan, W., Zhao, D., and Zhu, J. Explainable AI: A brief survey on history, research areas, approaches and challenges. In *CCF international conference on natural language processing and Chinese computing* (2019), Springer, pp. 563–574.

[128] Yu, D., Hamid, N. A., and Shao, Z. Building certified libraries for PCC: Dynamic storage allocation. *Science of Computer Programming 50*, 1-3 (2004), 101–127.

[129] Zarriess, B., and Classen, J. Decidable verification of GOLOG programs over non-local effect actions. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.* (2016), D. Schuurmans and M. P. Wellman, Eds., AAAI Press, pp. 1109–1115.

Appendix A

Inconsistency Example

```
-- A simple example that demonstrates violation of the
-- implicit consistency assumpion
data Predicate : Set where
  handEmpty : Predicate

-- There is only one, naughty, action, which violates
-- the implicit consistency assumption
data Action : Set where
  naughty : Action

-- The naughty action does not have any preconditions and
-- introduces an atomic predicate and its negation as
-- postconditions
Γ : Context
Γ (naughty) = record {
  preconditions = [] ;
  effects = (+ , handEmpty) :: (- , handEmpty) :: []}

-- Despite the obvious inconsistency,
-- the following plan has a derivation that type checks:
plan = naughty :: halt
```

```
Q : Form
Q = atom handEmpty ∧ ¬ handEmpty

Derivation : Γ ⊢ [] ⇝ (Q ↓[ + ] []) : plan
Derivation = seq (λ { ()}) (halt λ { (here px)
                    → there (here px) ; (there (here px)) → here px})

-- But, at the same time, action naughty
-- invalidates consistency of entire development
-- (given the implicit consistency assumption):
postulate
  implicit-consistency-assumption : (z : Polarity) (R : Predicate)
                                  → ∀ S → (z , R) ∈ S → (neg z , R) ∉ S

prop-inconsistent : ⊥
prop-inconsistent =
  implicit-consistency-assumption + handEmpty (effects (Γ naughty))
    (here refl) (there (here refl))
```

Appendix B

PDDL Examples

This appendix contains the PDDL domain and problem definitions for the test examples shown in Table 5.2 and Table 6.1. All definitions were taken from a benchmark repository [5].

B.1 Blocks World

B.1.1 Domain Definition

```
  (define (domain blocksworld)
    (:requirements
        :strips
    )

    (:predicates
        (handEmpty)
        (holding ?x)
        (onTable ?x)
        (on ?x ?y)
        (clear ?x))

(:action pickup_from_table
    :parameters
        (?x)
    :precondition
        (and
           (handEmpty)
           (onTable ?x)
           (clear ?x)
        )
    :effect
        (and
           (not (handEmpty))
           (not (onTable ?x))
           (holding ?x)
        ))

(:action putdown_on_table
    :parameters
        (?x)
    :precondition
        (and
            (holding ?x)
        )
    :effect
        (and
            (not (holding ?x))
            (onTable ?x)
            (handEmpty)
        ))

(:action pickup_from_stack
    :parameters
        (?x ?y)
    :precondition
        (and
            (on ?x ?y)
            (clear ?x)
            (handEmpty)
        )
    :effect
        (and
            (not (on ?x ?y))
            (not (handEmpty))
            (holding ?x)
            (clear ?y)
        ))

(:action putdown_on_stack
    :parameters
        (?x ?y)
    :precondition
        (and
            (holding ?x)
            (clear ?y)
        )
    :effect
        (and
            (not (holding ?x))
            (not (clear ?y))
            (on ?x ?y)
            (handEmpty)
        ))
)
```

B.1.2 Problem Definitions

Problem A

```
(define (problem blocksworld-problem1)
  (:domain blocksworld)
  (:objects a b c d e f1)
  (:init (onTable a) (onTable b) (onTable c) (onTable d)
         (onTable e) (onTable f1) (clear a) (clear b)
         (clear c) (clear d) (clear e) (clear f1)
         (handEmpty))
  (:goal (and (on a b) (on b c) (on c d) (on d e) (on e f1)))
)
```

Problem B

```
(define (problem BLOCKS-7-0)
(:domain blocksworld)
(:objects C F A B G D E )
(:INIT (CLEAR E) (ONTABLE D) (ON E G) (ON G B) (ON B A) (ON A F) (ON F C)
 (ON C D) (HANDEMPTY))
(:goal (AND (ON A G) (ON G D) (ON D B) (ON B C) (ON C F) (ON F E)))
)
```

Problem C

```
(define (problem BLOCKS-11-0)
(:domain blocksworld)
(:objects F A K H G E D I C J B )
(:INIT (CLEAR B) (CLEAR J) (CLEAR C) (ONTABLE I) (ONTABLE D) (ONTABLE E)
 (ON B G) (ON G H) (ON H K) (ON K A) (ON A F) (ON F I) (ON J D) (ON C E)
 (HANDEMPTY))
(:goal (AND (ON A J) (ON J D) (ON D B) (ON B H) (ON H K) (ON K I) (ON I F)
            (ON F E) (ON E G) (ON G C)))
)
```

B.2 Logistics

B.2.1 Domain Definition

```
(define (domain logistics)
  (:requirements :strips)
  (:predicates  (package ?obj)
        (truck ?truck1)
(airplane ?airplane1)
                (airport ?airport)
                (location ?loc)
(in-city ?obj ?city1)
                (city ?city1)
(at ?obj ?loc)
(isin ?obj ?obj))

(:action load-truck
  :parameters
   (?obj
    ?truck1
    ?loc)
  :precondition
   (and (package ?obj) (truck ?truck1) (location ?loc)
   (at ?truck1 ?loc) (at ?obj ?loc))
  :effect
   (and (not (at ?obj ?loc)) (isin ?obj ?truck1)))

(:action load-airplane
  :parameters
   (?obj
    ?airplane1
    ?loc)
  :precondition
   (and (package ?obj) (airplane ?airplane1) (location ?loc)
   (at ?obj ?loc) (at ?airplane1 ?loc))
  :effect
   (and (not (at ?obj ?loc)) (isin ?obj ?airplane1)))
```

```
(:action unload-truck
  :parameters
   (?obj
    ?truck1
    ?loc)
  :precondition
   (and (package ?obj) (truck ?truck1) (location ?loc)
        (at ?truck1 ?loc) (isin ?obj ?truck1))
  :effect
   (and (not (isin ?obj ?truck1)) (at ?obj ?loc)))

(:action unload-airplane
  :parameters
   (?obj
    ?airplane1
    ?loc)
  :precondition
   (and (package ?obj) (airplane ?airplane1) (location ?loc)
        (isin ?obj ?airplane1) (at ?airplane1 ?loc))
  :effect
   (and (not (isin ?obj ?airplane1)) (at ?obj ?loc)))

(:action drive-truck
  :parameters
   (?truck1
    ?loc-from
    ?loc-to
    ?city1)
  :precondition
   (and (truck ?truck1) (location ?loc-from)
        (location ?loc-to) (city ?city1)
   (at ?truck1 ?loc-from)
   (in-city ?loc-from ?city1)
   (in-city ?loc-to ?city1))
  :effect
   (and (not (at ?truck1 ?loc-from)) (at ?truck1 ?loc-to)))

(:action fly-airplane
  :parameters
   (?airplane1
    ?loc-from
    ?loc-to)
  :precondition
   (and (airplane ?airplane1) (airport ?loc-from) (airport ?loc-to)
(at ?airplane1 ?loc-from))
  :effect
   (and (not (at ?airplane1 ?loc-from)) (at ?airplane1 ?loc-to)))
)
```

B.2.2 Problem Definitions

Problem A

```
(define (problem logistics-4-0)
(:domain logistics)
(:objects apn1 apt2 pos2 apt1 pos1 cit2 cit1 tru2 tru1 obj23 obj22 obj21
            obj13 obj12 obj11 )
(:init (package obj11) (package obj12) (package obj13) (package obj21)
 (package obj22) (package obj23) (truck tru1) (truck tru2) (city cit1)
 (city cit2) (location pos1) (location apt1) (location pos2) (location apt2)
 (airport apt1) (airport apt2) (airplane apn1) (at apn1 apt2) (at tru1 pos1)
 (at obj11 pos1) (at obj12 pos1) (at obj13 pos1) (at tru2 pos2)
 (at obj21 pos2) (at obj22 pos2) (at obj23 pos2) (in-city pos1 cit1)
 (in-city apt1 cit1) (in-city pos2 cit2) (in-city apt2 cit2))
(:goal (and (at obj11 apt1) (at obj23 pos1) (at obj13 apt1) (at obj21 pos1)))
)
```

Problem B

```
(define (problem logistics-7-0)
(:domain logistics)
(:objects apn1 apt3 pos3 apt2 pos2 apt1 pos1 cit3 cit2 cit1
          tru3 tru2 tru1 obj33 obj32 obj31 obj23 obj22 obj21
          obj13 obj12 obj11 )
(:init (package obj11) (package obj12) (package obj13) (package obj21)
 (package obj22) (package obj23) (package obj31) (package obj32)
 (package obj33) (truck tru1) (truck tru2) (truck tru3) (city cit1)
 (city cit2) (city cit3) (location pos1) (location apt1) (location pos2)
 (location apt2) (location pos3) (location apt3) (airport apt1)
 (airport apt2) (airport apt3) (airplane apn1) (at apn1 apt1) (at tru1 pos1)
 (at obj11 pos1) (at obj12 pos1) (at obj13 pos1) (at tru2 pos2)
 (at obj21 pos2)  (at obj22 pos2) (at obj23 pos2) (at tru3 pos3)
 (at obj31 pos3) (at obj32 pos3)  (at obj33 pos3) (in-city pos1 cit1)
 (in-city apt1 cit1) (in-city pos2 cit2) (in-city apt2 cit2)
 (in-city pos3 cit3) (in-city apt3 cit3))
```

```
(:goal (and (at obj22 pos2) (at obj33 apt1) (at obj12 pos2) (at obj13 apt3)
            (at obj31 apt2) (at obj23 apt1) (at obj32 pos1)))
)
```

Problem C

```
(define (problem logistics-11-0)
(:domain logistics)
(:objects apn1 apt4 pos4 apt3 pos3 apt2 pos2 apt1 pos1 cit4 cit3 cit2
          cit1 tru4 tru3 tru2 tru1 obj43 obj42 obj41 obj33 obj32 obj31
          obj23 obj22 obj21 obj13 obj12 obj11 )
(:init (package obj11) (package obj12) (package obj13) (package obj21)
 (package obj22) (package obj23) (package obj31) (package obj32)
 (package obj33) (package obj41) (package obj42) (package obj43)
 (truck tru1) (truck tru2) (truck tru3) (truck tru4) (city cit1) (city cit2)
 (city cit3) (city cit4)(location pos1) (location apt1) (location pos2)
 (location apt2) (location pos3) (location apt3) (location pos4)
 (location apt4) (airport apt1) (airport apt2) (airport apt3) (airport apt4)
 (airplane apn1) (at apn1 apt3) (at tru1 pos1) (at obj11 pos1) (at obj12 pos1)
 (at obj13 pos1) (at tru2 pos2) (at obj21 pos2) (at obj22 pos2)
 (at obj23 pos2) (at tru3 pos3) (at obj31 pos3) (at obj32 pos3)
 (at obj33 pos3) (at tru4 pos4) (at obj41 pos4) (at obj42 pos4)
 (at obj43 pos4) (in-city pos1 cit1) (in-city apt1 cit1) (in-city pos2 cit2)
 (in-city apt2 cit2) (in-city pos3 cit3) (in-city apt3 cit3)
 (in-city pos4 cit4) (in-city apt4 cit4))
(:goal (and (at obj33 apt1) (at obj22 apt2) (at obj43 pos4) (at obj11 pos1)
            (at obj23 pos1) (at obj31 pos1) (at obj12 apt2) (at obj13 pos4)
            (at obj42 apt2) (at obj21 pos4) (at obj41 pos4)))
)
```

B.3 Satellite

B.3.1 Domain Definition

```
(define (domain satellite)
(:requirements :equality :strips)
(:predicates
     (on_board ?i ?s) (supports ?i ?m) (pointing ?s ?d) (power_avail ?s)
     (power_on ?i) (calibrated ?i) (have_image ?d ?m)
     (calibration_target ?i ?d)(satellite ?x) (direction ?x)
     (instrument ?x) (mode ?x))
(:action turn_to
 :parameters ( ?s ?d_new ?d_prev)
 :precondition
    (and (satellite ?s) (direction ?d_new) (direction ?d_prev)
        (pointing ?s ?d_prev))
 :effect
    (and (pointing ?s ?d_new) (not (pointing ?s ?d_prev))))

(:action switch_on
 :parameters ( ?i ?s)
 :precondition
    (and (instrument ?i) (satellite ?s)  (on_board ?i ?s)
        (power_avail ?s))
 :effect
    (and (power_on ?i) (not (calibrated ?i)) (not (power_avail ?s))))

(:action switch_off
 :parameters ( ?i ?s)
 :precondition
    (and (instrument ?i) (satellite ?s)  (on_board ?i ?s) (power_on ?i))
 :effect
    (and (power_avail ?s) (not (power_on ?i))))

(:action calibrate
 :parameters ( ?s ?i ?d)
 :precondition
    (and (satellite ?s) (instrument ?i) (direction ?d)  (on_board ?i ?s)
        (calibration_target ?i ?d) (pointing ?s ?d) (power_on ?i))
 :effect
     (calibrated ?i))

(:action take_image
 :parameters ( ?s ?d ?i ?m)
 :precondition
    (and (satellite ?s) (direction ?d) (instrument ?i) (mode ?m)
        (calibrated ?i) (on_board ?i ?s) (supports ?i ?m)
        (power_on ?i) (pointing ?s ?d))
 :effect
     (have_image ?d ?m)))
```

B.3.2 Problem Definitions

Problem A

```
(define (problem strips-sat-x-1)
(:domain satellite)
(:objects
satellite0
instrument0
image1
spectrograph2
thermograph0
Star0
GroundStation1
GroundStation2
Phenomenon3
Phenomenon4
Star5
Phenomenon6
)
(:init
(satellite satellite0)
(instrument instrument0)
(supports instrument0 thermograph0)
(calibration_target instrument0 GroundStation2)
(on_board instrument0 satellite0)
(power_avail satellite0)
(pointing satellite0 Phenomenon6)
(mode image1)
(mode spectrograph2)
(mode thermograph0)
(direction Star0)
(direction GroundStation1)
(direction GroundStation2)
(direction Phenomenon3)
(direction Phenomenon4)
```

```
(direction Star5)
(direction Phenomenon6)
)
(:goal (and
(have_image Phenomenon4 thermograph0)
(have_image Star5 thermograph0)
(have_image Phenomenon6 thermograph0)
))
)
```

Problem B

```
(define (problem strips-sat-x-1)
(:domain satellite)
(:objects
satellite0
instrument0
satellite1
instrument1
instrument2
infrared0
infrared1
thermograph2
GroundStation1
Star0
Star2
Planet3
Star4
Planet5
Star6
Star7
Phenomenon8
Phenomenon9
)
(:init
```

```
(satellite satellite0)
(instrument instrument0)
(supports instrument0 thermograph2)
(supports instrument0 infrared0)
(calibration_target instrument0 Star0)
(on_board instrument0 satellite0)
(power_avail satellite0)
(pointing satellite0 Star6)
(satellite satellite1)
(instrument instrument1)
(supports instrument1 infrared0)
(supports instrument1 thermograph2)
(supports instrument1 infrared1)
(calibration_target instrument1 Star2)
(instrument instrument2)
(supports instrument2 thermograph2)
(supports instrument2 infrared1)
(calibration_target instrument2 Star2)
(on_board instrument1 satellite1)
(on_board instrument2 satellite1)
(power_avail satellite1)
(pointing satellite1 Star0)
(mode infrared0)
(mode infrared1)
(mode thermograph2)
(direction GroundStation1)
(direction Star0)
(direction Star2)
(direction Planet3)
(direction Star4)
(direction Planet5)
(direction Star6)
(direction Star7)
(direction Phenomenon8)
```

```
(direction Phenomenon9)
)
(:goal (and
(pointing satellite1 Planet5)
(have_image Planet3 infrared1)
(have_image Star4 infrared1)
(have_image Planet5 thermograph2)
(have_image Star6 infrared1)
(have_image Star7 infrared0)
(have_image Phenomenon8 thermograph2)
(have_image Phenomenon9 infrared0)
))
)
```

Problem C

```
(define (problem strips-sat-x-1)
(:domain satellite)
(:objects
satellite0
instrument0
satellite1
instrument1
instrument2
instrument3
satellite2
instrument4
instrument5
instrument6
satellite3
instrument7
satellite4
instrument8
thermograph2
image3
```

```
infrared1
spectrograph4
infrared0
Star1
Star4
Star0
GroundStation3
Star2
Star5
Planet6
Phenomenon7
Star8
Phenomenon9
Star10
Star11
Star12
Planet13
Planet14
Phenomenon15
Planet16
Star17
Star18
Planet19
)
(:init
(satellite satellite0)
(instrument instrument0)
(supports instrument0 spectrograph4)
(calibration_target instrument0 Star0)
(on_board instrument0 satellite0)
(power_avail satellite0)
(pointing satellite0 Star8)
(satellite satellite1)
(instrument instrument1)
```

```
(supports instrument1 infrared0)
(supports instrument1 infrared1)
(calibration_target instrument1 GroundStation3)
(instrument instrument2)
(supports instrument2 infrared1)
(supports instrument2 infrared0)
(calibration_target instrument2 Star2)
(instrument instrument3)
(supports instrument3 spectrograph4)
(supports instrument3 infrared1)
(supports instrument3 thermograph2)
(calibration_target instrument3 Star0)
(on_board instrument1 satellite1)
(on_board instrument2 satellite1)
(on_board instrument3 satellite1)
(power_avail satellite1)
(pointing satellite1 GroundStation3)
(satellite satellite2)
(instrument instrument4)
(supports instrument4 infrared1)
(supports instrument4 image3)
(supports instrument4 infrared0)
(calibration_target instrument4 Star2)
(instrument instrument5)
(supports instrument5 thermograph2)
(supports instrument5 spectrograph4)
(calibration_target instrument5 Star0)
(instrument instrument6)
(supports instrument6 infrared0)
(calibration_target instrument6 GroundStation3)
(on_board instrument4 satellite2)
(on_board instrument5 satellite2)
(on_board instrument6 satellite2)
(power_avail satellite2)
```

```
(pointing satellite2 Star4)
(satellite satellite3)
(instrument instrument7)
(supports instrument7 image3)
(calibration_target instrument7 Star2)
(on_board instrument7 satellite3)
(power_avail satellite3)
(pointing satellite3 Phenomenon9)
(satellite satellite4)
(instrument instrument8)
(supports instrument8 infrared0)
(supports instrument8 spectrograph4)
(supports instrument8 infrared1)
(calibration_target instrument8 Star2)
(on_board instrument8 satellite4)
(power_avail satellite4)
(pointing satellite4 Phenomenon9)
(mode thermograph2)
(mode image3)
(mode infrared1)
(mode spectrograph4)
(mode infrared0)
(direction Star1)
(direction Star4)
(direction Star0)
(direction GroundStation3)
(direction Star2)
(direction Star5)
(direction Planet6)
(direction Phenomenon7)
(direction Star8)
(direction Phenomenon9)
(direction Star10)
(direction Star11)
```

```
(direction Star12)
(direction Planet13)
(direction Planet14)
(direction Phenomenon15)
(direction Planet16)
(direction Star17)
(direction Star18)
(direction Planet19)
)
(:goal (and
(pointing satellite0 Phenomenon9)
(pointing satellite1 Star4)
(pointing satellite4 Star11)
(have_image Star5 image3)
(have_image Planet6 infrared1)
(have_image Phenomenon7 infrared1)
(have_image Star8 image3)
(have_image Star10 thermograph2)
(have_image Star11 infrared1)
(have_image Planet13 spectrograph4)
(have_image Planet14 thermograph2)
(have_image Phenomenon15 infrared0)
(have_image Planet16 image3)
(have_image Star17 infrared0)
))
)
```

B.4 MPrime

B.4.1 Domain Definition

```
(define (domain mystery-prime-strips)
   (:requirements :negative-preconditions :equality)
   (:predicates
       (province ?x)
       (planet ?x)
       (food ?x)
       (pleasure ?x)
       (pain ?x)
       (eats ?n1 ?n2)
       (craves ?v ?n)
       (fears ?c ?v)
       (locale ?n ?a)
       (harmony ?v ?s)
       (attacks ?i ?j)
       (orbits ?i ?j))

   (:action overcome
       :parameters (?c ?v ?n ?s1 ?s2)
       :precondition (and (pain ?c)
                    (pleasure ?v)
                          (craves ?c ?n)
                          (craves ?v ?n)
                          (food ?n)
                          (harmony ?v ?s2)
                          (planet ?s2)
                          (orbits ?s1 ?s2)
                          (planet ?s1))
       :effect (and (not (craves ?c ?n))
                    (fears ?c ?v)
                    (not (harmony ?v ?s2))
                    (harmony ?v ?s1)))
   (:action feast
       :parameters (?v ?n1 ?n2 ?l1 ?l2)
       :precondition (and (craves ?v ?n1)
                          (food ?n1)
                          (pleasure ?v)
                          (eats ?n1 ?n2)
                          (food ?n2)
                          (locale ?n1 ?l2)
                          (attacks ?l1 ?l2))
       :effect (and (not (craves ?v ?n1))
                    (craves ?v ?n2)
                    (not (locale ?n1 ?l2))
                    (locale ?n1 ?l1)))
```

```
   (:action succumb
       :parameters (?c ?v ?n ?s1 ?s2)
       :precondition (and (fears ?c ?v)
                          (pain ?c)
                          (pleasure ?v)
                          (craves ?v ?n)
                          (food ?n)
                          (harmony ?v ?s1)
                          (orbits ?s1 ?s2))
       :effect (and (not (fears ?c ?v))
                    (craves ?c ?n)
                    (not (harmony ?v ?s1))
                    (harmony ?v ?s2)))
   (:action drink
      :parameters (?n1 ?n2 ?l11 ?l12 ?l13 ?l21 ?l22)
      :precondition (and (not (= ?n1 ?n2))
                         (locale ?n1 ?l11)
                         (attacks ?l12 ?l11)
                         (attacks ?l13 ?l12)
                         (locale ?n2 ?l21)
                         (attacks ?l21 ?l22))
      :effect (and (not (locale ?n1 ?l11))
                   (locale ?n1 ?l12)
                   (not (locale ?n2 ?l21))
                   (locale ?n2 ?l22)))
)
```

B.4.2 Problem Definition

```
(define (problem strips-mprime-x-4)
   (:domain mystery-prime-strips)
   (:objects muffin ham scallion shrimp cherry grapefruit bacon
             arugula scallop wurst aesthetics hangover dread sciatica
             jealousy loneliness abrasion anger surrey quebec bosnia oregon
             kentucky mars vulcan)
   (:init (food muffin)
          (food ham)
          (food scallion)
          (food shrimp)
          (food cherry)
          (food grapefruit)
          (food bacon)
```

```
(food arugula)
(food scallop)
(food wurst)
(pleasure aesthetics)
(pain hangover)
(pain dread)
(pain sciatica)
(pain jealousy)
(pain loneliness)
(pain abrasion)
(pain anger)
(province surrey)
(province quebec)
(province bosnia)
(province oregon)
(province kentucky)
(planet mars)
(planet vulcan)
(locale cherry kentucky)
(eats ham muffin)
(eats cherry shrimp)
(locale scallion quebec)
(craves dread ham)
(eats cherry ham)
(eats grapefruit scallop)
(craves sciatica grapefruit)
(eats wurst bacon)
(eats muffin ham)
(attacks oregon kentucky)
(eats arugula scallop)
(eats arugula bacon)
(eats bacon wurst)
(eats arugula muffin)
(craves anger wurst)
```

```
(eats scallion shrimp)
(eats arugula wurst)
(locale arugula kentucky)
(eats grapefruit wurst)
(craves loneliness arugula)
(harmony aesthetics vulcan)
(eats muffin cherry)
(eats scallop arugula)
(locale muffin kentucky)
(locale grapefruit surrey)
(craves hangover muffin)
(eats cherry arugula)
(eats shrimp scallion)
(locale ham bosnia)
(eats muffin scallion)
(eats arugula cherry)
(eats scallop grapefruit)
(craves abrasion scallop)
(eats bacon arugula)
(eats ham cherry)
(eats cherry muffin)
(locale bacon quebec)
(locale wurst surrey)
(attacks bosnia oregon)
(locale scallop oregon)
(eats shrimp cherry)
(eats wurst arugula)
(attacks quebec bosnia)
(eats muffin arugula)
(attacks surrey quebec)
(craves aesthetics shrimp)
(eats scallion muffin)
(orbits mars vulcan)
(locale shrimp bosnia)
```

```
        (craves jealousy bacon)
        (eats wurst grapefruit))
(:goal (and (craves sciatica wurst))))
```