LINEAR PLANNING LOGIC AND LINEAR LOGIC GRAPH PLANNER: DOMAIN INDEPENDENT TASK PLANNERS BASED ON LINEAR LOGIC

A DISSERTATION SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER ENGINEERING

By

Sıtar Kortik

September, 2017

Linear Planning Logic and Linear Logic Graph Planner: Domain Independent Task Planners Based On Linear Logic
By Sıtar Kortik
September, 2017

We certify that we have read this dissertation and that in our opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Varol Akman(Advisor)

Uluç Saranlı(Co-Advisor)

Pınar Karagöz

Uğur Güdükbay

Özgür Ulusoy

Faruk Polat

Approved for the Graduate School of Engineering and Science:

Ezhan Karaşan
Director of the Graduate School

ABSTRACT

LINEAR PLANNING LOGIC AND LINEAR LOGIC GRAPH PLANNER: DOMAIN INDEPENDENT TASK PLANNERS BASED ON LINEAR LOGIC

Sıtar Kortik
Ph.D. in Computer Engineering
Advisor: Varol Akman
Co-Advisor: Uluç Saranlı
September, 2017

Linear Logic is a non-monotonic logic, with semantics that enforce single-use assumptions thereby allowing native and efficient encoding of domains with dynamic state. Robotic task planning is an important example for such domains, wherein both physical and informational components of a robot's state exhibit non-monotonic properties. We introduce two novel and efficient theorem provers for automated construction of proofs for an exponential multiplicative fragment of linear logic to encode deterministic STRIPS planning problems in general. The first planner we introduce is Linear Planning Logic (LPL), which is based on the backchaining principle commonly used for constructing logic programming languages such as Prolog and Lolli, with a novel extension for LPL to handle program formulae with non-atomic conclusions. We demonstrate an experimental application of LPL in the context of a robotic task planner, implementing visually guided autonomous navigation for the RHex hexapod robot. The second planner we introduce is the Linear Logic Graph Planner (*LinGraph*), an automated planner for deterministic, concurrent domains, formulated as a graph-based theorem prover for a propositional fragment of intuitionistic linear logic. The new graph-based theorem prover we introduce in this context substantially improves planning performance by reducing proof permutations that are irrelevant to planning problems particularly in the presence of large numbers of objects and agents with identical properties (e.g. robots within a swarm, or parts in a large factory). We illustrate LinGraph's application for planning the actions of robots within a concurrent manufacturing domain and provide comparisons with four existing automated planners, BlackBox, Symba-2, Metis and the Temporal Fast Downward (TFD), covering a wide range of state-of-the-art automated planning techniques and implementations that are well-known in the literature for their performance on various of problem types and domains. We show that even though LinGraph does not rely on any heuristics, it still outperforms these systems for concurrent domains with large numbers of identical objects and agents, finding feasible plans that they cannot identify. These gains persist even when existing methods on symmetry reduction and numerical fluents are used, with LinGraph capable of handling problems with thousands of objects. Following these results, we also formally show that plan construction with LinGraph is equivalent to multiset rewriting systems, establishing a formal relation between LinGraph and intuitionistic linear logic.

Keywords: Automated planning, Linear logic, Multiset rewriting, Automated theorem prover.

ÖZET

DOĞRUSAL PLANLAMA MANTIĞI VE DOĞRUSAL MANTIK GRAFİK PLANLAYICI: DOĞRUSAL MANTIK TABANLI ALAN BAĞIMSIZ GÖREV PLANLAYICILAR

Sıtar Kortik
Bilgisayar Mühendisliği, Doktora
Tez Danışmanı: Varol Akman
İkinci Tez Danışmanı: Uluç Saranlı
Eylül, 2017

Doğrusal mantık, tek kullanımlık varsayımları kullanmaya zorlayan tekdüze olmayan bir mantık olduğu için, dinamik durumlu alanları etkili olarak göstermeye olanak sağlıyor. İçinde, bir robotun durumunda fiziksel ve bilgisel bileşenlerin birlikte tekdüze olmayan özellikler sergilendiği robotik görev planlaması, bu tür alanlar için önemli bir örnektir. STRIPS planlama problemleri için, doğrusal mantıkta ispatları otomatik ortaya çıkaracak iki adet yeni ve etkili teorem ispatlayıcı ortaya koyuyoruz. Ortaya koyduğumuz ilk planlayıcı olan Doğrusal Planlama Mantığı, Prolog ve Lolli gibi programlama dillerinde sıkça kullanılan geriye zincirleme prensibiyle çalışmaktadır ve atomik olmayan sonuçları da ele alacak şekilde genişletilmiştir. Bu yeni planlayıcının deneysel bir uygulaması olan RHex robotu için görsel yönlendirmeyle otomatik gezinme, robotik görev planlayıcı kapsamında gösterilmiştir. Ortaya koyduğumuz ikinci planlayıcı olan Doğrusal Mantık Grafik Planlayıcısı, doğrusal mantık için grafik tabanlı teorem ispaylayıcı olarak formüle edilmis, rastgele olmayan ve eş zamanlı alanlar icin otomatik bir planlayıcıdır. Bu yeni grafik tabanlı teori ispatlayıcı, çoklu sayıdaki özdeş nesnelerin olduğu zamanlarda (sürü icindeki robotlar, büyük fabrikadaki parçalar), özellikler planlama problemleriyle alakası olmayan ispat permütasyonlarını azaltarak planlama performansını arttırıyor. İkinci planlayıcının, eş zamanlı üretim alanında eylem planlaması icin uygulamasını örnek üzerinde gösteriyoruz ve literatürde farklı problem tiplerinde ve alanlarında performanslarıyla bilinen, dört farklı otomatik planlayıcı olan BlackBox, Symba-2, Metis ve Temporal Fast Downward (TFD) ile karsılaştırmasını sağlıyoruz. Yeni planlayıcımızın herhangi bir buluşsala bağlı olmamasına rağmen, diğer sistemleri çoklu özdeş nesnelerin varlığında eş zamanlı alanlarda yendiğini gösteriyoruz. Simetri azaltma ve sayısal akışkanlar ile ilgili mevcut metodlar kullanılsa bile, yukarıdaki kazanımlar sürüyor ve yeni planlayıcımız binlerce nesneli problemleri çözebiliyor. Bu çıkarımlara ek olarak, bu yeni planlayıcı ile plan oluşturmanın, çoklu küme yeniden yazım sistemlerine eşit olduğunu gösteriyoruz.

Anahtar sözcükler: Otomatik planlama, Doğrusal mantık, Çoklu küme yeniden yazımı, Otomatik teorem ispatlayıcı.

Acknowledgement

I thank my advisor Uluç Saranlı for helping me develop and finish this thesis. His vision, knowledge and creativity carried me forward to this day.

I also thank my advisor Varol Akman for being with me from beginning of my MS thesis until this day. His advises and guidance gave me strength throughout these years.

This work would not have been possible without supervision of Frank Pfenning, whose collaboration and advice helped me coming up with the idea of LinGraph during my visit to the Carnegie Mellon University.

I thank my thesis committee, Pınar Karagöz, Uğur Güdükbay, Özgür Ulusoy and Faruk Polat for their support and giving advice for this thesis.

Many people directly or indirectly contributed to my completion of this thesis. I thank the Middle East Technical University Couple Dances Club and all of its members for creating the wonderful social setting in which I have discovered my passion for tango. I thank TUBITAK, the Scientific and Technical Research Council of Turkey for their financial support during my visiting of Carnegie Mellon University. I am also thankful to the Computer Engineering Department of Bilkent University for their financial support during my PhD studies.

I owe my deepest gratitude to my parents and my sister for their endless love supported me through many years of separation.

Contents

1 Introduction 1

1.1 Motivation and Scope . . . 1

1.2 Contributions . . . 5

1.2.1 Linear Planning Logic (LPL) . . . 5

1.2.2 Linear Logic Graph Planner (LinGraph) . . . 6

1.3 Organization of the Thesis . . . 7

2 Background and Related Work 8

2.1 Task (Action) Planning . . . 8

2.2 Intuitionistic Linear Logic: Language and Proof Theory . . . 13

2.2.1 The Grammar and Connectives of Linear Logic . . . 13

2.2.2 Sequent Calculus for Linear Logic . . . 15

2.2.3 Proof Search in Linear Logic . . . 19

3 Linear Planning Logic: An Efficient Language and Theorem Prover for Robotic Task Planning 24

3.1 An Overview of a Resolution System for Linear Hereditary Harrop Formulas (LHHF) . . . 25

3.1.1 Resource Management . . . 25

3.1.2 Backchaining and Residuation in LHHF . . . 27

3.2 The LPL Grammar and Sequent Definitions . . . 28

3.2.1 The LPL Grammar . . . 28

3.2.2 Sequent Definitions of LPL . . . 29

3.3 Proof Theory of LPL . . . 30

3.3.1 Multiplicative Right Rules . . . 30

3.3.2 Right Rules for Exponentials and Quantifiers 31
3.3.3 Resolution in LPL 32
3.3.4 Backchaining and Residuation in LPL 34
3.4 Incorporating Alternative Conjunction 36
3.4.1 Eliminating Nondeterminism Caused by Additive Conjunction (&) 37
3.5 A Task Planning Example With The RHex Hexapod 38
3.6 Action Types in LPL 42
3.7 Expressivity and Efficiency of LPL 43

4 Soundness Proof of Backchaining System for LPL 49
4.1 A Reference Logic System (RL) 49
4.1.1 Proof Annotations for RL 50
4.1.2 Sequent Calculus Proof System for RL 51
4.2 Backchaining System for LPL (BL) 52
4.2.1 Proof Annotations for **BL** 53
4.2.2 LPL Inference Rules with Proof Terms 53
4.3 Proof Rewriting 54
4.3.1 An "Intermediate" Proof System for Proof Transformations 54
4.3.2 Proof Rewriting Rules 56
4.3.3 Proof Rewriting Procedure 57
4.3.4 Proof Rewriting Rules for Soundness 62
4.3.5 Commutativity of Rules within System IL* 65

5 LinGraph: A Graph Based Linear Logic Theorem Prover for Task Planning 81
5.1 Linear Logic Graph Planner: The Language 82
5.2 Assembly Planning Domain Example 85
5.2.1 Domain Definition 85
5.2.2 Encoding the Assembly Planning Domain in LGPL 86
5.3 Linear Logic Graph Planner: Plan Construction 88
5.3.1 GraphPlan 88
5.3.2 Planning with LinGraph 90
5.3.3 A More Challenging Assembly Planning Example 104

5.4 Experimental Results and the Performance of Plan Search 107
5.4.1 Domain-1: Assembly Planning with Two Types of Components 109
5.4.2 Domain-2: Assembly Domain Without Trivial Parallelism . 116
5.4.3 RHex hexapod robot domain 118
5.4.4 Domain-3: Cooperative Multi-Robot Assembly 119
5.5 LinGraph, Multiset Rewriting and Linear Logic 122

6 Conclusion and Future Work **129**

Appendices **133**

A Formal Representation of LinGraph **134**

List of Figures

2.1 Proof rules for multiplicative connectives. 16
2.2 Proof rules for additive connectives 17
2.3 Proof rules for quantifiers 18
2.4 Proof rules for exponentials 18
2.5 Hypotheses 18
2.6 Predicates and actions for the blocks world planning domain. . . . 19
2.7 Encodings of blocks world planning domain actions in linear logic. 19

3.1 Resolution rules for the linear context and the intuitionistic context. 27
3.2 Multiplicative right sequent rules for the **BL** proof theory. 31
3.3 Exponential and quantifier right sequent rules for the **BL** proof theory. 32
3.4 Resolution rules for atomic goals within the LPL backchaining proof theory 33
3.5 Residuation rules for atomic goals within the LPL backchaining proof theory 35
3.6 The alternative conjunction (&) rules for the **BL** proof theory. . . 36
3.7 Predicates and actions for the robotic planning domain. 38
3.8 Encodings of walking and tagging actions within LPL. 39
3.9 A snapshot from the experimental setup in its initial state. Six colored landmarks are scattered throughout the environment, observable through a camera mounted on the RHex robot. 39

3.10 Left: A red-yellow landmark as seen by the robot. Right: blobs extracted from the image with their centers marked with a "+" sign. Computed landmark location is the midpoint of the two blobs and is marked with a white circle. 41
3.11 Snapshots during the execution of the behavioral plan for RHex generated by the LPL theorem prover. 42
3.12 Objects, states and actions of the mail delivery example. 46
3.13 Encodings of the mail delivery actions in LPL. 46
3.14 Incorrect proof attempt using first the $\otimes R$ rule, then the $\otimes L$ rule. 47
3.15 Correct proof using the $\otimes L$ rule first, then the $\otimes R$ rule. 48

4.1 System RL: A reference sequent calculus system for the LPL fragment of intuitionistic linear logic. 51
4.2 Reduction rules with proof annotations for the **BL** proof theory. . 54
4.3 Resolution and residuation rules with proof annotations for the **BL** proof theory. 55
4.4 Proof Rewriting procedure. 56
4.5 System IL: Initial and left sequent calculus rules. 57
4.6 Rewriting rules for proofs ending with **il-** $\multimap$ L, preceded by **il-**$\otimes$L$_{01}$ or **il-**$\otimes$L$_{00}$ on the subgoal (left) branch. 80

5.1 Sequent calculus proof rules for $LGPL$. 84
5.2 Objects types and notation for the robotic assembly planning domain. 86
5.3 An overview of the algorithm for plan search through incremental construction of a LinGraph instance. 90
5.4 Visualization of LinGraph nodes for states, actions and goals, distinguished by their shapes. Each node shows its corresponding LGPL formula, the number of its instances n and its unique label. 91
5.5 Visualization of the LinGraph following the creation of nodes for the initial state and the goal for the planning problem of (5.10). This figure also illustrates initial constraints and the first goal check attempt, which fails since no state nodes match the goal node P. . 93

5.6 LinGraph expansion example illustrating second level nodes created both through copying (s_3 and s_4) from the first level as well as the application of a new instance of the MakeP action (s_5 and s_6). 95
5.7 Final state of the LinGraph for the planning problem of (5.10), showing all constraints used for the last successful goal check step with all goals satisfied. 97
5.8 LinGraph expansion step checking for feasibility of a new instance of the MakeP action through *action constraints*. 100
5.9 Plan extraction using the solution obtained from the constraint solver. Each state node shows the number of utilized resources in the lower right corner. Unused state nodes are faded out for clarity. 101
5.10 Solution to the modified planning problem of (5.12) 101
5.11 Objects and actions encoding of the real world planning example. 105
5.12 Linear logical encodings of actions within the more complex assembly planning problem domain. 105
5.13 Final LinGraph for the complex assembly planning example of (5.13). Some actions and nodes for levels larger than 2 were omitted for visual clarity. Nodes that are part of the final plan are shown in red. 106
5.14 Solution to the complex assembly planning problem defined by (5.13) 107
5.15 Encodings of actions for Domain-1 in LGPL. 109
5.16 PDDL domain file encoding Domain-1, an assembly planning example with two types of components. 109
5.17 Encoding the problem with four components for Domain-1 in PDDL. 110
5.18 PDDL encoding of the durative version of the MakeS1 action for Domain-1 111
5.19 PDDL encoding of Domain-1 using numerical fluents to capture object multiplicity. This figure only shows the definition of the MakeS1 action 112
5.20 An example PDDL problem definition for Domain-1 using numerical fluents, featuring six manipulators to produce a total of three products 112

5.21 New action definition in the PDDL domain file encoding of Domain-2. 117
5.22 The cooperative multi-robot domain consisting of two supply stations, l_1 and l_2 and a base station l_0. This figure also illustrates the initial state for our examples, wherein $2n$ wheels, n body parts and r robots are available in their corresponding stations. Arrows indicate connections between locations traversable by robots. . . . 119
5.23 LGPL encoding of the example for cooperative multi-robot assembly of bicycles. Propositions encoding components of the state, together with LGPL action definitions and their descriptions are given. 120
5.24 The desired final state for the example multi-robot cooperative assembly domain. n fully assembled bicycles and r robots should be located in the base station l_0. 121
5.25 Solution to the multi-robot example encoded by (5.20) with $n = 1$ and $r = 7$, generated by the LinGraph planner. 121
5.26 Sequent calculus proof rules for $LGPL$ 126

List of Tables

5.1 Execution times (in seconds) for Domain-1 with twice the number of initially available manipulators as the number of initial components n. LinGraph solutions have 4 levels 113

5.2 Execution times (in seconds) for Domain-1 with the same number n of initially available components and manipulators. LinGraph solutions have 4 levels 114

5.3 Execution times (in seconds) for Domain-1 with the number m of initially available manipulators a non-integer multiple of the number n of initially available components. LinGraph solutions have 4 levels 114

5.4 LinGraph level, node and constraint statistics for different instances of the planning problem in (5.17). 115

5.5 Node and constraint statistics for each level of the LinGraph solution for a Domain-1 problem instance with $n = 10000$ and $m = 10000$. 115

5.6 Execution times (in seconds) for the problem of (5.18). LinGraph plans have 4 levels 117

5.7 Execution times (in seconds) for the problem of (5.19). LinGraph plans have 4 levels 118

5.8 Created nodes and constraints at each level for the Rhex example in LinGraph. 119

5.9 Execution times (in seconds) for different planners in solving increasingly large instances of the planning problem encoded by (5.20). LinGraph extends to level 6 in all cases, corresponding to 5 steps. Other planners are unable to solve any of the problem instances 122
5.10 Execution times (in seconds) for different planners on the planning problem of (5.20) with fewer manipulators. LinGraph extends to level 10 in all cases (9 steps). Other planners abort with errors beyond a certain problem size 122

Chapter 1

Introduction

1.1 Motivation and Scope

In 1996, Pathfinder successfully landed to the surface of Mars. After landing to the surface, the rover traveled about 100 meters during its 90 day lifetime, more than half of which was spent doing nothing due to plan failure caused by unpredictabilities of resources and other environmental factors while the execution of a given plan [1, 2]. It was not easy to intervene the rover in the existence of a plan failure because of the communication delay between Earth and Mars. Remotely controlling the rover was also very difficult with the same reason. Instead of teleoperation, for each day operators were sending an one-day plan at once such that moving from one location to another or analyzing the surface of Mars. Subsequently, the rover would run the plan sent previous day [3]. A typical one-day plan included approximately 100 actions, which is very long to be executed without any failures. Unfortunately, during the execution of a given plan, there are many uncertainties such as the power required, the duration of each action, the position and orientation of the rover and other environmental factors like dust on solar panels, etc [4]. A rover capable of fully autonomous decision making could have overcome most of these issues. The last rover on Mars, Curiosity [5], has an autonomous navigation mode. However, even this dexterous rover depends on operators on Earth. We can hence say that autonomy for mobile robotic platforms is a difficult, but important challenge. In the near future, it will be possible to see fully autonomous cars on roads and robot assistants in houses. Such autonomous platforms need planners which can effectively and accurately represent the problem domain and make efficient decisions to achieve given real world goals. One of our motivations is the need for such automated and domain-independent planners.

In this context, we focus on task planning (a.k.a. action planning) [6–8]. Task planning is the problem of selecting an ordered list of *actions*, starting with a set of possible *initial states*, to achieve a particular *goal state*. Physical behavior of a particular system associated with an action is summarized through a list of *preconditions* and a list of *effects* for the action, providing a discrete abstraction of system behavior. To be able to use an action, all preconditions must be satisfied. When an action is used, the environment is modified to satisfy the effects of the action. This conceptual framework assumes a discretized view of time. Task planning seeks a plan through these abstract behaviors, manipulating the system from its initial state to the desired final state. The final plan is either a partial or total order of selected actions [9].

Automatic task planning has been among important problems in Artificial Intelligence, with numerous applications in robotics, scheduling, resource planning and even automated programming. Different approaches have been proposed to represent and solve task planning problems. Among these methods, linear logic [10, 11] has been proposed in the literature, capable of handling real world planning problems [12], since linear logic allows native support for representing dynamic state as single-use (or *consumable*) resources. Consumable resource representation in linear logic stands in contrast to fact representation in classical logic that can be used multiple times or not at all. Together with consumable resources, linear logic can also represent persistent resources that can be used infinitely many times whenever they are needed. Resource representation of linear logic is capable of effectively addressing the well-known frame problem that often occurs in embeddings of planning problems within logical formalisms such as situation calculus [13–16]. Briefly, the *frame problem* is the need and challenge of representing possibly irrelevant non-effects of an action in addition to its more relevant effects.

Our main motivation in this thesis is the exploration of logical approaches with theorem proving for task planning. Unfortunately, using logical reasoning methods and theorem provers in task planning generally runs into complexity issues because of the multitude of valid proofs (all corresponding to the same plan) for the same problem. This fact causes nondeterminism beyond what is inherent in the problem domain itself, and makes the search problem much more difficult [17]. Addressing this issue and introducing an expressive and efficient task planner is challenging. Most research efforts either simplify the representational language, hence sacrificing expressivity, or impose a particular structure on proof search that might impair soundness and completeness of the logic [18].

In this thesis, we propose two different languages and associated theorem provers for task planning using linear logic. The first language that we introduce is Linear Planning Logic (*LPL*), which extends on Linear Hereditary Harrop formulas (*LHHF*) that underlie linear logic programming languages such as Lolli [19]. LPL provides increased expressivity compared to Lolli, while managing to similarly keep out-of-domain nondeterminism to a minimum. Expressivity is increased by allowing negative occurrences of conjunction, which as necessary to represent multiple simultaneous preconditions for an action. Adding negative occurrences of conjunction while still preserving a backchaining method for proof search is problematic for linear logic. The proof theory that we propose for this language preserves the deterministic back-chaining structure of provers designed for LHHF. To show expressivity and feasibility of our planning formalism, we illustrate an experimental application for a hexapod robot RHex [20], navigating in an environment populated with visual landmarks.

Our second proposed language for automated generation of plans for STRIPS-based [21] planning problems is Linear Logic Graph Planner (*LinGraph*), which is a graph-based theorem prover for a multiplicative exponential fragment of propositional linear logic. LinGraph reduces nondeterminism particularly in the

presence of large numbers of functionally identical resources and agents. We introduce a number of different planning domains such as manufacturing to illustrate the application of LinGraph for planning the actions of agents. We also compare LinGraph to four modern planners, four existing automated planners, BlackBox, Symba-2, Metis and the Temporal Fast Downward (TFD), covering a wide range of state-of-the-art automated planning techniques and implementations that are well-known in the literature for their performance on various of problem types and domains.. We show that for specific domains where there are large numbers of identical objects and agents, LinGraph can outperform these systems, finding feasible plans that they cannot identify.

1.2 Contributions

In this section, we summarize main contributions of the thesis in two categories. For each category, we itemize and summarize individual contributions.

1.2.1 Linear Planning Logic (LPL)

- Introducing a novel logic language and theorem prover:
  One of the main contributions of this thesis is introducing a novel logic language and an associated theorem prover for task planning. We call our new language Linear Planning Logic (LPL), consisting of a fragment of linear logic well suited for representing dynamic states and constructing automatic plans efficiently.

- Extending the expressivity of Linear Hereditary Harrop Formulas (LHHF):
  LPL extends on the expressivity of Linear Hereditary Harrop Formulas (LHHF) by adding negative occurrences of conjunction, allowing actions to create multiple resources.

- Implementing LPL in SWI-Prolog:
  In order to show performance of LPL, we implemented it in SWI-Prolog.

- Introducing a new robotic planning domain example:
  To test the implementation and show the expressivity of LPL, we introduced a new robotic planning domain example and encoded a number of example planning problems in LPL. We have shown both the correctness and performance of our LPL theorem prover for this domain.

- Using LPL to construct plans for the hexapod platform RHex
  After showing the expressivity of LPL on simple planning domains, we used the hexapod platform RHex for an experimental demonstration of its use.

1.2.2 Linear Logic Graph Planner (LinGraph)

- Introducing a graph-based theorem prover for task planning:
  The second main contribution of this thesis is introducing a graph based theorem prover for task planning, within a propositional fragment of linear logic. We call this framework Linear Logic Graph Planner (LinGraph).

- Implementing LinGraph in SML:
  We implemented a prototype of LinGraph in SML to compare its performance with other planners.

- Introducing additional new planning domain examples:
  To show expressivity of LinGraph, we introduced additional planning domain examples and encoded these domains in our implementation.

- Comparing the performance of LinGraph with other planners:
  Another contribution of this thesis is comparing the performance of LinGraph with other modern planners, including BlackBox, Symba-2, Metis and the Temporal Fast Downward (TFD) and showing that LinGraph outperforms other planners for certain planning domains in the presence of large numbers of identical objects and agents.

- Establishing a formal relation between LinGraph and linear logic:
  We also formally show that plan construction with LinGraph is equivalent to multiset rewriting systems, establishing a formal relation between LinGraph and intuitionistic linear logic.

1.3 Organization of the Thesis

We will begin by reviewing related literature in Chapter 2, followed by a definition of the fragment of Linear Logic we use for encoding planning problems in Chapter 3. We then present soundness proof of LPL in Chapter 4. In Chapter 5, we introduce LinGraph and an assembly planning domain to present a detailed description of our method, followed by conclusion and future work in Chapter 6.

Chapter 2

Background and Related Work

2.1 Task (Action) Planning

Task, or action planning is finding an ordered sequence of *actions* (also called a *plan*), which will take an agent from a set of possible *initial states* to a particular *goal state*. Automation of finding such plans has been among important problems in Artificial Intelligence and can be used for numerous applications in robotics, scheduling, resource planning and even automated programming. Many successful automated planners are based on a discretized view of task planning, which are based on composing continues, *low-level* behavioral primitives through their *high-level* abstractions. This structure has been as a basis for numerous languages and methods. Among these, the STRIPS language [21] is one of the most basic but fundamentally useful modeling languages. More capable alternatives to STRIPS such as ADL [22, 23] and the more recent PDDL [24] eliminate these limitations and recover substantial expressivity. However, the majority of real-world implementations of automated planners still rely on internal translations to a STRIPS representation, that are exponential in complexity and hence do not eliminate nondeterministic components inherent in the planning problem itself.

We can categorize automated task planning systems according to the configurability of the planner to work in different planning domains [25]. As such, we can talk about domain-specific, domain-independent and domain-configurable planners. Among these, domain independent planners are those that are capable of working in any problem domain. Domain-independent planners [26] have mainly two varieties, state-space and plan-space planners. However, this categorization may not be enough to cover and denote all relevant properties of a planning system. Another categorization could be according to the step-optimality of a planner. Within this classification, our research focuses on domain-independent and step-optimal planning.

Among existing planners, partial-order planning (POP) implements plan-space search rather than state-space search, and eliminates unnecessary nondeterminism in the search for plan search originating from irrelevant total orderings of action sequences [27]. The Universal Conditional Partial Order Planner (UCPOP) method [28] is an important variation of POP and achieves greater expressivity by allowing actions that have variables, conditional effects, disjunctive preconditions and universal quantification. One of more recent optimal planners for STRIPS problems is GraphPlan. Even though GraphPlan has reduced expressivity, it is a practical alternative to UCPOP with better performance [29]. A number of recent extensions and improvements for GraphPlan were been introduced in the literature. In [30], the authors try to combine the expressivity of UCPOP with the efficiency of GraphPlan by transforming domains in UCPOP to their equivalent in GraphPlan. In [31], GraphPlan is extended to contingent planning problems incorporating uncertainty and sensing actions. In [32], the authors explore the use of GraphPlan for probabilistic planning, with a known world but probabilistic actions. A number of introduced planners support nondeterministic actions using GraphPlan's features. One of such planners is the Fast, Iterative Planner (FIP), illustrated for a simple omelette problem [33]. It has been capable of outperforming other planners, including the model-based MBP system [34] for nondeterministic domains in certain cases. In [35], the authors extend the classic GraphPlan algorithm to support dynamic creation and destruction of objects through new operators specifically introduced for this purpose. Another method formulates planning problems as Constraint Satisfaction Problems (CSP) instances [36], claiming that this structure subsumes the representation induced by GraphPlan and leads to a more efficient planner. This also relies on imposing a bound on the length of the plan, reducing search complexity to NP.

Among most successful recent modern optimal task planners, Blackbox [37, 38] and SatPlan implementations [38–41] convert problems specified in STRIPS notation into satisfiability problems. Formulating planning as a satisfiability problem has been introduced by [42], where the central idea is to translate the planning problem into a finite collection of axioms, whose satisfaction corresponds to the solution.

Even though these planners are among the most successful optimal task planners, their plan search complexity is NP-hard [43]. Modern planners based on *heuristic search planning* (HSP) address this issue of complexity by using a *relaxed problem*, which ignores negative effects of actions and resulting mutexes, to compute the heuristic [44–46]. However, these planners based on HSP sacrifice optimality for efficiency. One of the well-known planners based on this idea is Fast-Forward (FF) planner [47]. A more recent state of the art non-optimal planner is Madagascar [48, 49], capable of quickly finding feasible plans. Another alternative approach for non-optimal planning is using a hierarchical structure for plan search. Hierarchical Task Network (HTN) planning [50, 51] is based on this approach. Initially a high-level task network consists of initial and final states, with high-level abstract tasks, which are then iteratively decomposed until primitive actions are reached.

Another, preceding these modern planners, earlier approaches to task planning used deductive reasoning and theorem proving. However these methods lost their popularity in recent decades, since nondeterministic choices introduced by logical encodings of problem domains often increase the computational complexity of plan search as exemplified by classical logic encodings. Representing dynamic states is also problematic in classical logic. For example, situation calculus attempts to represent dynamic states within a planning problem [52] but it suffers from the *frame problem*, resulting in substantially increased complexity [53].

In this context, linear Logic [10, 11] is an effective solution to the frame problem, allowing native modeling of dynamic states as single-use (consumable) resources within the language itself. Hereby, linear logic eliminates the complexity cost associated with complex solutions to the frame problem [15, 16, 54]. Linear logic's non-monotonic reasoning in solving the frame problem was also used in other planners, using logic programming and answer sets [14, 55–57]. When using linear logic to encode and solve planning problems, we can compose states using the *simultaneous conjunction* operator, represent actions with the *linear implication* connective and model nondeterministic effects with the *additive disjunction* connective [12, 17, 58, 59]. Linear logic can also model concurrency as explored in [60, 61]. Even though linear logic has these appealing properties, construction of efficient linear logic theorem provers having these properties has been challenging. One of the main issues arises from the *logical nondeterminism*, introduced by explicit encodings of planning problems through the use of logical connectives. A number of theorem provers such as Lolli restrict the expressivity of the logical language to increase efficiency [19].

Petri Nets can successfully represent concurrent processes and have semantic connections to linear logic [62–65]. The PetriPlan system uses this connection and transforms the planning problem into a reachability query in an acyclic Petri Net, and then uses Integer Programming to find a valid solution.

Another technique for task planning is to use model-based reasoning [34]. In this method, plan construction to solve planning problems is transferred into the problem of searching through a logical semantic structure, without being connected to an explicit proof theory. One of the popular applications of this idea is the use of temporal logic [18]. This method abstractly represents continuous behaviors as atomic propositions and models temporal components through the evaluation of a finite state machine. However, some properties of this method such as its dependence on temporal logic, model checking and abstraction of the physical world decrease expressivity and do not allow deductive reasoning.

We can use a working domain-independent planner to solve the planning problem in different environments and domains such as manipulation planning, assembly line planning and mobile robot planning. The manipulation planning problem usually includes a manipulator robot which interacts with an object, grasping and moving the object to another location in the environment. Among most efficient control architectures for robot manipulators, behavior-based methods stand out for practical applications. However behavior-based methods either require infeasible computation time or do not ensure completeness. To solve these problems, in [66] the authors present an algorithm for manipulation planning. This algorithm plans paths in configuration spaces with multiple constraints such as torque limits, constraints on the pose of an object held by a robot, and constraints for following workspace surfaces. Another manipulation planning method is presented in [67], where the authors introduce a formal tool which they call the *motion grammar*, for task decomposition and hybrid control. This work is interesting in issue of a context free grammar for manipulation planning. They test this formal tool on a chess game with a manipulator robot.

Another application area of robotic planning is assembly line planning. In [68], the author addresses the problem of designing a distributed, hybrid factory given a description of an assembly process and a palette of controllers for basic assembly operations. Another important point of this work is incorporating petri nets into the planning system.

Finally, we briefly mention an existing drawback of efficiency issues in planning, *loop*, non terminating proof search while constructing a plan. To prevent loops, we need a loop detection and a loop preventing mechanism. In [69], the authors provide a loop detection mechanism for propositional affine logic adapting the history techniques used for intuitionistic logic. This work is good for understanding the loop detection mechanism and how it is used in linear logic. Although loops in proof search may cause efficiency issues, in some cases loops can be necessary in planning. In [70] and [71], a different approach for planning is introduced, where they use loops in planning to deal with an unknown and unbounded planning parameter.

2.2 Intuitionistic Linear Logic: Language and Proof Theory

In this section, we present the basic ideas and principles underlying intuitionistic linear logic and show its expressivity on a well-known planning example, the blocks world. *Linear logic* was first introduced by Girard [10] as a reasoning language that rejects the contraction and weakening rules. Linear logic can be considered both in *intuitionistic* and *classical* formulations. In this thesis, we focus on intuitionistic linear logic, since *proofs* in *intuitionistic* languages correspond to executable *programs* as formalized by the Curry-Howard isomorphism [72]. This makes it possible to use a proof generated by an automatic theorem prover as a solution to a particular planning problem, assuming that a suitable, adequate encoding of the domain, including its initial state, goal state and actions, is provided. However, we must note that the more expressive a logical language is, the less efficient associated theorem provers become, since the number of alternatives to consider increase with each step of the proof construction. Fortunately, linear logic promises to support efficient proof search, together with sufficient expressivity to be used as an automatic tool to solve planning problems. The expressivity of linear logic can handle real world planning problems, being able to model dynamic state components as consumable resources. Its consideration of assumptions as single-use resources allows a natural encoding of planning problem domains and states.

2.2.1 The Grammar and Connectives of Linear Logic

In this section, we present the grammar for the full language of intuitionistic linear logic and then describe all of its connectives. We can formally describe with the following grammar:

$$A := a \mid 1 \mid A \otimes A \mid T \mid 0 \mid A \,\&\, A \mid A \multimap A \mid A \supset A \mid \forall x.A \mid A \oplus A \mid !A \mid \exists x.A \ ,$$

where a denotes atomic formulae defined according to the specifics of a particular application domain through an associated term grammar in the usual way.

Below, we briefly introduce all connections of linear logic, explaining the meaning of each in the context of task planning problems.

- Simultaneous Conjunction ($\otimes$):
  We can pronounce $A \otimes B$ as A *and* B or A *tensor* B, meaning that both A and B are simultaneously available. In the task planning domain, this connective is used for composing resources that exist simultaneously in a state, either as resources or as goals. For instance, if we a coffee and a chocolate at the same time, we can represent this state with simultaneous conjunction as (*coffee* $\otimes$ *chocolate*).

- Unit ($\mathbf{1}$):
  Unit is the trivial resource which can be produced from nothing, without using any resources. Unit is the identity of simultaneous conjunction where $\mathbf{1} \otimes A \equiv A$. In planning problems, we can use $\mathbf{1}$ for the empty resource, meaning that we can delete $\mathbf{1}$ whenever we see it in resources.

- Alternative Conjunction (&):
  The connective & is pronounced as *additive conjunction*, *with* or *internal choice*. If we have $A \& B$, we can conclude A or B with the same resources, but not both simultaneously. For example, if we have 1 Dollar and the price of a tea or a chocolate is 1 Dollar, we can say *tea* & *chocolate*, meaning that we can buy a tea or a chocolate, but not both at the same time.

- Top ($\top$):
  We can also call $\top$ as *truth*. Regardless of available resources, we can always achieve Top. If there are a number of resources, Top consumes all of them. Top is the identity of alternative conjunction such that $\top \& A \equiv A$.

- Linear Implication ($\multimap$):
  $A \multimap B$ is pronounced as A *linearly implies* B or A *lolli* B, meaning that we can achieve B by using A exactly once (no more, no less). In the task planning domain, linear implication connective is used for relating preconditions to effects for individual actions.

- Disjunction ($\oplus$):
  $\oplus$ is also called *external choice*. If existing resources can make either A or B true, then we can present as $A \oplus B$. Considering an example in planning problem, if we can have a pie or a tart in a restaurant according to availability, we encode this as *pie* $\oplus$ *tart*.

- Impossibility ($\mathbf{0}$):
  Impossibility corresponds to impossible resource. We can conclude any goals with $\mathbf{0}$. Note that, it is the identity of disjunction, $\mathbf{0} \oplus A \equiv A$. In planning problems, we do not need $\mathbf{0}$.

- Bang (!):
  We can also call the ! operator as *Of Course Modality*, creating unrestricted resources when placed in front of hypotheses. For example, if we have unlimited coffee, we can show this expression with the unary operator as *!coffee*.

- Unrestricted Implication ($\supset$):
  We pronounce $A \supset B$ as A *implies* B, correspond to $(!A) \multimap B$, meaning that we can achieve B by using A as many times as we want.

2.2.2 Sequent Calculus for Linear Logic

We formalize proof construction within linear logic using *sequent calculus*, which was originally developed by Gentzen as a tool for studying natural deduction [73]. Below, we present a general sequent definition for linear logic, encoding the provability of a goal by using a set of *ephemeral resources* and a set of *persistent resources*.

$$\Gamma; \Delta \Rightarrow G \ , \tag{2.1}$$

where Γ is a multiset used to denote persistent resources (unrestricted assumptions) that can be used either or multiple times or none at all, while Δ is a multiset containing resources that must be consumed (linear assumptions). We can prove the goal formula G using the resources in Γ and Δ.

For each connective in sequent calculus formulations, there should be *left* and *right* inference rules. Right rules in sequent calculus (corresponding to introduction rules in natural deduction) decompose a particular goal G, while left rules in sequent calculus (corresponding to elimination rules in natural deduction) decompose resources. Below, we present all left and right rules for linear logic sequent calculus.

We group related rules together such that *multiplicative connectives*, *additive connectives*, *quantifiers*, *exponentials* and *structural rules*. We present the first group, *multiplicative connectives*, in Figure 2.1, including $\multimap$, $\otimes$ and $\mathbf{1}$. The $\multimap R$ rule proves the conclusion by adding the resource A into available resources. The $\multimap L$ rule first proves the goal A, then proves the conclusion while splitting available resources as necessary. The $\otimes R$ rule shows how to achieve a conjunctive goal by decomposing available resources, while the $\otimes L$ rule defines how a conjunctive assumption can be decomposed. The $\mathbf{1}R$ rule says that the trivial resource *unit* can be produced from nothing. Using the $\mathbf{1}L$ rule, we can delete $\mathbf{1}$ whenever we see it in resources.

$$\frac{\Gamma; \Delta, A \Longrightarrow B}{\Gamma; \Delta \Longrightarrow A \multimap B} \multimap R \qquad \frac{\Gamma; \Delta_1 \Longrightarrow A \qquad \Gamma; \Delta_2, B \Longrightarrow G}{\Gamma; \Delta_1, \Delta_2, A \multimap B \Longrightarrow G} \multimap L$$

$$\frac{\Gamma; \Delta_1 \Longrightarrow A \qquad \Gamma; \Delta_2 \Longrightarrow B}{\Gamma; \Delta_1, \Delta_2 \Longrightarrow A \otimes B} \otimes R \qquad \frac{\Gamma; \Delta, A, B \Longrightarrow G}{\Gamma; \Delta, A \otimes B \Longrightarrow G} \otimes L$$

$$\frac{}{\Gamma; \cdot \Longrightarrow 1} 1R \qquad \frac{\Gamma; \Delta \Longrightarrow G}{\Gamma; \Delta, 1 \Longrightarrow G} 1L$$

Figure 2.1: Proof rules for multiplicative connectives.

Additive connectives are presented in Figure 2.2, including $\&, T, \oplus$ and $\mathbf{0}$. The $\&R$ rule decomposes the goal and proves both subgoals G_1 and G_2 using the same resources. We need two left rules for the & connective, $\&L_1$ and &Ł$_2$, selecting one of the sub-resources, A or B respectively, to prove the goal G. The $\top R$ rule consumes all resources. There is not a left rule for T. The $\oplus$ connective has two right rules. The $\oplus R_1$ rule selects the first sub-goal A to prove, while the $\oplus R_2$ rule selects the second sub-goal B to prove. The $\oplus L$ rule decomposes the $A \oplus B$ resource and then tries to prove the goal G in two different ways, one is adding A and the other one is adding B to resources. There is not a left rule for **0**, while we can conclude any goals using the $\mathbf{0}L$ rule.

$$\frac{\Gamma; \Delta \Longrightarrow G_1 \qquad \Gamma; \Delta \Longrightarrow G_2}{\Gamma; \Delta \Longrightarrow G_1 \& G_2}\ \& R$$

$$\frac{\Gamma; \Delta, A \Longrightarrow G}{\Gamma; \Delta, A\&B \Longrightarrow G}\ \& L_1 \qquad \frac{\Gamma; \Delta, B \Longrightarrow G}{\Gamma; \Delta, A\&B \Longrightarrow G}\ \& L_2$$

$$\frac{}{\Gamma; \Delta \Longrightarrow T}\ TR \qquad \text{No T left rule}$$

$$\frac{\Gamma; \Delta \Longrightarrow A}{\Gamma; \Delta \Longrightarrow A \oplus B}\ \oplus R_1 \qquad \frac{\Gamma; \Delta \Longrightarrow B}{\Gamma; \Delta \Longrightarrow A \oplus B}\ \oplus R_2$$

$$\frac{\Gamma; \Delta, A \Longrightarrow G \qquad \Gamma; \Delta, B \Longrightarrow G}{\Gamma; \Delta, A \oplus B \Longrightarrow G}\ \oplus L$$

$$\text{No } \mathbf{0} \text{ right rule} \qquad \frac{}{\Gamma; \Delta, \mathbf{0} \Longrightarrow G}\ \mathbf{0}L$$

Figure 2.2: Proof rules for additive connectives

We present proof rules for quantifiers in Figure 2.3. The $\forall R^a$ rule and the $\exists L^a$ rule postpone instantiating the variable x, replacing x with a parameter a. The new parameter a is a fresh variable and can not exist before. In the $\forall L$ rule and the $\exists R$ rule, a term t is supplied and substituted for the variable x.

In Figure 2.4, proof rules for exponentials are presented. The $\supset R$ rule adds A to unrestricted resources and proves the goal G. In the $\supset L$ rule, we first need to prove A using only unrestricted resources, and then we prove the goal G using the resource B.

Finally, we present hypotheses, the *init* rule and the *copy* rule in Figure 2.5. The *init* rule connects atomic assumptions to atomic goals. This rule occurs

$$\frac{\Gamma; \Delta \Longrightarrow [a/x]G}{\Gamma; \Delta \Longrightarrow \forall x.G}\ \forall R^a \qquad \frac{\Gamma; \Delta, [t/x]A \Longrightarrow G}{\Gamma; \Delta, \forall x.A \Longrightarrow G}\ \forall L$$

$$\frac{\Gamma; \Delta \Longrightarrow [t/x]G}{\Gamma; \Delta \Longrightarrow \exists x.G}\ \exists R \qquad \frac{\Gamma; \Delta, [a/x]A \Longrightarrow G}{\Gamma; \Delta, \exists x.A \Longrightarrow G}\ \exists L^a$$

Figure 2.3: Proof rules for quantifiers

$$\frac{(\Gamma, A); \Delta \Longrightarrow G}{\Gamma; \Delta \Longrightarrow A \supset G}\ \supset R \qquad \frac{\Gamma; \cdot \Longrightarrow A \qquad \Gamma; \Delta, B \Longrightarrow G}{\Gamma; \Delta, A \supset B \Longrightarrow G}\ \supset L$$

$$\frac{\Gamma; \cdot \Longrightarrow A}{\Gamma; \cdot \Longrightarrow !A}\ !R \qquad \frac{(\Gamma, A); \Delta \Longrightarrow G}{\Gamma; (\Delta, !A) \Longrightarrow G}\ !L$$

Figure 2.4: Proof rules for exponentials

at the top leaves of the proof tree whose nodes are instantiations of left and right sequent rules. The *copy* rule copies a persistent resource into the list of consumption resources.

$$\frac{}{\Gamma; A \Longrightarrow A}\ init \qquad \frac{(\Gamma, A); (\Delta, A) \Longrightarrow G}{(\Gamma, A); \Delta \Longrightarrow G}\ copy$$

Figure 2.5: Hypotheses

2.2.3 Proof Search in Linear Logic

In this section, we review the literature and methods for proof search in linear logic in the context of a planning example. To this end, we start with reviewing a well-known planning domain example, the Blocks World [74]. This domain features a number of blocks that are stacked on a table and a robotic arm capable of picking up blocks and placing them on top of other blocks or on the table. The goal is to order blocks vertically in a given order. We present all necessary predicates and actions for blocks world domain in Figure 2.6.

Predicates:		
empty	:	The robot arm is empty
table	:	Each block can be on the table
$on(X, Y)$	:	The block X is on the block Y
$clear(X)$	:	Top of the block X is clear
$holds(X)$	:	The robot arm holds the block X
Actions:		
pickOn(X,Y)	:	The robot arm picks the block X standing on Y.
putOn(X,Y)	:	The robot arm puts the block X on Y (a block or table).

Figure 2.6: Predicates and actions for the blocks world planning domain.

Subsequently, we associate these predicates and actions with linear logic expressions to formally represent their preconditions and effects. Encodings of two actions, pickOn(X,Y) and putOn(X,Y), are given in Figure 2.7. Here, X and Y correspond to particular objects in the domain. When using the pickOn(X,Y) action, if the robot's hand is empty, the X block is on the Y block and top of the X is clear, the robot holds X. Using the other action putOn(X,Y), if the robot is holding the X block, the robot places X onto Y, where Y can be either a block or the table.

pickOn(X,Y)	:	$empty \otimes on(X, Y) \otimes clear(X) \multimap holds(X) \otimes clear(Y)$
putOn(X,Y)	:	$holds(X) \otimes clear(Y) \multimap empty \otimes on(X, Y) \otimes clear(X)$

Figure 2.7: Encodings of blocks world planning domain actions in linear logic.

Now that we have given formal descriptions of actions for the blocks world planning domain, we can illustrate the use of linear logic for task planning with a simple scenario. Assume that there are initially three blocks a, b and c such that a is on b while b and c are on the table. The robot's hand is initially empty. Formally, we can encode the initial state of this scenario by the resource

$$\Delta = on(a,b) \otimes clear(a) \otimes on(b,table) \otimes on(c,table) \otimes clear(c) \otimes empty \ .$$

If we want to achieve the final desired state of the blocks as b is on the table and a is on c while c is on the table, we can encode the final state as

$$G = on(b,table) \otimes clear(b) \otimes on(a,c) \otimes clear(a) \otimes on(c,table) \otimes empty \ .$$

To achieve the subgoal $on(a,c)$, the robot must hold a and then put it on c to conclude the plan. The final plan for this example is the sequence of actions [pickOn(a,b), putOn(a,c)].

We should note that, using *classical logic* for encoding and solving of planning problems may cause inconsistency. For an example, if we would try to encode the pickOn(X,Y) action in *classical logic* for the given blocks world domain example, the new action encoding would be

$$\forall x.\forall y.(empty \wedge on(X,Y) \wedge clear(X) \supset holds(X) \wedge clear(Y)) \ .$$

When we use the pickOn(X,Y) action as the given encoding, we could derive contradictory propositions such that ($empty \wedge$ holds(X)), where they can not be true at the same time.

Even though the introduced example has a small domain size, unfortunately usual real world problems have much larger domain sizes. At this point, searching proofs efficiently plays an important role. To this end, we need to prune the proof search tree as much as possible, eliminating unnecessary nondeterminism. We categorize proof search techniques two main groups, according to decomposing order of the given theorem. The first proof search category is *backward proof search*, first decomposing the goal and then trying to prove all subgoals. The second proof search category is *forward proof search*, first creating all possible variations of resources and then trying to reach the goal from list of created resources.

2.2.3.1 Backward (or Bottom-Up) Proof Search

In *backward proof search*, we decompose the given goal to subgoals and then prove all subgoals using resources. This approach is *goal-oriented*, since we start proof search with a given goal sequent, refining the goal using inference rules in backward direction until having only axioms (atomic propositions) in the goal and resources. If we apply inference rules arbitrary, a number of nondeterminism may arise due to wrong choices of inference rules.

Focusing method [75] reduces nondeterminism of disjunctive choices in proof search, categorizing connectives according to inversion and determining an order for applying inference rules. Main idea of focusing depends on inversion of inference rules. We already know that we can conclude the conclusion from premises in natural deduction and sequent calculus (top-down). An inference rule of a connective is invertible, if we can also conclude premises from the conclusion (bottom-up). In focusing method, we first apply invertible rules (no matter right or left invertible rules), decomposing the formula to sub-formulas until the goal is no longer invertible and resources are non-invertible propositions. Then we focus on a non-invertible proposition in resources (left) or the goal (right). Subsequently, we decompose the focused proposition until reaching either an invertible connective or an atomic proposition.

We can also call invertible connectives *asynchronous* and non-invertible connectives *synchronous*, as Andreoli called in [75]. We can separate all connectives into two main groups, *negative* and *positive*. If the right inference rule of a connective is invertible, then it is a negative connective. If the left inference rule of a connective is invertible, then it is a positive connective. Below we present all negative and positive connectives.

$$\begin{aligned} Negative\,Connectives &: A \multimap B,\, A \& B,\, T\,, \\ Positive\,Connectives &: A \otimes B,\, 1,\, !A,\, A \oplus B,\, 0\,. \end{aligned}$$

We should note that we can't incorporate atomic formulas here, since they do not have any left or right rules.

In linear logic proof search, we should also consider resource management while splitting resources among parallel goals to eliminate a significant amount of non-determinism. The need for resource management specially arises for non-deterministic choices to split resources while applying the right rule of $\otimes$ and the left rule of $\multimap$. In the worst case, all possible alternatives during splitting resources must be exhaustively searched which is 2^n. Fortunately, using *Input / Output model (lazy splitting)* can solve these nondeterministic decisions [19]. In this model, each sub sequent partially consumes resources stored in their input context and returns left over resources in their output context. However, if the truth constant $\top$ is in the goal, then an additional nondeterminism may arise. The presence of T in the goal allows consumption of an arbitrary number of input resources. In the worst, there are 2^n possible alternatives. In [76], this problem is solved by introducing an additional flag, considering the presence of such a flexible resource splitting in the later stages of proof search. We use a similar approach in our LPL system.

2.2.3.2 Forward (Top-Down) Proof Search

In contrast to backward proof search methods like tableaux method or *uniform proofs* [77], first decomposing the goal into subgoals and then proving subgoals, forward proof search methods like resolution or inverse method are top-down approaches [78], using initial assumptions and rules to produce new assumptions and then checking if the given goal can be reached from the created set of resources. In forwards proof search, due to the nature of top-down approach, we don't need resource management anymore as in backward proof search for multiplicative conjunction ($\otimes$) or linear implication ($\multimap$).

Among the forward proof search methods, *focused inverse method* introduced in [79] is a practical forward reasoning method. Focused inverse method successfully composes focusing [75] and inverse method, while presenting that the focused inverse method is notably faster than the non-focused version of inverse method.

We must note that, the way of proof construction in forward reasoning is corresponding to Graphplan planner [29], searching plans through initial level to last level. Since our novel theorem prover and planner, LinGraph, searches proofs in the same direction with Graphplan, we can conclude that LinGraph is a forward proof search method. On the other hand, our other novel theorem prover and planner, LPL, searches proofs in a backward reasoning style, since proof search is goal-directed.

Chapter 3

Linear Planning Logic: An Efficient Language and Theorem Prover for Robotic Task Planning

In this chapter, we introduce one of our main contributions in the thesis, Linear Planning Logic (LPL). LPL is a novel logic language and theorem prover for robotic task planning. It is a fragment of intuitionistic linear logic, whose resource-conscious semantics are well suited for reasoning with dynamic state, while its structure admits efficient theorem provers for automatic plan construction. We can see LPL as an extension of Linear Hereditary Harrop Formulas (LHHF), increasing expressivity while keeping nondeterminism in proof search to a minimum by using a backward proof search strategy and backchaining [19, 76]. At the end of this chapter, we present the main ideas behind LPL and our associated theorem prover on an example problem domain of behavioral planning for the hexapod robot RHex, navigating in an environment populated with visual landmarks.

3.1 An Overview of a Resolution System for Linear Hereditary Harrop Formulas (LHHF)

In this section, we present a multiplicative fragment of Linear Hereditary Harrop Formulas (LHHF) and the efficient backchaining resolution system described in [76], which also inspired our resolution system for LPL. The grammar for LHHF is given as

$$\begin{array}{lrcl} \text{Program formulas:} & D & ::= & a \mid G \multimap D \mid G \supset D \mid \forall x.\, D\ , \\ \text{Goal formulas:} & G & ::= & a \mid G_1 \otimes G_2 \mid D \multimap G \mid D \supset G \\ & & & \mid \forall x.\, G \mid a_1 \doteq a_2 \mid !G \mid \exists x.\, G\ , \end{array}$$

where a denotes atomic formulas. We should note that, LHHF does not include simultaneous conjunction ($\otimes$) in program formulas, reducing expressivity in order to eliminate nondeterminism caused by this connective. Consequently, the multiplicative unit $\mathbf{1}$ is also not needed in program formulas. In the following sections, we describe important features of theorem provers constructed for LHHF.

3.1.1 Resource Management

Even though consumable resources in linear logic increase the expressivity of the logical language, they also result one of the main sources of non-determinism. This issue is called the *resource management problem*, arising from the need to decide on a correct splitting of the resource context while processing multiplicative goals. We can illustrate this problem on an example for proving $G_1 \otimes G_2$ by using consumable resources in the context where $\Delta = \{\Delta_1, \Delta_2\}$. The example is given as

$$\frac{\Gamma; \Delta_1 \Longrightarrow G_1 \qquad \Gamma; \Delta_2 \Longrightarrow G_2}{\Gamma; \Delta \Longrightarrow G_1 \otimes G_2} \otimes R\ .$$

In this example, while proceeding with the bottom-up proof search, we do not initially know how to split Δ into the two desired sub-contexts, Δ_1 and Δ_2, unless we try all possible partitionings. If we assume that there are n formulas in the linear context, in the worst case, finding a correct split of the initial context Δ will require trying all 2^n possibilities.

In [19], Hodas and Miller introduced a deterministic solution for the resource management problem, which is known as the *Input/Output model* or the *I/O model*. This model allows lazy splitting of resources between each conjuncted goal, eliminating the kind of non-determinism caused by the resource management problem. The main idea behind their approach is proving the first subgoal using necessary resources from the initial resources, subsequently proving the other subgoal using the rest of the initial resources. The new sequent judgment of the resolution system with the I/O model takes the form

$$\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow G \ ,$$

where Δ_I is the multiset of linear resources that is given as input in order to prove the goal G. Part of the input context Δ_I is used to prove G and the rest of resources that are not used is returned as the output context Δ_O. In [76], they call this deductive system as resource management system instead of I/O model, and represent as RM_1.

To illustrate the I/O model, we present the incorporation of the input and output contexts into the $\otimes$ rule,

$$\frac{\Gamma; \Delta_I \setminus \Delta_M \Longrightarrow G_1 \qquad \Gamma; \Delta_M \setminus \Delta_O \Longrightarrow G_2}{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow G_1 \otimes G_2} \otimes R \ ,$$

where Δ_I has the initial resources to prove the goal $G_1 \otimes G_2$. According to the *I/O model*, all resources are available to prove the subgoal G_1. After proving G_1 using the resources $(\Delta_I - \Delta_M)$, the remaining resources in Δ_M are used to prove G_2. The output context Δ_O includes unused resources. This mechanism deterministically identifies the proper splitting of the input context Δ_I into the two subproofs.

3.1.2 Backchaining and Residuation in LHHF

In the resolution system for LHHF proposed in [76], the authors first apply right rules to decompose and reduce non-atomic goal formulas. When the goal formula is finally atomic (a), a program formula (D) is selected from either the intuitionistic context or from the linear context. Resolution rules for selecting from the linear context and from the intuitionistic context are given in Figure 3.1.

$$\frac{D \gg a \setminus G \quad \Gamma, D; \Delta_I \setminus \Delta_O \Longrightarrow G}{\Gamma, D; \Delta_I \setminus \Delta_O \Longrightarrow a}\ \textbf{res-atm\_int}$$

$$\frac{D \gg a \setminus G \quad \Gamma; \Delta_I \setminus \Delta_O \Longrightarrow G}{\Gamma; \Delta_I, D \setminus \Delta_O \Longrightarrow a}\ \textbf{res-atm\_lin}$$

Figure 3.1: Resolution rules for the linear context and the intuitionistic context.

In both cases, the program formula and the atomic goal are passed to a *residuation* judgment in order to produce a residual subgoal G to be subsequently proven. We should note that, if the program formula D is selected from the linear context, then D is removed from the linear context to prevent reusing it. The *residuation* judgment takes the form

$$D \gg a \setminus G\ ,$$

which stands to mean that the atomic goal a can be proven from the program D, assuming that a proof for the subgoal G can later be generated.

For example, the residuation rule for atomic resources is given as

$$\frac{}{a' \gg a \setminus a' \doteq a}\ \textbf{dec-atm}\ ,$$

where a and a' are atomic formulas and we can have a solution if a and a' are equal.

The residuation rule for linear implication is given as

$$\frac{D \gg a \setminus G'}{G \multimap D \gg a \setminus G' \otimes G}\ \textbf{dec-}\multimap\ ,$$

where if we have the program formula $G \multimap D$, then we prove the atomic goal a from the program formula D, while we add new subgoals G' and G to the goal context to prove all goals later.

3.2 The LPL Grammar and Sequent Definitions

In this section, we first introduce the LPL grammar, which is a similar version of the LHHF grammar. Afterwards, we introduce sequent definitions for LPL. We call this new resolution system **BL**.

3.2.1 The LPL Grammar

LPL is an extension of LHHF, with the addition of negative occurrences of simultaneous conjunction ($\otimes$). On the other hand, we exclude alternative conjunction ($\&$), Top ($\top$) and $\mathbf{1}$ from the LPL language, since we can still represent STRIPS style planning problems without these connectives. We implement these changes by first defining LPL through the grammar

$$\begin{array}{lrcl} \text{Program formulas:} & D & ::= & a \mid D_1 \otimes D_2 \mid G \multimap D \mid \forall x.\, D \\ \text{Goal formulas:} & G & ::= & a \mid G_1 \otimes G_2 \mid D \multimap G \mid D \supset G \\ & & & \mid \forall x.\, G \mid a_1 \doteq a_2 \mid\, !G \mid \exists x.\, G \ \ , \end{array}$$

where a denotes atomic formulas. The most important difference between the LHHF grammar and the LPL grammar is that we incorporate simultaneous conjunction ($\otimes$) into program formulas. This addition lets us encode actions within task planning domain with formulas that can have multiple resources in their effects for planning problems. In this grammar, we also exclude program formulas of the form $G \supset D$, since we can model this type of form through $(!G) \multimap D$. However, we need formulas of the form $D \supset G$ in goal formulas since we do not allow negative occurrences of exponentials, $!D$.

3.2.2 Sequent Definitions of LPL

In order to present our backchaining resolution system for LPL, we now describe an associated sequent calculus formulation. Our sequent takes the form

$$\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} G \; , \tag{3.1}$$

which enhances the system described in [76] with a number of improvements. This sequent definition captures the statement that the goal G can be proven using unrestricted resources from the multiset Γ and consumable resources from the multiset Δ_I, leaving resources in the *output* multiset Δ_O unused. The *forbidden* set, F, above the sequent arrow keeps a set of assumption labels from Δ_I such that no subproof of this sequent is allowed to use them in any way.

All program formulas in Γ and Δ are assumed to be of the form $(u : {}^{L}D)$, where u is an unique label assigned for each resource D. The presence of D in Γ or Δ depends on a set of previously decomposed resources, with the superscript set L storing labels of these resources. We keep track of such dependencies to control boundaries beyond which partially decomposed resources in Δ_O should not travel.

We now define our second judgment, *residuation*, motivated by [76] with a number of important improvements as

$$u : {}^{L}D \gg a \setminus \Delta_G \triangleright \Delta_O \; , \tag{3.2}$$

where D is the program formula that we focus on (with dependency labels L), and a is an atomic goal. Δ_G is a multiset of subgoal formulas, filled in by negative occurrences of implication rules. Each resource in Δ_G is the form of $(u_g : G^F)$, where u_g is a unique subgoal label, G is the subgoal formula, and F is a list of resource labels that should not be used in the proof of this subgoal. Δ_O is a multiset of sub-formulas of D that are left unused, filled in through simultaneous conjunction within D.

Using the sequent definitions of (3.1) and (3.2), we can prove a goal formula

G in LPL if we can prove the sequent

$$\cdot;\cdot\setminus\cdot\xRightarrow{\emptyset}G\ .$$

These enhancements of the language make it possible to incorporate negative occurrences of simultaneous conjunction ($\otimes$), increasing the expressivity of LPL while preserving the complexity of the language compared to efficient languages such as LHHF. We should also note that unlike LHHF provers like Lolli, we are not particularly concerned with the *uniformity* of LPL proofs, since our goal is not the specification of a logic programming language. Instead of the *uniformity*, we are much more interested in the *efficiency* of proof search.

3.3 Proof Theory of LPL

In this section, we present a novel sequent calculus to perform proof search for **BL** , using the *I/O model* for resource management. This calculus, much like the resolution calculus for LHHF (RM_2), begins proof search by eagerly decomposing the goal formula in the given sequent, using appropriate right rules (also called *reduction* rules). We categorize right sequent rules to two groups, multiplicative right rules in Section 3.3.1 and right rules for exponentials and quantifiers in Section 3.3.2. Once the goal formula is an atomic goal a, proof search continues by decomposing a program formula from either the unrestricted or the linear context, given in Section 3.3.3.

3.3.1 Multiplicative Right Rules

We already mentioned the meaning of each linear logic connective in Section 2.2.2. While describing proof rules for **BL**, we mostly focus on differences of **BL** comparing to RM_2. We start with multiplicative right rules presented in Figure 3.2. If the goal formula is composed with a simultaneous conjunction ($\otimes$) or a linear implication ($\multimap$), we can apply multiplicative right rules.

$$\frac{\Gamma; \Delta_I \setminus \Delta_M \overset{F}{\Longrightarrow} G_1 \quad \Gamma; \Delta_M \setminus \Delta_O \overset{F}{\Longrightarrow} G_2}{\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} G_1 \otimes G_2}\ \textbf{bl-}\otimes$$

$$\frac{\Gamma; \Delta_I, (u : {}^{[u]}D) \setminus \Delta_O \overset{F}{\Longrightarrow} G \qquad \text{if } (v : {}^{L}D_k) \in \Delta_O \text{ and } v \notin \Delta_I \text{, then } u \notin L}{\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} D \multimap G}\ \textbf{bl-}\multimap$$

Figure 3.2: Multiplicative right sequent rules for the **BL** proof theory.

In the **bl-**$\otimes$ rule, we decompose the goal and then achieve the first goal using the input context Δ_I, storing the leftover resources in Δ_M. Subsequently, we achieve the second goal using the leftover resources in Δ_M.

Among multiplicative right rules for **BL**, the most interesting is perhaps the right rule for linear implication, **bl-**$\multimap$. Much like the standard right rule for this connective, we attempt to prove the subgoal G with the addition of new assumption D into the linear context. The primary difference is in the requirement that none of the output resources in Δ_O are allowed to depend on the newly introduced resource D. This is necessary since, as we will see shortly, Δ_O is no longer a strict subset of the input context. Consequently, not only it is possible that the newly introduced resource ends up in the output context, but other resources that are derived from it by delayed application of left rules may also be left unused. Allowing those resources to cross this linear implication boundary would result in an unsound proof theory. The set of dependence labels, L, for resources in the output context will be maintained by resolution rules as we describe in the following sections.

3.3.2 Right Rules for Exponentials and Quantifiers

In this section, we present four right rules for exponentials and quantifiers, given in Figure 3.3.

When proving $D \supset G$ using the **bl-**$\supset$ rule, we add D to unrestricted resources and then prove the goal G. We should note that, similar to the **bl-** $\multimap$ rule, none of the output resources in Δ_O are allowed to depend on the newly introduced resource D.

Using the **bl-!** rule, we can prove a given goal using only unrestricted resources from Γ. The **bl-**$\forall$ rule postpones instantiating the variable x, replacing x with a parameter c. The new parameter a is a fresh variable and can not exist before. In the **bl-**$\exists$ rule, a term t is supplied and substituted for the variable x.

$$\frac{\Gamma, (u : {}^{[u]}D); \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} G \qquad \text{if } (v : {}^{L}D_k) \in \Delta_O, \text{ then } u \notin L}{\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} D \supset G}\ \textbf{bl-}\supset$$

$$\frac{\Gamma; \cdot \setminus \Delta_O \overset{\emptyset}{\Longrightarrow} G}{\Gamma; \Delta_I \setminus \Delta_I \overset{F}{\Longrightarrow} !G}\ \textbf{bl-!}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} [c/x]G}{\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} \forall x.\ G}\ \textbf{bl-}\forall \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} [t/x]G}{\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} \exists x.\ G}\ \textbf{bl-}\exists$$

Figure 3.3: Exponential and quantifier right sequent rules for the **BL** proof theory.

3.3.3 Resolution in LPL

Once all the applicable right rules are exhausted, the right hand side of the sequent will contain an atomic goal a. Much like backchaining systems for LHHF, when the right hand side of a sequent has this form, our system selects a program formula from either the unrestricted or the linear context and decomposes it to determine whether it can be used to achieve the atomic goal, recording necessary subgoals through the residuation judgement. In our system, this *resolution* step is where all the structural nondeterminism is aggregated, both in selecting program formulas to be decomposed, as well as the nondeterminism in the residuation judgement that results from negative occurrences of simultaneous conjunction. Resolution rules for our system are given in Figure 3.4.

Resolution

$$\frac{u : {}^{L}D \gg a \setminus (g_1 : G_1^{F_1}, ..., g_n : G_n^{F_n}) \triangleright \Delta_{O,0} \quad ... \quad \Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \overset{F_k \cup F}{\Longrightarrow} G_k \quad ...}{\Gamma; \Delta_I, (u : {}^{L}D) \setminus \mathrm{AddLb}(L, \Delta_{O,n}, F) \overset{F}{\Longrightarrow} a}\ \textbf{bl-lin}$$

$$\frac{v : {}^{L\cup\{v\}}D \gg a \setminus (g_1 : G_1^{F_1}, ..., g_n : G_n^{F_n}) \triangleright \Delta_{O,0} \quad ... \quad \Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \overset{F_k \cup F}{\Longrightarrow} G_k \quad ...}{\Gamma, (u : {}^{L}D); \Delta_I \setminus \mathrm{AddLb}(L, \Delta_{O,n}, F) \overset{F}{\Longrightarrow} a}\ \textbf{bl-int}$$

Notes:

- $u \notin F$ and $k = 1, .., n$,
- $\Delta_{I,1} = \Delta_I, \Delta_{O,0}$.

Figure 3.4: Resolution rules for atomic goals within the LPL backchaining proof theory

Below, we use the linear resolution rule to describe a number of important issues. The linear resolution rule for our system takes the form

$$\frac{u : {}^{L}D \gg a \setminus (g_1 : G_1^{F_1}, ..., g_n : G_n^{F_n}) \triangleright \Delta_{O,0} \quad ... \quad \Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \overset{F_k \cup F}{\Longrightarrow} G_k \quad ...}{\Gamma; \Delta_I, (u : {}^{L}D) \setminus \mathrm{AddLb}(L, \Delta_{O,n}, F) \overset{F}{\Longrightarrow} a}\ \textbf{bl-lin}\,.$$

A number of side conditions accompany this rule:

- $u \notin F$, ensuring that forbidden resources are never focused on and hence consumed,
- $k = 1, ..., n$,
- $\Delta_{I,1} = \Delta_I, \Delta_{O,0}$ providing the input context to the first subgoal,
- $\Delta_{I,k+1} = \Delta_{O,k}$, transferring output resources from one subgoal to the next,
- $\mathrm{AddLb}(L, \Delta_{O,n}, F)$ in the final output context makes sure that dependence lists of all forbidden input resources (i.e. resources whose labels are in F) are augmented with the dependence list of the resource being focused on.

Note that if a resource has its label in F, that means that somehow, its presence depends on the success of this proof. Consequently, context elements used in this proof and their dependencies must in turn be added as new dependencies to these forbidden resources to ensure sound reasoning. The necessity of the AddLb() function in the resolution rule is illustrated by the following goal formula

$$G = (c \multimap a \otimes b) \multimap (c \multimap b) \otimes a$$

which should not be provable in a sound system. However, if we let the leftover resource a from the first resource spill over beyond the place where the resource to prove the associated subgoal c (from $c \multimap b$), then it might end up being used to prove the overall goal a. The use of the AddLb() function prevents this from happening. More formally, the AddLb() function is defined as

$$\text{AddLb}(L, \Delta, F) := \left\{ (u_i : {}^{\bar{L}_i} D_i) \mid (u_i : {}^{L_i} D_i) \in \Delta, \text{ and } \bar{L}_i = \left\{ \begin{array}{ll} L_i \cup L & \text{if } u_i \in F \\ L_i & \text{otherwise} \end{array} \right\} \right. ,$$

The resolution rule for the unrestricted context (**bl-int**) is very similar to the linear resolution rule (**bl-lin**) and shown in Figure 3.4.

3.3.4 Backchaining and Residuation in LPL

Both of these resolution rules appeal to the *residuation* judgement, presented in Figure 3.5. Much like efficient backchaining proof theories for LHHF, this judgement extracts subgoals necessary to achieve a given atomic goal using a particular program formula. The key difference in our system, however, is in the way simultaneous conjunction on the left is handled (which is altogether missing from LHHF) since in proving an atomic conclusion, one of the conjuncts must be left unused. Consequently, the associated residuation rule in Figure 3.5 records the unused resource on the *output* section of the sequent, leaving it to be used by later parts of the proof search. The challenge, and the primary contribution of our work is in making sure that proper boundaries are established for where and when these subformula outputs can be used within the proof without compromising soundness.

Residuation

$$\frac{\text{unify}(a', a)}{u : {}^{L}(a') \gg a \setminus \cdot \triangleright \cdot}\ \mathbf{d}_{lpl}\textbf{-atm}$$

$$\frac{v : {}^{L\cup\{v\}}D \gg a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(G_1 \multimap D) \gg a \setminus (j : G_1^{Lb(\Delta_O)}), \Delta_G \triangleright \Delta_O}\ \mathbf{d}_{lpl}\text{-}\multimap$$

$$\frac{v_1 : {}^{L\cup\{v_1\}}D_1 \gg a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(D_1 \otimes D_2) \gg a \setminus \Delta_G \triangleright \Delta_O, (v_2 : {}^{L\cup\{v_2\}}D_2)}\ \mathbf{d}_{lpl}\text{-}\otimes_{1L}$$

$$\frac{v_1 : {}^{L\cup\{v1\}}D_2 \gg a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(D_1 \otimes D_2) \gg a \setminus \Delta_G \triangleright \Delta_O, (v_2 : {}^{L\cup\{v_2\}}D_1)}\ \mathbf{d}_{lpl}\text{-}\otimes_{1R}$$

$$\frac{v : {}^{L\cup\{v\}}([t/x]D) \gg a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(\forall x.\ D) \gg a \setminus \Delta_G \triangleright \Delta_O}\ \mathbf{d}_{lpl}\text{-}\forall$$

Figure 3.5: Residuation rules for atomic goals within the LPL backchaining proof theory

The $\mathbf{d}_{lpl}$**-atm** rule unifies the atomic resource (a') with the atomic goal (a). The $\mathbf{d}_{lpl}\text{-}\multimap$ rule tries to prove the atomic goal a using the resource D, and also adds the goal G_1 to the goal context to prove them later. We should note that, all labels coming from the output context are added to the forbidden list of the goal G_1, since we can not use these resources to prove this goal. There are two rules for simultaneous conjunction, $\mathbf{d}_{lpl}\text{-}\otimes_{1L}$ and $\mathbf{d}_{lpl}\text{-}\otimes_{1R}$. The former one is using the first resource D_1 to prove the atomic goal a and putting the second resource D_2 into the output context. The latter rule is doing reverse of the first rule, using the second resource to prove the goal while putting the first resource to the output context. In the last rule $\mathbf{d}_{lpl}\text{-}\forall$, a term t is supplied and substituted for the variable x in the given resource D to prove the atomic goal.

The need of forbidden set of assumptions is exemplified by the sequent

$$\cdot; a \multimap (a \otimes b) \setminus \cdot \xRightarrow{F} b.$$

This sequent should not be provable. The proof of the subgoal a cannot use the leftover resource from the succedent of the implication $(a \otimes b)$. However, if the forbidden set of assumptions F was not used, this sequent could be proven in a wrong way. An important note about this judgment is that $\Delta_O \subseteq \Delta_I$ property does not hold anymore, since the output context Δ_O may contain sub-formulas of certain resources in Δ_I. This is a direct result of allowing negative occurrences of simultaneous conjunction.

3.4 Incorporating Alternative Conjunction

In this section, we briefly describe how to inject the alternative conjunction connective into the current proof system. If we would have the alternative conjunction connective (&) in our proof system, we could describe the related rules as in Figure 3.6.

$$\frac{\Gamma;\Delta_I \setminus \Delta^0_{O1},\Delta^1_{O1} \overset{F}{\Longrightarrow} G_1 \qquad \Gamma;\Delta_I \setminus \Delta^0_{O2},\Delta^1_{O2} \overset{F}{\Longrightarrow} G_2 \qquad \begin{array}{l}\Delta^0_{O1} \subseteq (\Delta^0_{O2},\Delta^1_{O2}) \text{ and} \\ \Delta^0_{O2} \subseteq (\Delta^0_{O1},\Delta^1_{O1})\end{array}}{\Gamma;\Delta_I \setminus \mathrm{merge}(\Delta^0_{O1},\Delta^1_{O1},\Delta^0_{O2},\Delta^1_{O2}) \overset{F}{\Longrightarrow} G_1 \& G_2}\ \textbf{bl-}\&$$

$$\frac{v : {}^{L\cup\{v\}}D_1 \gg a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(D_1 \& D_2) \gg a \setminus \Delta_G \triangleright \Delta_O}\ \textbf{d-}\&_1$$

$$\frac{v : {}^{L\cup\{v\}}D_2 \gg a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(D_1 \& D_2) \gg a \setminus \Delta_G \triangleright \Delta_O}\ \textbf{d-}\&_2$$

Figure 3.6: The alternative conjunction (&) rules for the **BL** proof theory.

The **bl-**& rule needs care in matching up leftover resources on the outputs of its two premises. Subproofs leading to the generation of partial leftovers must be identical in order to ensure that they can be merged later during proof transformation. We can describe the merge function used in the **bl-**& rule as

$$\mathrm{merge}(\Delta^0_{O1},\Delta^1_{O1},\Delta^0_{O2},\Delta^1_{O2}) := (\Delta^0_{O1} \cup \Delta^0_{O2}) \cup (\Delta^1_{O1} \cap \Delta^1_{O2}),$$

where the union and intersection operations are normal set operations since resource labels are unique. This definition basically retrieves resources common to both output contexts, making the slack indicator zero if it is zero in one of the premises.

3.4.1 Eliminating Nondeterminism Caused by Additive Conjunction (&)

Even though the I/O model handles an important type of nondeterminism, we still need to eliminate a number of other sources of non-determinism in the proof system. One issue is caused by the *additive conjunction* (&). Hodas and Miller introduced the right rule for this connective in [19], taking the form

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow G_1 \qquad \Gamma; \Delta_I \setminus \Delta_O \Longrightarrow G_2}{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow G_1 \& G_2} \;\&R \;,$$

where we first prove the goal G_1, returning an output context Δ_O. And then we prove G_2, returning another output context which has to be exactly the same as Δ_O. We need to prove both premises and then compare the returned outputs. If the outputs are not same, we need to backtrack and then search other possibilities. In [76], they propose a better solution, which eliminates the need to backtrack.

The new rule takes the form

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow G_1 \qquad \Gamma; \Delta_I - \Delta_O \setminus \cdot \Longrightarrow G_2}{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow G_1 \& G_2} \;\&R \;.$$

In this rule, instead of checking the output contexts of two premises, the second premise uses the exact resources it needed, coming from the difference of the input context and the output context for the first premise computed while proving the subgoal G_1.

3.5 A Task Planning Example With The RHex Hexapod

In this section, we introduce a task planning example domain to present concepts and algorithms for Linear Planning Logic. In this domain, a mobile robot, the RHex hexapod [20] navigates in an environment populated with uniquely identifiable colored landmarks. Each path between a pair of landmarks has a different surface property, such as being mud, rough or smooth, allowing no movement, walking or running respectively. The robot also has the capability of *tagging* landmarks, corresponding to a particular action carried out at that location. Each landmark has a state of *tagged* or *untagged*. In our introduced example domain, we have four actions, the Walk(X) action is used for walking on rough ground towards the landmark X, the Tag(X) action is used for tagging the landmark X, the Run(X) action is used for running on firmer ground towards the landmark X and the Seek(X) action is used for visually searching for the landmark X. We summarize predicate and action definitions to represent the robotic planning domain in Figure 3.7.

Predicates:		
at(X)	:	Robot is at the landmark X.
tagged(X)	:	Landmark X is tagged by the robot.
untagged(X)	:	Landmark X is not tagged by the robot.
see(X)	:	Robot can see the landmark X.
surface(X, Y, Z)	:	Path between X and Y has type Z.
Actions:		
Walk(X)	:	Robot walks to the landmark X.
Tag(X)	:	Robot tags the landmark X.
Run(X)	:	Robot runs to the landmark X.
Seek(X)	:	Robot searches for the landmark X.

Figure 3.7: Predicates and actions for the robotic planning domain.

The next step is to associate these actions with LPL expressions to formally represent their preconditions and effects. In Figure 3.8, we present encodings for all actions, defined parametrically on the landmark identifier X. If the robot sees an arbitrary landmark Y, using the Seek(X) action the robot searches for a specific landmark X by continually turning in place while visually scanning the environment. Using the Tag(X) action, the robot can tag an untagged landmark X while remaining in the same position. If the robot can see the landmark X and the path surface is rough, then using the Walk(X) action the robot can walk from the landmark Y to the landmark X. If the robot can see the landmark X and the path surface is smooth, then using the Run(X) action the robot can run from the landmark Y to the landmark X.

$$\boxed{\begin{array}{lcl}
\mathrm{Seek}(X) & : & \mathrm{see}(Y) \multimap \mathrm{see}(X) \\
\mathrm{Tag}(X) & : & \mathrm{at}(X) \otimes \mathrm{untagged}(X) \multimap \mathrm{at}(X) \otimes \mathrm{tagged}(X) \\
\mathrm{Walk}(X) & : & \mathrm{at}(Y) \otimes \mathrm{see}(X) \otimes \mathrm{surface}(Y, X, \mathrm{rough}) \multimap \\
 & & \mathrm{at}(X) \otimes \mathrm{see}(X) \otimes \mathrm{surface}(Y, X, \mathrm{rough})) \\
\mathrm{Run}(X) & : & \mathrm{at}(Y) \otimes \mathrm{see}(X) \otimes \mathrm{surface}(Y, X, \mathrm{smooth}) \multimap \\
 & & \mathrm{at}(X) \otimes \mathrm{see}(X) \otimes \mathrm{surface}(Y, X, \mathrm{smooth})
\end{array}}$$

Figure 3.8: Encodings of walking and tagging actions within LPL.

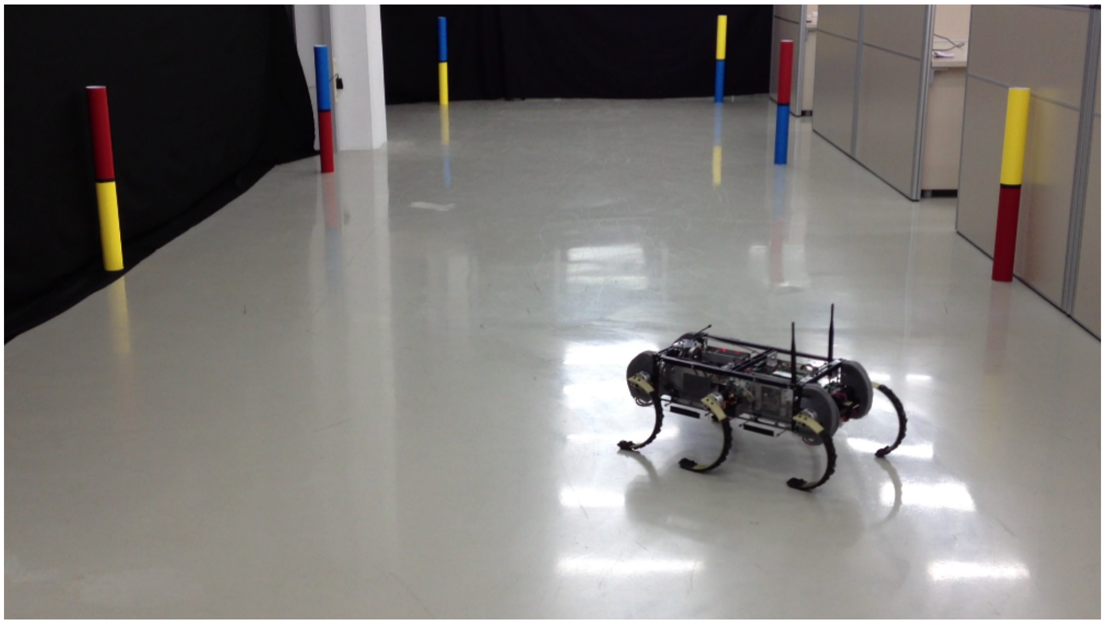

Figure 3.9: A snapshot from the experimental setup in its initial state. Six colored landmarks are scattered throughout the environment, observable through a camera mounted on the RHex robot.

An example planning problem in this domain would be navigating through landmarks and tagging a desired set of landmarks while leaving others untagged, using the given set of actions in Figure 3.8. We now set up a real planning scenario shown in Figure 3.9, with six uniquely colored landmarks and a RHex robot. Initially, the robot is at the Start location and the landmark b_0 is visible. The initial state of the example such as tagged information of landmarks and properties of path surfaces is encoded as

$$\begin{aligned} D_i = \ & \text{at}(\text{Start}) \otimes \text{untagged}(b_3) \otimes \text{untagged}(b_5) \\ & \otimes \text{surface}(\text{Start}, b_1, \text{rough}) \otimes \text{surface}(b_1, b_0, \text{rough}) \\ & \otimes \text{surface}(b_0, b_3, \text{rough}) \otimes \text{surface}(b_3, b_4, \text{smooth}) \\ & \otimes \text{surface}(b_4, b_2, \text{smooth}) \otimes \text{surface}(b_2, b_5, \text{smooth}) \\ & \otimes \text{see}(b_0) \ . \end{aligned}$$

The final state of the example is encoded as

$$G = \text{at}(b_5) \otimes \text{tagged}(b_3) \otimes \text{tagged}(b_5) \otimes \top \ ,$$

where the robot is expected to be at the landmark b_5 while landmarks b_3 and b_5 are tagged. The *Top* symbol $\top$ at the end of the goal formula is used to indicate that whichever resources are leftover at the end of the reasoning are not to be considered and can be consumed.

In our example domain, we use cylindrical landmarks with a height of $1.5m$ and a radius of $8cm$ each. Each landmark has two distinct colors among red, yellow and blue. Relative ordering of a landmark (top vs. bottom) distinguishes it from other landmarks. An example of such a landmark is shown on the left side of Figure 3.10. The RHex robot navigates through landmarks using the movement actions and searches the target landmark using the Seek(X) action. To be able to use these actions, the robot has to find bearing and distance information for each landmark.

The RHex robot has two on-board RTD CME137LX single-board computer units. One of the computer unit is used for visual processing of color frames captured by a Point Grey Flea2 camera at 5Hz. In our image processing algorithm, we first apply a color filtering algorithm based on Mahalanobis distance (a color model) in RGB space (calibrated through data collected offline) to convert the original colored image into a labeled bitmap. Subsequently, we use the the

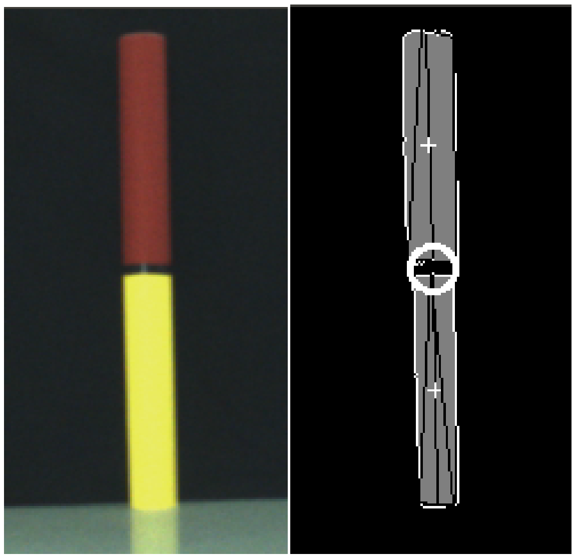

Figure 3.10: Left: A red-yellow landmark as seen by the robot. Right: blobs extracted from the image with their centers marked with a "+" sign. Computed landmark location is the midpoint of the two blobs and is marked with a white circle.

cvBlob library [80] to extract connected blobs in the bitmap according to domain specific conditions such as blob size, aspect ratio and orientation. Pairs of blobs are then associated with each other based on their relative locations constrained by the structure of our landmark design, yielding the final landmark locations as shown in the right side of Figure 3.10. When we determine the pair, we can also determine the landmark location as the midpoint of two blobs. Using this information, we can determine the distance to the landmark through the total blob area.

We implemented the LPL theorem prover described in Section 3.3 using SWI-Prolog. When we give an initial state, a goal state and all action encodings in in LPL, the implemented prover will find a plan if it exists. When we apply LPL planner to the initial state D_i and the goal state G together with the encoded action in Figure 3.8, the LPL planner extracts the plan (sequence of actions) as

$$\begin{aligned}&[\mathrm{Seek}(b_1), \mathrm{Walk}(b_1), \mathrm{Seek}(b_0), \mathrm{Walk}(b_0), \mathrm{Seek}(b_3),\\ &\mathrm{Walk}(b_3), \mathrm{Tag}(b_3), \mathrm{Seek}(b_4), \mathrm{Run}(b_4), \mathrm{Seek}(b_2),\\ &\mathrm{Run}(b_2), \mathrm{Seek}(b_5), \mathrm{Run}(b_5), \mathrm{Tag}(b_5)]\ .\end{aligned}$$

Visual landmark distance and bearing information is communicated back to an external workstation running a Matlab program. The LPL planner provides this program with a sequence of actions to be executed with the same labelings of landmarks as those identified by the visual tracker. The final plan execution system brings these components together, tracking the robot's progress and initiating the behaviors prescribed by the plan by communicating with the control board on RHex. We show the generated sequence of actions (14 total, computed in 90 seconds on a Intel Pentium Dual-Core CPU E6500 Processor 2.93GHz PC with 2 GB of RAM with an unoptimized SWI-Prolog implementation) for the resulting plan in Figure 3.11.

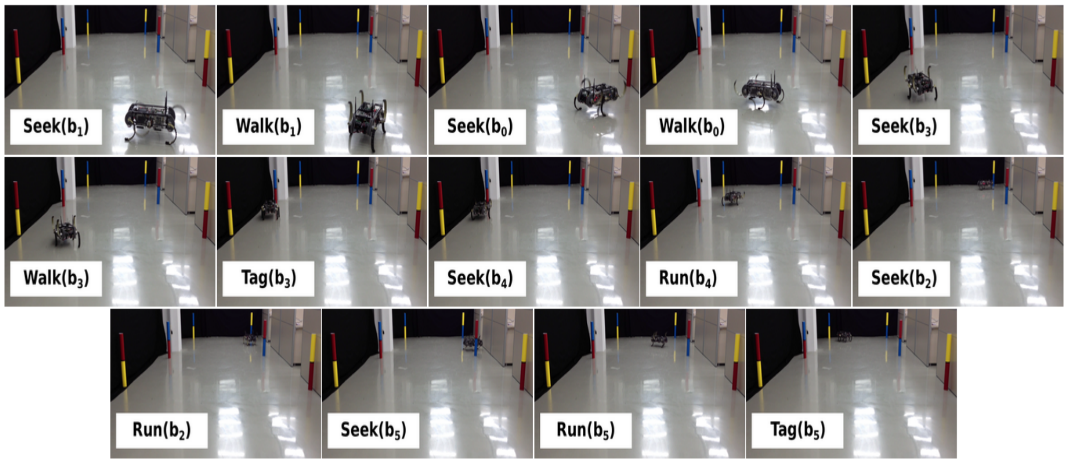

Figure 3.11: Snapshots during the execution of the behavioral plan for RHex generated by the LPL theorem prover.

3.6 Action Types in LPL

In this section, we briefly present a number of action types that can be represented in LPL. We can connect each unrestricted action to a goal with the unrestricted implication connective ($\supset$), while we connect each resource to a goal with the linear implication connective ($\multimap$). On the other hand, we usually represent planning problems in the LPL implementation as

$$Action_1 \supset Action_2 \supset \ldots \supset Action_n \Rightarrow state_1 \multimap state_2 \ldots state_m \multimap G,$$

where

$$\begin{array}{ll} \textit{Action} & : \text{Action formula,} \\ \textit{state} & : \text{Resource formula,} \\ G & : \text{Goal formula to be proven.} \end{array}$$

If an action is allowed to be used many times, it is placed into the context of unrestricted resources (Γ). Otherwise, actions that are allowed to be use exactly one time can be placed into the context of ephemeral resources (Δ). Below, we present a number of action types that can be represented in LPL, illustrating action types on example formulas.

- The basic action type
  $\textit{Action} : a \multimap b$

  If we have a, then we consume a and we can use b.

- The main action may create other actions
  $\textit{Action} : a \multimap (b \multimap (c \multimap d))$ can be split into sub-actions as
  $\textit{Sub-Action1} : b \multimap (c \multimap d)$
  $\textit{Sub-Action2} : c \multimap d$

  If we have a resource a, then we can activate *Action*. If we also have a resource b, then we can activate *Sub-Action1*. Finally, using the resource c, we can activate *Sub-Action2*.

- Pre-actions may be needed to activate for the main action
  $\textit{Action} : ((a \multimap b) \multimap c) \multimap d$ can be split into the sub-action as
  $\textit{Sub-Action1} : (a \multimap b)$

  To be able to activate *Action*, we should first activate *Sub-Action1*.

3.7 Expressivity and Efficiency of LPL

In this section, we discuss the expressivity power and efficiency of LPL, comparing LPL with other efficient theorem provers such as Dialoguell and Lolli. LPL grammar is an extension of the LHHF grammar by incorporating left conjunction. Although we extend the grammar in LPL and thereby increase the expressivity of LPL, we preserve efficiency of proof search as in Lolli.

We already mentioned that Lolli language is based on the LHHF grammar. According to the grammar used in Lolli, an action can not create multiple effects in a planning problem. This limitation comes from the absence of the left conjunction connective in the grammar. However, some implementation tricks can exceed this limitation and allow us to use actions with multiple effects as in the Dialoguell [54] planning language. In Dialoguell, subgoals are first achieved like in the forward chaining method. After all actions are applied and subgoals are achieved, then initially given goals are achieved. This approach exponentially increases complexity. We show a complexity issue in Dialoguell with an example. Assuming that an action's effects are same as preconditions of that action, then Dialoguell may apply this action infinitely many times, since the action's preconditions are created with the action's effects. To prevent loops for this type of examples, Dialoguell applies sequenced actions until reaching a certain limit, thereby searching a plan until to a predetermined depth.

We now compare expressivity of LPL with Lolli. We first present an action type that both LPL and Lolli can encode such as the example below

$$(R_1 \multimap (G_1 \otimes G_2)) \multimap R_2,$$

where G_1 and G_2 are subgoals that we need to prove and R_1 and R_2 are resources. Afterwards, we continue with and example action type where LPL can encode but Lolli cannot encode. Below, we give an example action for this type

$$((G_1 \multimap (R_1 \otimes R_2)) \multimap G_2) \multimap R_3,$$

where we first decompose the $(R_1 \otimes R_2)$ resource and then we place both R_1 and R_2 resources into the consumable resource context (Δ). However, in Lolli it is not allowed to have the $(R_1 \otimes R_2)$ type resources.

Another additional expressivity of LPL is being able to encode an action that makes another action available. We continue with an example to show this type of expressivity. In a card game, assume that we can use a *joker* card for any missing card in the hand. Any players win the round when all cards follow a sequence. We encode the dealing a joker to a user action in LPL as:

$$DealJoker : (myTurn \otimes notGetACard) \multimap joker \ , \tag{3.3}$$

then we can encode another action for finishing the round as:

$$FinishTheRound : (joker \otimes missingOneCard) \multimap finish \ , \tag{3.4}$$

where if we have a joker and also if we are missing only one card to finish, then we can finish the round. Since we will use the *FinishTheRound* action once, we can put this action into the effects of the *DealJoker* action. Then whenever a player missing a card and also have a joker can finish the game. The composed *DealJoker* action is as follow:

$$DealJoker : (myTurn \otimes notGetACard) \multimap (missingOneCard \multimap finish) \ , \tag{3.5}$$

where dealing a joker to the user triggers another action for finishing the game whenever the user is missing only one card. These two different representations for the *DealJoker* action, in (3.3) and (3.5), are doing the same work. However, the former one increases the number of actions which may also increase the complexity of plan search.

One may think to encode this action as below

$$DealJoker : (myTurn \otimes notGetACard \otimes missingOneCard) \multimap finish \ , \tag{3.6}$$

where *missingOneCard* resource is carried to the precondition part of the action. Then the meaning of dealing a joker action will change. We can't finish the game immediately when we get a joker. We need to wait until the next round and then check our hand if we are missing only one card.

Next, we give another example about a mail delivery robot to show expressivity of LPL. In this example domain, a mail delivery robot in a company automatically takes mails and then delivers them to corresponding rooms. Basically, the robot has 3 actions, *Move*, *GetMail* and *DeliverMail*. We describe objects, states and

Predicates:		
at(X)	:	Robot is at the location X
receivedMail(X)	:	A mail is received belonging to the room X
hasMail(X)	:	Robot has the mail for the room X
deliveredMail(X)	:	A mail belonging to the room X is delivered
Objects:		
mailRoom	:	The mail room where mails are collected
Actions:		
Move(X, Y)	:	Robot moves from the room X to the room Y.
GetMail(X)	:	At the mail room, robot gets the mail belonging to the room X.
DeliverMail(X)	:	At the room X, robot delivers the mail belonging to the room X.

Figure 3.12: Objects, states and actions of the mail delivery example.

Move(X, Y)	:	$at(X) \multimap at(Y)$.
GetMail(X)	:	$at(mailRoom) \otimes receivedMail(X) \multimap at(mailRoom) \otimes hasMail(X)$.
DeliverMail(X)	:	$at(X) \otimes hasMail(X) \multimap at(X) \otimes deliveredMail(X)$.

Figure 3.13: Encodings of the mail delivery actions in LPL.

actions of the mail delivery example in Figure 3.12, while we present encodings of the given actions in Figure 3.13.

In LPL, we can compose the *GetMail* action and the *DeliverMail* action in one action as represented in (3.7), where the *GetMail* action triggers the *DeliverMail* action that can be applied later. However, it is not possible to compose these actions in Lolli with the same way.

$$\begin{aligned} GetMail(X) \quad : \quad & at(mailRoom) \otimes receivedMail(X) \multimap \\ & ((at(X) \multimap (at(X) \otimes deliveredMail(X))) \otimes at(mailRoom)). \end{aligned} \tag{3.7}$$

If we try to compose the *GetMail* action and the *DeliverMail* action in another way as represented in (3.8), we will have a contradiction where the robot is at the mail room and at the room X at the same time.

$$\begin{aligned} GetMail(X) \quad : \quad & (at(mailRoom) \otimes receivedMail(X) \otimes at(X)) \multimap \\ & ((at(mailRoom) \otimes at(X) \otimes deliveredMail(X)). \end{aligned} \tag{3.8}$$

Using the alternative conjunction (&) connective in actions of planning examples, we can encode internal choices. We now introduce an example to show the expressivity of (&). Assume that we want to make a holiday plan, visiting the Eiffel Tower in Paris or the Statue of Liberty in New York. However, we want to decide the exact destination later. Thereby, we should have two plans for both destinations. We can encode the goal for this example as

$$(at(Paris)\ \&\ at(NewYork)) \multimap (\text{visit(Eiffel Tower)}\ \&\ visit(StatueofLiberty)),$$

where the *at* predicate represents the location and the *visit* predicate represents the visiting place. Since we can not be at the both cities, we need to use the & operator. We also need two actions to complete this example such as:

$$\begin{aligned} \text{Visit(Eiffel Tower)} &\ :\ \text{at(Paris)} \multimap \text{visit(Eiffel Tower)} \\ \text{Visit(Statue of Liberty)} &\ :\ \text{at(New York)} \multimap \text{visit(Statue of Liberty)}. \end{aligned}$$

We will now illustrate an important source of nondeterminism in proof search that is directly addressed by LPL. Recalling the planning domain in Section 3.5, assume that we have the following sequent at an intermediate stage within our example domain:

$$\text{at}(b_2) \otimes \text{tagged}(b_2) \Rightarrow \text{tagged}(b_2) \otimes \text{at}(b_2)\ .$$

An automated prover will consider all applicable alternatives in proof search, creating substantial nondeterminism in the search. We might first apply the $\otimes R$ rule

$$\boxed{\dfrac{\dfrac{\dfrac{?}{\text{at}(b_2), \text{tagged}(b_2) \Rightarrow \text{tagged}(b_2)}\ ?}{\text{at}(b_2) \otimes \text{tagged}(b_2) \Rightarrow \text{tagged}(b_2)}\ \otimes L \qquad \dfrac{?}{\cdot \Rightarrow \text{at}(b_2)}\ ?}{\text{at}(b_2) \otimes \text{tagged}(b_2) \Rightarrow \text{tagged}(b_2) \otimes \text{at}(b_2)}\ \otimes R}$$

Figure 3.14: Incorrect proof attempt using first the $\otimes R$ rule, then the $\otimes L$ rule.

to decompose the goal into its subgoals. Logic programming languages generally follow the strategy of first decomposing goals and then following backchaining.

We show the proof tree for this attempt in Figure 3.14. Unfortunately, this attempt does not lead to a valid proof. The valid proof for this case is given in Figure 3.15. The availability of these two alternatives, combined with the combinatorial complexity of splitting resources is a significant source of nondeterminism in proof search.

$$\dfrac{\dfrac{\dfrac{}{\mathrm{at}(b_2) \Rightarrow \mathrm{at}(b_2)}\,\mathit{init} \qquad \dfrac{}{\mathrm{tagged}(b_2) \Rightarrow \mathrm{tagged}(b_2)}\,\mathit{init}}{\mathrm{at}(b_2), \mathrm{tagged}(b_2) \Rightarrow \mathrm{at}(b_2) \otimes \mathrm{tagged}(b_2)}\,\otimes R}{\mathrm{at}(b_2) \otimes \mathrm{tagged}(b_2) \Rightarrow \mathrm{at}(b_2) \otimes \mathrm{tagged}(b_2)}\,\otimes L$$

Figure 3.15: Correct proof using the $\otimes L$ rule first, then the $\otimes R$ rule.

Chapter 4

Soundness Proof of Backchaining System for LPL

In this chapter, we present the soundness proof of LPL. To this end, we start with defining a reference logic system (RL), giving annotated proof terms and inference rules for this reference linear logic which is already proved to be sound. And then we continue by giving annotated proof terms and inference rules with proof terms for **BL** which is our core language. Afterwards, we introduce a proof rewriting structure, including an intermediate linear logic combining inference rules of RL and **BL**, which will be used in proof rewriting rules. Finally, we finish the proof with proof rewriting rules, which converts LPL proofs to linear logic proofs.

4.1 A Reference Logic System (RL)

We prove the soundness of the System **BL** with respect to a sequent calculus system for intuitionistic linear logic with resource management, limited to the LPL fragment, which we call this reference system the System RL. We start with giving proof annotations and then the sequent calculus proof system for RL, respectively.

4.1.1 Proof Annotations for RL

Clearly, the utility of the **BL** language is tied to the practicality of constructing an efficient theorem prover that can automatically find a proof for a given formula. Regardless of the details of this proof procedure, however, another necessity for our domain is the ability to extract a solution to the original planning problem, i.e. a plan, out of such a proof. In this section, we describe an annotation language for RL proof terms based on a simple Natural Deduction system that is definitionally useful but inappropriate for proof search. This language closely follows the linear type checking and lambda calculus presentation of [81] and also allows effective proof checking through careful resource management.

The proof term language for the reference system is defined through the grammar

Introduction Terms:	M ::=	$\hat{\lambda} w.M$	linear function definition
		$\mid M \otimes N$	mult. pair constructor
		$\mid$ let $w_1 \otimes w_2 = E$ in M	mult. pair deconstructor
		$\mid \lambda w.M$	function definition
		$\mid\, !M$	unrestricted constructor
		$\mid \lambda^{\forall} x.M$	universal constructor
		$\mid M \diamond^{\exists} t$	existential constructor
		$\mid$ let $u = E$ in M	substitution let form
		$\mid e(E)$	coercion
Elimination Terms:	E ::=	u	variables
		$\mid E\,\hat{}\,M$	linear function application
		$\mid M \diamond^{\forall} t$	universal deconstructor
		$\mid M : G$	coercion

Note that these proof terms are separated into introduction and elimination forms with explicit coercion terms. This allows the proof checker to alternate between type checking and type synthesis, eliminating the need for redundant type annotations within the proof term. Except the coercion from elimination terms to introduction terms, this language captures normal proof terms.

4.1.2 Sequent Calculus Proof System for RL

Hypotheses: RL

$$\frac{}{\Gamma; (\Delta_I, u : a) \setminus \Delta_I \Longrightarrow e(u) : a}\ \textbf{rl-init}$$

$$\frac{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \Longrightarrow M : G \qquad (v : D) \notin \Delta_O}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \Longrightarrow (\text{let } v = u \text{ in } M) : G}\ \textbf{rl-copy}$$

(Note: The newly introduced resource is not allowed in the output since the application of this rule would be unnecessary if the new resource was not consumed)

Multiplicative Connectives

$$\frac{\Gamma; (\Delta_I, \Delta, u : D) \setminus \Delta_I \Longrightarrow M : G}{\Gamma; \Delta_I, \Delta \setminus \Delta_I \Longrightarrow (\hat{\lambda} u.M) : D \multimap G}\ \textbf{rl-}{\multimap}\text{R}$$

$$\frac{\Gamma; \Delta_I, \Delta_1, \Delta_2 \setminus \Delta_I, \Delta_2 \Longrightarrow M : G \qquad \Gamma; (\Delta_I, \Delta_2, w : D) \setminus \Delta_I \Longrightarrow N : C}{\Gamma; (\Delta_I, \Delta_1, \Delta_2, u : G \multimap D) \setminus \Delta_I \Longrightarrow (\text{let } w = u\,\hat{}\,M \text{ in } N) : C}\ \textbf{rl-}{\multimap}\text{L}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta' \Longrightarrow M_1 : G_1 \quad \Gamma; \Delta' \setminus \Delta_O \Longrightarrow M_2 : G_2}{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow M_1 \otimes M_2 : G_1 \otimes G_2}\ \textbf{rl-}{\otimes}\text{R}$$

$$\frac{\Gamma; (\Delta_I, \Delta, w_1 : D_1, w_2 : D_2) \setminus \Delta_I \Longrightarrow M : G}{\Gamma; (\Delta_I, \Delta, u : D_1 \otimes D_2) \setminus \Delta_I \Longrightarrow (\text{let } w_1 \otimes w_2 = u \text{ in } M) : G}\ \textbf{rl-}{\otimes}\text{L}$$

(Note: New assumptions cannot appear in the output context.)

Quantifiers

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow M : [c/x]G}{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow \lambda^{\forall} x.([x/c]M) : \forall x.\, G}\ \textbf{rl-}{\forall}\text{R} \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow M : [t/x]G}{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow M \diamond^{\exists} t : \exists x.\, G}\ \textbf{rl-}{\exists}\text{R}$$

$$\frac{\Gamma; (\Delta_I, \Delta, w : [t/x]D) \setminus \Delta_I \Longrightarrow M : G}{\Gamma; (\Delta_I, \Delta, u : \forall x.\, D) \setminus \Delta_I \Longrightarrow (\text{let } w = u \diamond^{\forall} t \text{ in } M) : G}\ \textbf{rl-}{\forall}\text{L}$$

Exponentials

$$\frac{(\Gamma, u : D); \Delta_I \setminus \Delta_O \Longrightarrow M : G}{\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow \lambda u.M : D \supset G}\ \textbf{rl-}{\supset}\text{R} \qquad \frac{\Gamma; \cdot \setminus \cdot \Longrightarrow M : G}{\Gamma; \Delta_I \setminus \Delta_I \Longrightarrow\, !M :!G}\ \textbf{rl-}!\text{R}$$

Figure 4.1: System RL: A reference sequent calculus system for the LPL fragment of intuitionistic linear logic.

We prove the soundness of the System **BL** with respect to a sequent calculus system for intuitionistic linear logic with resource management, limited to the LPL fragment. This reference system, which we call the System RL, is based on the sequent definition

$$\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow M : G$$

which indicates that the goal formula G is provable with unrestricted assumptions in Γ and linear resources in Δ_I with resources in $\Delta_O \subseteq \Delta_I$ left unused. Finally, the proof term M records the proof construction as usual. Figures 4.1 illustrate associated sequent calculus rules.

Note that both the structure and proof terms in this reference system are chosen to be compatible with the System **BL**. In Section 4.3.1, we introduce an extended "intermediate" logic, that combines rules from the System **BL** with those in the reference System RL to create a system which is a superset of both, and which will be invariant under proof transformations through pairwise changes in rule ordering. Before we proceed with the description of this intermediate system, we note a few features of reference system.

- The side condition associated with the copy rule **rl-copy** ensures that the copied resource is always consumed to prevent unnecessary consideration of unrestricted resources.
- All left rules require that if a new resource has been introduced to the linear context, it must be consumed by the subgoal.

4.2 Backchaining System for LPL (BL)

We already mentioned proof theory and inference rules for LPL in Chapter 3. In this section, we introduce annotated proof terms and inference rules with proof terms for LPL.

4.2.1 Proof Annotations for BL

In this section, we introduce proof annotations for the System **BL**, similar to the proof terms for the reference system RL given in Section 4.1.1. We add two new introduction terms for multiple pair deconstructor, def_1 and def_2, instead of the "let $w_1 \otimes w_2 = E$ in M".

We present the proof term language for the **BL** system through the grammar

Introduction Terms:	M	$::=$	$\hat{\lambda} w.M$	linear function definition
			$\mid M \otimes N$	mult. pair constructor
			$\mid \text{def}_1\ w_1 {\otimes} w_2 = E$ in M	mult. pair deconstructor (1)
			$\mid \text{def}_2\ w_1 \otimes w_2 = E$ in M	mult. pair deconstructor (2)
			$\mid \lambda w.M$	function definition
			$\mid !M$	unrestricted constructor
			$\mid \lambda^{\forall} x.M$	universal constructor
			$\mid M \diamond^{\exists} t$	existential constructor
			$\mid$ let $u = E$ in M	substitution let form
			$\mid e(E)$	coercion
Elimination Terms:	E	$::=$	u	variables
			$\mid E\hat{\ }M$	linear function application
			$\mid M \diamond^{\forall} t$	universal deconstructor
			$\mid M : G$	coercion

4.2.2 LPL Inference Rules with Proof Terms

We already detailed LPL proof system and introduced inference rules in Section 3.3. In addition to these, in Figure 4.2 and Figure 4.3, we give all LPL inference rules incorporated proof terms.

Reduction

$$\frac{\Gamma; \Delta_I \setminus \Delta_M \xRightarrow{F} M_1 : G_1 \quad \Gamma; \Delta_M \setminus \Delta_O \xRightarrow{F} M_2 : G_2}{\Gamma; \Delta_I \setminus \Delta_O \xRightarrow{F} M_1 \otimes M_2 : G_1 \otimes G_2}\ \textbf{bl-}\otimes$$

$$\frac{\Gamma; \Delta_I, (u : {}^{[u]}D) \setminus \Delta_O \xRightarrow{F} M : G \qquad \text{if } (v : {}^{L}D_k) \in \Delta_O \text{ and } v \notin \Delta_I, \text{ then } u \notin L}{\Gamma; \Delta_I \setminus \Delta_O \xRightarrow{F} \hat{\lambda} u.M : D \multimap G}\ \textbf{bl-}\multimap$$

$$\frac{\Gamma, (u : {}^{[u]}D); \Delta_I \setminus \Delta_O \xRightarrow{F} M : G \qquad \text{if } (v : {}^{L}D_k) \in \Delta_O, \text{ then } u \notin L}{\Gamma; \Delta_I \setminus \Delta_O \xRightarrow{F} \lambda u.M : D \supset G}\ \textbf{bl-}\supset$$

$$\frac{\Gamma; \cdot \setminus \Delta_O \xRightarrow{\emptyset} M : G}{\Gamma; \Delta_I \setminus \Delta_I \xRightarrow{F} !M : !G}\ \textbf{bl-!}$$

$$\frac{\Gamma; \Delta_I \setminus \Delta_O \xRightarrow{F} M : [c/x]G}{\Gamma; \Delta_I \setminus \Delta_O \xRightarrow{F} \lambda^{\forall} x.([x/c]M) : \forall x.\, G}\ \textbf{bl-}\forall \qquad \frac{\Gamma; \Delta_I \setminus \Delta_O \xRightarrow{F} M : [t/x]G}{\Gamma; \Delta_I \setminus \Delta_O \xRightarrow{F} M \diamond^{\exists} t : \exists x.\, G}\ \textbf{bl-}\exists$$

Figure 4.2: Reduction rules with proof annotations for the **BL** proof theory.

4.3 Proof Rewriting

4.3.1 An "Intermediate" Proof System for Proof Transformations

In order to support our proof term rewriting rules, this section introduces an "intermediate" sequent calculus system, which we call the System IL, as a superset both the basic system of Section 4.1 and the backchaining system we proposed in Section 4.2. This system allows us to clearly define proof rewriting transformations under which it remains invariant. Transformation of a proof within the System **BL** to a proof within the System RL corresponds to the soundness of our backchaining proof theory while the opposite direction establishes completeness. The latter is considerably harder since arbitrary proofs in the System RL lack the rigid structure imposed by the System **BL**. We illustrate this transformation in Figure 4.4. We first start with a **BL** proof $\mathcal{E}_0$, and then eliminate resolution rules resulting a proof $\mathcal{E}_0^*$ in IL. Afterwards, we continue applying appropriate proof rewriting rules in IL and then finally reach to a RL proof $\mathcal{E}_f$.

Resolution

$$\frac{u : {}^{L}D \gg a \setminus (g_1 : G_1^{F_1}, ..., g_n : G_n^{F_n}) \triangleright \Delta_{O,0} \quad ... \quad \Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \xRightarrow{F_k \cup F} M_k : G_k \quad ...}{\Gamma; \Delta_I, (u : {}^{L}D) \setminus \mathrm{AddLb}(L, \Delta_{O,n}, F) \xRightarrow{F} ([M_n/g_n]...[M_1/g_1]\ M) : a}\ \textbf{bl-lin}$$

$$\frac{v : {}^{L\cup\{v\}}D \gg a \setminus (g_1 : G_1^{F_1}, ..., g_n : G_n^{F_n}) \triangleright \Delta_{O,0} \quad ... \quad \Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \xRightarrow{F_k \cup F} M_k : G_k \quad ...}{\Gamma, (u : {}^{L}D); \Delta_I \setminus \mathrm{AddLb}(L, \Delta_{O,n}, F) \xRightarrow{F} \text{let } v = u \text{ in } ([M_n/g_n]...[M_1/g_1]\ M) : a}\ \textbf{bl-int}$$

Residuation

$$\frac{\mathrm{unify}(a', a)}{u : {}^{L}(a') \gg e(u) : a \setminus \cdot \triangleright \cdot}\ \mathbf{d}_{lpl}\textbf{-atm}$$

$$\frac{v : {}^{L\cup\{v\}}D \gg M : a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(G_1 \multimap D) \gg (\text{let } v = u\hat{\ }j \text{ in } M) : a \setminus (j : G_1^{Lb(\Delta_O)}), \Delta_G \triangleright \Delta_O}\ \mathbf{d}_{lpl}\text{-}\multimap$$

$$\frac{v_1 : {}^{L\cup\{v_1\}}D_1 \gg M : a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(D_1 \otimes D_2) \gg (\text{def}_1\ v_1 \otimes v_2 = u \text{ in } M) : a \setminus \Delta_G \triangleright \Delta_O, (v_2 : {}^{L\cup\{v_2\}}D_2)}\ \mathbf{d}_{lpl}\text{-}\otimes_{1L}$$

$$\frac{v_1 : {}^{L\cup\{v1\}}D_2 \gg M : a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(D_1 \otimes D_2) \gg (\text{def}_2\ v_1 \otimes v_2 = u \text{ in } M) : a \setminus \Delta_G \triangleright \Delta_O, (v_2 : {}^{L\cup\{v_2\}}D_1)}\ \mathbf{d}_{lpl}\text{-}\otimes_{1R}$$

$$\frac{v : {}^{L\cup\{v\}}([t/x]D) \gg M : a \setminus \Delta_G \triangleright \Delta_O}{u : {}^{L}(\forall x.\ D) \gg \text{let } v = u \diamond^{\forall} t \text{ in } M : a \setminus \Delta_G \triangleright \Delta_O}\ \mathbf{d}_{lpl}\text{-}\forall$$

Figure 4.3: Resolution and residuation rules with proof annotations for the **BL** proof theory.

Giving just rewriting rules is not sufficient, one must also provide an algorithm to guide applications of these rules, resulting in a proof within the appropriate subset of the System IL. In this section, we only describe the System IL and proceed with the rewriting algorithm in Section 4.3.3.

This intermediate logic incorporates a new judgement

$$\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{E} M : G \tag{4.1}$$

similar to the one defined in (3.1) except the distinguishing single arrow, together with the same residuation judgement from System **BL**

$$u : {}^{L}D \gg \gamma \setminus \Delta_G \triangleright \Delta_O\ .$$

This intermediate proof system incorporates all **BL** rules from Figures 4.2 and

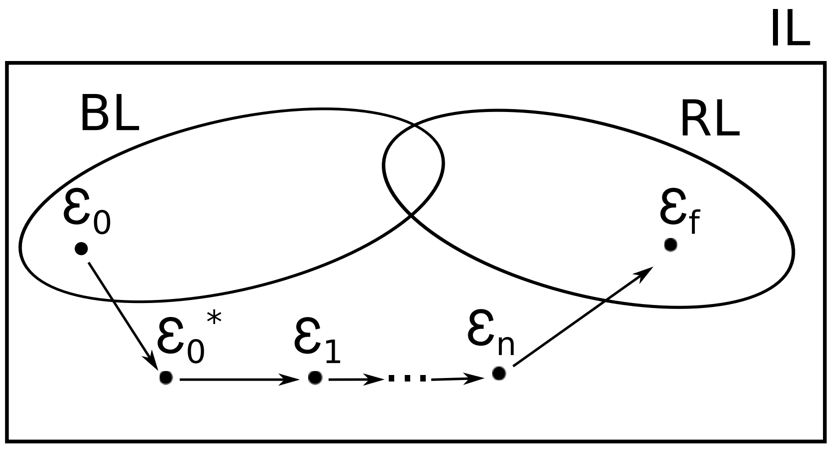

Figure 4.4: Proof Rewriting procedure.

4.3, adapted to the new judgement of (4.1) as well as adaptations of the left rules and init rules from the reference system RL as detailed in Figure 4.5. We should also note that, in the System IL we have three rules for decomposition of the simultaneous conjunction in resources. One rule is for when all decomposed resources are used and other two rules are for when only one of decomposed resources (left or right) is used.

4.3.2 Proof Rewriting Rules

We adopt a standard format to present all proof rewriting rules we use within the soundness proof. All of our rewriting rules use the following notation:

$$\boxed{\begin{array}{c} \mathcal{E} \\[1ex] \downarrow \quad \begin{array}{l} \{\text{precondition}\} \\ \underline{\text{rulename}} \end{array} \\[1ex] \mathcal{D} \end{array}} \tag{4.2}$$

meaning that the proof $\mathcal{E}$ can be transformed into a proof $\mathcal{D}$ when the optional *precondition* is satisfied and the rewriting rule *rulename* is applied. Note, also, that for visual simplicity and clarity, we will often omit dependency labels L associated with resources in contexts unless they are particularly relevant. The actual rewriting rules are presented in Section 4.3.4.

Hypotheses: IL

$$\frac{u \notin F}{\Gamma; (\Delta_I, u : {}^{L}a) \setminus \mathrm{AddLb}(L, \Delta_I, F) \xrightarrow{F} e(u) : a}\ \textbf{il-init}$$

$$\frac{(\Gamma, u : {}^{L}D); (\Delta_I, v : {}^{L\cup\{v\}}D) \setminus \Delta_O \xrightarrow{F} M : G \qquad v \notin \Delta_O}{(\Gamma, u : {}^{L}D); \Delta_I \setminus \Delta_O \xrightarrow{F} (\text{let } v = u \text{ in } M) : G}\ \textbf{il-copy}$$

Left Rules

$$\frac{\Gamma; \Delta_I \setminus \Delta' \xrightarrow{F} M : G \qquad \Gamma; (\Delta', iu : {}^{L\cup\{iu\}}D) \setminus \Delta_O \xrightarrow{F} N : C \quad (iu : D) \notin \Delta_O \quad u \notin F}{\Gamma; (\Delta_I, u : {}^{L}(G \multimap D)) \setminus \mathrm{UpdDep}(iu, \Delta_I - \Delta', \Delta_O) \xrightarrow{F} (\text{let } iu = u\hat{\ }M \text{ in } N) : C}\ \textbf{il-}\multimap\mathrm{L}$$

$$\frac{\Gamma; (\Delta_I, lu : {}^{L\cup\{lu\}}D_1, ru : {}^{L\cup\{ru\}}D_2) \setminus \Delta_O \xrightarrow{F} M : G \qquad (lu : D_1) \notin \Delta_O \quad (ru : D_2) \notin \Delta_O \quad u \notin F}{\Gamma; (\Delta_I, u : {}^{L}(D_1 \otimes D_2)) \setminus \Delta_O \xrightarrow{F} (\text{let } lu \otimes ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\mathrm{L}_{00}$$

$$\frac{\Gamma; (\Delta_I, lu : {}^{L\cup\{lu\}}D_1) \setminus \Delta_O \xrightarrow{F} M : G \qquad (lu : D_1) \notin \Delta_O \quad u \notin F}{\Gamma; (\Delta_I, u : {}^{L}(D_1 \otimes D_2)) \setminus (\Delta_O, ru : {}^{L\cup\{ru\}}D_2) \xrightarrow{F} (\text{def}_1\ lu \otimes ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\mathrm{L}_{01}$$

$$\frac{\Gamma; (\Delta_I, ru : {}^{L\cup\{ru\}}D_2) \setminus \Delta_O \xrightarrow{F} M : G \qquad (ru : D_2) \notin \Delta_O \quad u \notin F}{\Gamma; (\Delta_I, u : {}^{L}(D_1 \otimes D_2)) \setminus (\Delta_O, lu : {}^{L\cup\{lu\}}D_1) \xrightarrow{F} (\text{def}_2\ lu \otimes ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\mathrm{L}_{10}$$

$$\frac{\Gamma; (\Delta_I, au : {}^{L\cup\{au\}}([t/x]D)) \setminus \Delta_O \xrightarrow{F} M : G \quad (au : [t/x]D) \notin \Delta_O \quad u \notin F}{\Gamma; (\Delta_I, u : {}^{L}(\forall x.\ D)) \setminus \Delta_O \xrightarrow{F} (\text{let } au = u \diamond^{\forall} t \text{ in } M) : G}\ \textbf{il-}\forall\mathrm{L}$$

Notes:

- $\mathrm{UpdDep}(iu, \Delta_u, \Delta)' := \{u : {}^{L'}D \mid (u : {}^{L}D \in \Delta, L' = L \text{ if } iu \notin L,\ L' = L \cup \mathrm{DepLb}(\Delta_u)) \text{ otherwise } \}$, augments dependency labels of partial leftover resources in Δ that depend on iu with dependencies of resources in Δ_u, denoting resources consumed by the subgoal proof.
- $\mathrm{DepLb}(\Delta) := \bigcup_{(u:{}^{L}D)\in\Delta} (L)$ returns the union of dependency labels of all resources in Δ.

Figure 4.5: System IL: Initial and left sequent calculus rules.

4.3.3 Proof Rewriting Procedure

4.3.3.1 Elimination of Resolution and Residuation Rules (IL $\rightarrow$ IL*)

The first step in converting a proof in the System **BL** to a proof in the System RL is to eliminate all instances of the resolution rule by transforming all associated residuation rule applications into left rule applications within the intermediate system IL. The following theorem gets us started.

Theorem 1 *(Soundness of* **BL** *wrt IL) If* $\Gamma; \Delta_I \setminus \Delta_O \overset{F}{\Longrightarrow} M : G$ *is provable within the system* **BL***, then the sequent* $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{F} M : G$ *is provable within* *the system IL.*

Proof 1 *Trivially true since the system IL incorporates all rules of the system* **BL**.

We now define a constrained proof system, the System IL*, as the target of our simplifying rewrite procedure.

Definition 1 *The System IL* is defined as the proof system incorporating of all rules within System IL except the resolution rules,* **bl-lin** *and* **bl-int***, and the residuation rules shown in Figure 4.3. As such, it only consists of right and left rules for each connective together with the initial rule* **il-init** *and the copy rule* **il-copy***, given in Figure 4.5.*

Transformation of arbitrary proofs in the System IL into proofs in the System IL* is straightforward and involves recursive application of the rewrite rules presented in Section 4.3.4.1 to eliminate all occurences of the resolution rules as shown by the following theorem.

Theorem 2 *(Soundness of IL wrt IL*) If the judgement $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{F} M : G$ is provable within the system IL, then the same sequent $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{F} M : G$ is provable within the system IL*.*

Proof 2 *By induction on the structure of the derivation $\mathcal{E}$. All rules in the System IL except the resolution rule are trivially handled. When a resolution rule (with a particular residuation rule on its left premise) is encountered, the corresponding rewrite rule of Section 4.3.4.1 is applied. Combined with the inductive hypothesis on the smaller proof ending with a smaller resolution judgement, this rewrite yields a new proof with all resolution and residuation instances eliminated.*

Before we conclude this section, we establish a few theorems that will substantially facilitate the presentation of the rewrite rules in subsequent sections.

Theorem 3 *(Elimination of forbidden resources) Suppose that the judgement* $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{F} M : G$ *has a proof* $\mathcal{E}$ *of in the System IL**. *The forbidden resource sets of all sequents appearing in* $\mathcal{E}$ *are identical and equal to* F. *Moreover, the sequent* $\Gamma; \Delta_I - \Delta' \setminus \Delta_O - \Delta'' \xrightarrow{\emptyset} M : G$, *where* $\Delta' := \{u : {}^L D \in \Delta_I \mid u \in F\}$ *and* $\Delta'' := \{u : {}^L D \in \Delta_O \mid u \in F\}$ *is also provable in the System IL**.

Proof 3 *Straightforward induction on the structure of* $\mathcal{E}$ *since all rules of System IL** *use the same forbidden resource sets* F *in their premises and conclusions. Since left rules are restricted in their choice of principal formula to not be in* F, *a similar structural induction also establishes that the reduced sequent is also provable in the System IL**.

As a result of this theorem, we will be able to omit forbidden resource annotations in our presentation of rewriting rules in subsequent sections. This is also justified by the fact that when a proof is found for goal formula G in the System **BL**, it corresponds to the proof of a sequent $\cdot; \cdot \setminus \cdot \overset{\emptyset}{\Longrightarrow} M : G$ without any forbidden resources.

4.3.3.2 Elimination of Partial Leftover Resources (IL* $\rightarrow$ IL**)

Once all resolution rule instances are eliminated, the resulting proof is restricted to the System IL*, consisting only of right and left sequent rules of Figure 4.2 and Figure 4.1, respectively. The next step is to eliminate all instances of the rules **il-**$\otimes$L$_{01}$ and **il-**$\otimes$L$_{10}$ by transforming them to instances of the rule **il-**$\otimes$L$_{00}$ through the commuting rewrite rules described in Section 4.3.5. Once these transformations are completed, the resulting proof will also be a valid proof within the System RL. We begin by defining our target proof system.

Definition 2 *The System IL*** *is defined as the proof system incorporating of all rules within System IL** *except the two incomplete left residuation rules* **il-**$\otimes L_{01}$ *and* **il-**$\otimes L_{10}$ *shown in Figure 4.5.*

The System IL* has a number of properties that will help us establish its soundness with respect to the reference System RL.

Theorem 4 *(Subcontext for IL**) If the judgement* $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{\emptyset} M : G$ *is provable within the System IL**, then* $\Delta_O \subseteq \Delta_I$.

Proof 4 *Straightforward induction on the derivation for* $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{\emptyset} M : G$ *confined to the System IL**.*

The following theorem establishes that this is indeed the correct target system since it is equivalent to the reference System RL.

Theorem 5 *(Soundness of IL** wrt RL) If the judgement* $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{\emptyset} M : G$ *is provable within the System IL**, then the sequent* $\Gamma; \Delta_I \setminus \Delta_O \Longrightarrow M : G$ *is provable within the System RL.*

Proof 5 *To be completed by is fairly straightforward with Theorem 4 helping with the translation of rules in the System IL** to their counterparts in the System RL.*

As a result of this theorem, once we describe an algorithm that can rewrite proofs in the System IL* into proofs in the System IL**, we will have all the components necessary to establish the soundness of the System **BL** with respect to the reference System RL.

The key to this transformation is to systematically eliminate all instances of the rules **il-**$\otimes$L$_{01}$ and **il-**$\otimes$L$_{10}$. To this end, the first step is to "extract" sequences of left rule applications starting with these rules to the root of a derivation. When such a sequence of left rule applications occurs at the left branch of either **il-**$\otimes$ or **il-**$\multimap$L, with the right branch consuming the partial leftover resource from the left branch, commuting it to lie above the **il-**$\otimes$L$_{01}$ or **il-**$\otimes$L$_{10}$ instance associated with the partial leftover will enable us to transform this rule to a **il-**$\otimes$L$_{10}$ instance. The following theorem formalizes this procedure.

Theorem 6 *(Elimination of Partial Resources for IL*) Suppose that the judgement $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{\emptyset} M : G$ has a derivation $\mathcal{E}$ in the System IL*.*

(A) (Elimination) There exists a derivation $\mathcal{E}'$ for $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{\emptyset} M : G$ in the System IL* such that the only occurences of rules **il-**$\otimes L_{01}$ *or* **il-**$\otimes L_{10}$ *are those that generate the partial resources in $\Delta_O - \Delta_I$.*

(B) (Extraction) If there is a partial resource output $(u : {}^L D) \in \Delta_O$ such that $(u : {}^L D) \notin \Delta_I$, then there exists a derivation $\mathcal{E}''$ for $\Gamma; \Delta_I \setminus \Delta_O \xrightarrow{\emptyset} M : G$ in the System System IL* which ends with a connected sequence of focused left rule applications decomposing a single resource, beginning with either **il-**$\otimes L_{01}$ *or* **il-**$\otimes L_{10}$*, generating the partial leftover resource $(u : {}^L D)$.*

Proof 6 *The proof proceeds by nested induction for the two clauses above on the total number of rule applications in the derivation $\mathcal{E}$. In proving (B), (A) is first invoked on smaller derivations associated with the premises of the last rule in $\mathcal{E}$. Since the rewrite rules of Section 4.3.5 preserve the size of derivations, the inductive hypotheses of (B) can then be invoked on these rewritten smaller derivations. The last rule in the subderivation generating the partial leftover resource is then commuted up above the associated sequence of left rules, yielding the proof $\mathcal{E}''$ for (B). Once $\mathcal{E}''$ is obtained, its last rule is commuted up above the generating sequence of left rules, yielding the derivation $\mathcal{E}'$ for (A).*

Now that the rewriting algorithm is established, the following theorem concludes the soundness proof.

Theorem 7 *(Partial soundness of IL* wrt IL**) If the judgement $\Gamma; \cdot \setminus \cdot \xrightarrow{\emptyset} M : G$ is provable within the System IL*, then the same sequent is derivable within the System IL**.*

Proof 7 *Using clause (A) in Theorem 6 on the derivation of $\Gamma; \cdot \setminus \cdot \xrightarrow{\emptyset} M : G$, we can obtain a derivation $\mathcal{E}'$ without any instances of* **il-**$\otimes L_{01}$ *or* **il-**$\otimes L_{10}$ *since $\Delta_O - \Delta_I = \cdot$. By definition, $\mathcal{E}'$ is a derivation in the System IL**.*

4.3.4 Proof Rewriting Rules for Soundness

In this section, we introduce all proof rewriting rules for soundness of our backchaining logic **BL**. We should note that, we use Δ_G in rules instead of extended version $(g_1 : G_1^{F_1}, ..., g_n : G_n^{F_n})$.

4.3.4.1 Elimination of Residuation Judgements

Unrestricted resolution rules can be replaced by a copy rule followed by linear resolution.

$$\boxed{\begin{array}{c}
\dfrac{\overset{\mathcal{E}}{v : {}^{L}D \gg M : a \setminus \Delta_G \triangleright \Delta_{O0}} \quad ... \quad \overset{\mathcal{D}_k}{\Gamma; \Delta_{Ik} \setminus \Delta_{Ok} \xrightarrow{F_k \cup F} M_k : G_k} \quad ...}{\Gamma, (u : {}^{L}D); \Delta_I \setminus \Delta_O \xrightarrow{F} (\text{let } v = u \text{ in } ([M_n/g_n]...[M_1/g_1]\ M)) : a}\ \textbf{il-int} \\[2ex]
\downarrow\ \underline{\text{elimresint}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{v : {}^{L}D \gg M : a \setminus \Delta_G \triangleright \Delta_{O0}} \quad ... \quad \overset{\mathcal{D}_k}{\Gamma; \Delta_{Ik} \setminus \Delta_{Ok} \xrightarrow{F_k \cup F} M_k : G_k} \quad ...}{\Gamma, (u : {}^{L}D); \Delta_I, (v : {}^{L}D) \setminus \Delta_O \xrightarrow{F} ([M_n/g_n]...[M_1/g_1]\ M) : a}\ \textbf{il-lin}}{\Gamma, (u : {}^{L}D^{)}; \Delta_I \setminus \Delta_O \xrightarrow{F} \text{let } v = u \text{ in } ([M_n/g_n]...[M_1/g_1]\ M) : a}\ \textbf{il-copy}
\end{array}}$$

Linear resolution rules are always preceded by a residuation rule instance. Associated rewriting rules transform each residuation rule into an associated left rule, shifting the resolution rule one level up in the proof until it becomes **il-init**. Note, that for clarity, we omit dependency labels of resources in presenting the rewriting rules unless they are particularly relevant.

Rewriting of residuation rule $\mathbf{d}_{lpl}$-$\forall$, named elimresforall, takes the form

$$\cfrac{\cfrac{\overset{\mathcal{E}}{v : {}^{L\cup\{v\}}([t/x]D) \gg a \setminus \Delta_G \triangleright \Delta_O}}{u : {}^{L}(\forall x.\ D) \gg \text{let } v = u \diamond^{\forall} t \text{ in } M : a \setminus \Delta_G \triangleright \Delta_O}\ \mathbf{d}_{lpl}\textbf{-}\forall \quad \ldots \quad \overset{\mathcal{D}_k}{\Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \xrightarrow{F_k \cup F} M_k : G_k} \quad \ldots}{\Gamma; \Delta_I, (u : {}^{L}(\forall x.\ D)) \setminus \Delta_O \xrightarrow{F} ([M_n/g_n]\ldots[M_1/g_1]\ (\text{let } v = u \diamond^{\forall} t \text{ in } M)) : a}\ \textbf{il-lin}$$

$$\downarrow\ \underline{\text{elimresforall}}$$

$$\cfrac{\cfrac{\overset{\mathcal{E}}{v : {}^{L\cup\{v\}}([t/x]D) \gg M : a \setminus \Delta_G \triangleright \Delta_{O,0}} \quad \ldots \quad \overset{\mathcal{D}_k}{\Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \xrightarrow{F_k \cup F} M_k : G_k} \quad \ldots}{\Gamma; \Delta_I, (v : {}^{L\cup\{v\}}([t/x]D)) \setminus \Delta_O \xrightarrow{F} ([M_n/g_n]\ldots[M_1/g_1]\ M) : a}\ \textbf{il-lin}}{\Gamma; \Delta_I, (u : {}^{L}(\forall x.\ D)) \setminus \Delta_O \xrightarrow{F} (\text{let } v = u \diamond^{\forall} t \text{ in } ([M_n/g_n]\ldots[M_1/g_1]\ M)) : a}\ \textbf{il-}\forall\text{L}$$

In contrast, rewriting rules for instances of rules $\mathbf{d}_{lpl}\textbf{-}\otimes_{1L}$ and $\mathbf{d}_{lpl}\textbf{-}\otimes_{1R}$ require more care in managing resources. When the conjunct that is leftover from the residuation judgement is also left unused by subgoal proofs (i.e. $v_2 : D_2$ appears on the output), the rewriting rule takes the form

$$\cfrac{\cfrac{\overset{\mathcal{E}}{v_1 : D_1 \gg M : \gamma \setminus \Delta_G \triangleright \Delta_{O0}}}{u : (D_1 \otimes D_2) \gg \text{def}_1\ v_1 \otimes v_2 = u \text{ in } M : a \setminus \Delta_G \triangleright \Delta_{O0}, (v_2 : D_2)}\ \mathbf{d}_{lpl}\textbf{-}\otimes_{1L} \quad \ldots \quad \overset{\mathcal{D}_k}{\Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \xrightarrow{F_k \cup F} M_k : G_k} \quad \ldots}{\Gamma; \Delta_I, (u : (D_1 \otimes D_2) \setminus \Delta_O, (v_2 : D_2) \xrightarrow{F} ([M_n/g_n]\ldots[M_1/g_1]\ (\text{def}_1\ v_1 \otimes v_2 = u \text{ in } M)) : a}\ \textbf{il-lin}$$

$$\downarrow\ \underline{\text{elimressconj1p}}$$

$$\cfrac{\cfrac{\overset{\mathcal{E}}{v_1 : D_1 \gg M : a \setminus \Delta_G \triangleright \Delta_{O0}} \quad \ldots \quad \overset{\mathcal{D}'_k}{\Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \xrightarrow{F_k \cup F} M_k : G_k} \quad \ldots}{\Gamma; \Delta_I, v_1 : D_1 \setminus \Delta_O \xrightarrow{F} ([M_n/g_n]\ldots[M_1/g_1]\ M) : a}\ \textbf{il-lin}}{\Gamma; \Delta_I, u : (D_1 \otimes D_2) \setminus \Delta_O, (v_2 : D_2) \xrightarrow{F} (\text{def}_1\ v_1 \otimes v_2 = u \text{ in } ([M_n/g_n]\ldots[M_1/g_1]\ M)) : a}\ \textbf{il-}\otimes\text{L}_{01}$$

Since this resource is unused in the subgoal proofs, we can safely remove it from all input and output contexts within subgoal proofs $\mathcal{D}_k$ and their conclusions to yield the updated proofs $\mathcal{D}'_k$. This is also consistent with how it ends up in the resulting instance of the $\textbf{il-}\otimes\text{L}_{01}$ rule.

When this side condition does not hold, the transformation takes us directly to a let form. The associated rewriting rule takes the form

$$\dfrac{\dfrac{\overset{\mathcal{E}}{v_1 : D_1 \gg M : a \setminus \Delta_G \triangleright \Delta'}}{u : D_1 \otimes D_2 \gg (\text{def}_1\ v_1 \otimes v_2 = u \text{ in } M) : a \setminus \Delta_G \triangleright \Delta', ru : D_2}\ \mathbf{d}_{lpl}\textbf{-}\otimes_{1L} \quad \dots \quad \overset{\mathcal{D}_k}{\Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \xrightarrow{F_k \cup F} M_k : G_k} \quad \dots}{\Gamma; \Delta_I, u : (D_1 \otimes D_2) \setminus \Delta_O \xrightarrow{F} ([M_n/g_n]\dots[M_1/g_1]\ (\text{def}_1\ v_1 \otimes v_2 = u \text{ in } M)) : a}\ \textbf{il-lin}$$

$$\downarrow\ \underline{\text{elimressconj1n}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{v_1 : D_1 \gg M : a \setminus \Delta_G \triangleright \Delta'} \quad \dots \quad \overset{\mathcal{D}_k}{\Gamma; \Delta_{I,k} \setminus \Delta_{O,k} \xrightarrow{F_k \cup F} M_k : G_k} \quad \dots}{\Gamma; \Delta_I, v_1 : D_1, v_2 : D_2 \setminus \Delta_O \xrightarrow{F} ([M_n/g_n]\dots[M_1/g_1]\ M) : a}\ \textbf{il-lin}}{\Gamma; \Delta_I, u : (D_1 \otimes D_2) \setminus \Delta_O \xrightarrow{F} (\text{let } v_1 \otimes v_2 = u \text{ in } ([M_n/g_n]\dots[M_1/g_1]\ M)) : a}\ \textbf{il-}\otimes\text{L}_{00}$$

The most critical residuation rule in this elimination process is **d-** $\multimap$. The rewriting rule for occurrences of this rule takes the form

$$\dfrac{\dfrac{\overset{\mathcal{E}}{v : D \gg M : a \setminus \Delta_G \triangleright \Delta_{O0}}}{u : G_1 \multimap D \gg (\text{let } v = u\hat{\ }g_1 \text{ in } M) : a \setminus (g_1 : G_1^{F_1}), \Delta_G \triangleright \Delta_{O0}}\ \textbf{d-}\multimap \quad \overset{\mathcal{D}_1}{\Gamma; \Delta_{I1} \setminus \Delta_{O1} \xrightarrow{F_1 \cup F} M_1 : G_1} \quad \dots}{\Gamma; \Delta_I, (u : G_1 \multimap D) \setminus \Delta_O \xrightarrow{F} ([M_n/g_n]\dots[M_1/g_1]\ (\text{let } v = u\hat{\ }g_1 \text{ in } M)) : a}\ \textbf{il-lin}$$

$$\downarrow\ \underline{\text{elimreslimp}}$$

$$\dfrac{\overset{\mathcal{D}_1'}{\Gamma; (\Delta_{I1} - \Delta_{O0}) \setminus (\Delta_{O1} - \Delta_{O0}) \xrightarrow{F} M_1 : G_1} \quad \dfrac{\overset{\mathcal{E}}{v : D \gg M : a \setminus \Delta_G \triangleright \Delta_{O0}} \quad \overset{\mathcal{D}_2}{\Gamma; \Delta_{I2} \setminus \Delta_{O2} \xrightarrow{F_2 \cup F} M_2 : G_2} \quad \dots}{\Gamma; (\Delta_{O1} - \Delta_{O0}), (v : D) \setminus \Delta_O \xrightarrow{F} ([M_n/g_n]\dots[M_2/g_2]\ M) : a}\ \textbf{il-lin}}{\Gamma; \Delta_I, (u : G_1 \multimap D) \setminus \text{UpdDep}(v, \Delta_{I1} - \Delta_{O1}, \Delta_O) \xrightarrow{F} (\text{let } v = u\hat{\ }M_1 \text{ in } ([M_n/g_n]\dots[M_2/g_2]\ M)) : a}\ \textbf{il-}\multimap\text{L}$$

where $F_1 := Lb(\Delta_{O0})$. Since the subgoal proof for G_1 is not allowed to use any of the outputs from the residuation judgement, Δ_{O0}, its proof $\mathcal{D}_1$ can be transformed into a proof with these leftover judgements removed from all input and output contexts to yield an update proof $\mathcal{D}_1'$. This also allows removing the additional forbidden resource labels $F_1 = Lb(\Delta_{O0})$. However, since resources used within this subgoal will no longer be in the scope of the AddLb() function associated with the output of resolution rule, the UpdDep() function is necessary to update the leftover resources that depend on v. Finally, since the implication left rule is constructed such that the subgoal is proven first, the order in which the resolution judgement attempts the subgoals is preserved, allowing a rather straightforward rewriting of this rule.

The remaining residuation rule forms base case of the recursive rewriting process to eliminate resolution rules. When this is encountered, the associated rule can be eliminated through the following rewrite rule.

$$
\boxed{
\begin{array}{c}
\dfrac{\dfrac{}{u : {}^{L}a \gg e(u) : a \setminus \cdot \triangleright \cdot}\ \textbf{d-atm} \qquad u \notin F}{\Gamma; \Delta_I, (u : {}^{L}a) \setminus \mathrm{AddLb}(L, \Delta_I, F) \xrightarrow{F} e(u) : a}\ \textbf{il-lin} \\[2ex]
\downarrow\ \underline{\text{elimresatm}} \\[2ex]
\dfrac{u \notin F}{\Gamma; \Delta_I, (u : {}^{L}a) \setminus \mathrm{AddLb}(L, \Delta_I, F) \xrightarrow{F} e(u) : a}\ \textbf{il-init}
\end{array}
}
$$

Since this rewriting rule completely eliminates the resolution rule, the resulting init rule incorporates a call to the AddLb() function, making sure that dependencies of resources whose presence depends on this proof branch are properly updated.

Repeated application of this rewrite rule can be used to eliminate all instances of both linear and unrestricted resolution rules and associated residuation rules. We call the proof system that results from removing the resolution and residuation rules from the System IL as the System IL*, consisting only from left and right rules for each connective as well as the initial sequents and the copy rule. All of our rewrite rules in the remainder of the soundness proof will operate within this restricted subsystem.

4.3.5 Commutativity of Rules within System IL*

Our presentation of the rewriting rules below is categorized by the last rule in the proof to be transformed. Note, also, that right rules do not commute since they decompose a single goal. Similarly, left rules which focus on the same resource (such as those resulting from the same residuation judgement) do not commute either. Since the IL* proofs resulting from the transformations of the previous section have left rule applications aggregated towards the leaves of the proof tree, the rewrite rules we describe below focus on shifting them down in the proof tree.

4.3.5.1 Proofs ending with il-∀

Possible cases are categorized by the preceding rule.

Case il-$\otimes$L$_{01}$:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow M : [c/x]G} \qquad lu \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu {\otimes} ru = u \text{ in } M) : [c/x]G}\ \textbf{il-}\otimes\text{L}_{01}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow \lambda^{\forall} x.([x/c](\text{def}_1\ lu {\otimes} ru = u \text{ in } M)) : \forall x.\ G}\ \textbf{il-}\forall \\[2ex]
\downarrow\ \underline{\text{sconjL01forall}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow M : [c/x]G}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow \lambda^{\forall} x.([x/c]M) : \forall x.\ G}\ \textbf{il-}\forall \qquad lu \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu {\otimes} ru = u \text{ in } (\lambda^{\forall} x.([x/c]M))) : \forall x.\ G}\ \textbf{il-}\otimes\text{L}_{01}
\end{array}}$$

Case il-$\otimes$L$_{10}$:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow M : [c/x]G} \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } M) : [c/x]G}\ \textbf{il-}\otimes\text{L}_{10}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow \lambda^{\forall} x.([x/c](\text{def}_2\ lu \otimes ru = u \text{ in } M)) : \forall x.\ G}\ \textbf{il-}\forall \\[2ex]
\downarrow\ \underline{\text{sconjL10forall}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow M : [c/x]G}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow \lambda^{\forall} x.([x/c]M) : \forall x.\ G}\ \textbf{il-}\forall \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } (\lambda^{\forall} x.([x/c]M))) : \forall x.\ G}\ \textbf{il-}\otimes\text{L}_{01}
\end{array}}$$

Case il-$\otimes$L$_{00}$:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow M : [c/x]G} \qquad lu \notin \Delta_O \quad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } M) : [c/x]G}\ \textbf{il-}\otimes\text{L}_{00}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow \lambda^{\forall}x.([x/c](\text{let } lu \otimes ru = u \text{ in } M)) : \forall x.\ G}\ \textbf{il-}\forall \\[2ex]
\downarrow\ \underline{\text{sconjL00forall}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow M : [c/x]G}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow \lambda^{\forall}x.([x/c]M) : \forall x.\ G}\ \textbf{il-}\forall \qquad lu \notin \Delta_O \quad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } (\lambda^{\forall}x.([x/c]M))) : \forall x.\ G}\ \textbf{il-}\otimes\text{L}_{00}
\end{array}}$$

Case il- $\multimap$ **L**:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I \setminus \Delta' \longrightarrow M : G} \qquad \overset{\mathcal{D}}{\Gamma; (\Delta', iu : D) \setminus \Delta_O \longrightarrow N : [c/x]G_1} \qquad iu \notin \Delta_O}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta_O \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } N) : [c/x]G_1}\ \textbf{il-}\multimap\text{L}}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta_O \longrightarrow \lambda^{\forall}x.([x/c](\text{let } iu = u\hat{\ }M \text{ in } N)) : \forall x.\ G_1}\ \textbf{il-}\forall \\[2ex]
\downarrow\ \underline{\text{limpLforall}} \\[2ex]
\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I \setminus \Delta' \longrightarrow M : G} \qquad \dfrac{\overset{\mathcal{D}}{\Gamma; (\Delta', iu : D) \setminus \Delta_O \longrightarrow N : [c/x]G_1}}{\Gamma; (\Delta', iu : D) \setminus \Delta_O \longrightarrow \lambda^{\forall}x.([x/c](N)) : \forall x.\ G_1}\ \textbf{il-}\forall \qquad iu \notin \Delta_O}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta_O - \{iu\} \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } (\lambda^{\forall}x.([x/c](N)))) : \forall x.\ G_1}\ \textbf{il-}\multimap\text{L}
\end{array}}$$

Case il-copy:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow M : [c/x]G} \qquad v \notin \Delta_O}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } M) : [c/x]G}\ \textbf{il-}\textit{copy}}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow \lambda^{\forall}x.([x/c](\text{let } v = u \text{ in } M)) : \forall x.\ G}\ \textbf{il-}\forall \\[2ex]
\downarrow\ \underline{\text{copyforall}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow M : [c/x]G}}{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow \lambda^{\forall}x.([x/c]M) : \forall x.\ G}\ \textbf{il-}\forall \qquad v \notin \Delta_O}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } (\lambda^{\forall}x.([x/c]M))) : \forall x.\ G}\ \textbf{il-}\textit{copy}
\end{array}}$$

Case il-$\forall$L:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, au : ([t/x]D)) \setminus \Delta_O \longrightarrow M : [c/x]G}}{\Gamma; (\Delta_I, u : (\forall x.\ D)) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t \text{ in } M) : [c/x]G}\ \textbf{il-}\forall\text{L}}{\Gamma; (\Delta_I, u : (\forall x.\ D)) \setminus \Delta_O \longrightarrow \lambda^{\forall}x.([x/c](\text{let } au = u \diamond^{\forall} t \text{ in } M)) : \forall x.\ G}\ \textbf{il-}\forall \\[2ex]
\downarrow\ \underline{\text{forallLforall}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, au : ([t/x]D)) \setminus \Delta_O \longrightarrow M : [c/x]G}}{\Gamma; (\Delta_I, au : ([t/x]D) \setminus \Delta_O \longrightarrow \lambda^{\forall}x.([x/c]M) : \forall x.\ G}\ \textbf{il-}\forall}{\Gamma; (\Delta_I, u : (\forall x.\ D)) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t \text{ in } (\lambda^{\forall}x.([x/c]M))) : \forall x.\ G}\ \textbf{il-}\forall\text{L}
\end{array}}$$

4.3.5.2 Proofs ending with il-$\exists$

Case il-$\otimes$L$_{01}$:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow M : [t/x]G} \qquad lu \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu{\otimes}ru = u \text{ in } M) : [t/x]G}\ \textbf{il-}\otimes\text{L}_{01}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu{\otimes}ru = u \text{ in } M) \diamond^{\exists} t : \exists x.\ G}\ \textbf{il-}\exists \\[2ex]
\downarrow\ \underline{\text{sconjL01exists}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow M : [t/x]G}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow M \diamond^{\exists} t : \exists x.\ G}\ \textbf{il-}\exists \qquad lu \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu{\otimes}ru = u \text{ in } (M \diamond^{\exists} t)) : \exists x.\ G}\ \textbf{il-}\otimes\text{L}_{01}
\end{array}}$$

Case il-$\otimes$L$_{10}$:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow M : [t/x]G} \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } M) : [t/x]G}\ \textbf{il-}\otimes\text{L}_{10}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } M) \diamond^{\exists} t : \exists x.\ G}\ \textbf{il-}\exists \\[2ex]
\downarrow\ \underline{\text{sconjL10exists}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow M : [t/x]G}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow M \diamond^{\exists} t : \exists x.\ G}\ \textbf{il-}\exists \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } (M \diamond^{\exists} t)) : \exists x.\ G}\ \textbf{il-}\otimes\text{L}_{01}
\end{array}}$$

Case il-$\otimes$L$_{00}$:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow M : [t/x]G} \qquad lu \notin \Delta_O \quad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } M) : [t/x]G}\,\textbf{il-}\otimes\text{L}_{00}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } M) \diamond^{\exists} t : \exists x.\, G}\,\textbf{il-}\exists \\[2ex]
\downarrow\ \underline{\text{sconjL00exists}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow M : [t/x]G}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow M \diamond^{\exists} t : \exists x.\, G}\,\textbf{il-}\exists \qquad lu \notin \Delta_O \quad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } (M \diamond^{\exists} t)) : \exists x.\, G}\,\textbf{il-}\otimes\text{L}_{00}
\end{array}}$$

Case il-$\multimap$L:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I \setminus \Delta' \longrightarrow M : G} \qquad \overset{\mathcal{D}}{\Gamma; (\Delta', iu : D) \setminus \Delta_O \longrightarrow N : [t/x]G_1} \qquad iu \notin \Delta_O}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta'_O \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } N) : [t/x]G_1}\,\textbf{il-}\multimap\text{L}}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta'_O \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } N) \diamond^{\exists} t : \exists x.\, G_1}\,\textbf{il-}\exists \\[2ex]
\downarrow\ \underline{\text{limpLexists}} \\[2ex]
\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I \setminus \Delta' \longrightarrow M : G} \qquad \dfrac{\overset{\mathcal{D}}{\Gamma; (\Delta', iu : D) \setminus \Delta_O \longrightarrow N : [t/x]G_1}}{\Gamma; (\Delta', iu : D) \setminus \Delta'_O \longrightarrow N \diamond^{\exists} t : \exists x.\, G_1}\,\textbf{il-}\exists \qquad iu \notin \Delta_O}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta'_O - \{iu\} \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } (N \diamond^{\exists} t)) : \exists x.\, G_1}\,\textbf{il-}\multimap\text{L}
\end{array}}$$

Case il-copy:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow M : [t/x]G} \qquad v \notin \Delta_O}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } M) : [t/x]G}\,\textbf{il-}\textit{copy}}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } M) \diamond^{\exists} t : \exists x.\, G}\,\textbf{il-}\exists \\[2ex]
\downarrow\ \underline{\text{copyexists}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow M : [t/x]G}}{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow M \diamond^{\exists} t : \exists x.\, G}\,\textbf{il-}\exists \qquad v \notin \Delta_O}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } (M \diamond^{\exists} t)) : \exists x.\, G}\,\textbf{il-}\textit{copy}
\end{array}}$$

Case il-$\forall$L:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, au : ([t'/x]D)) \setminus \Delta_O \longrightarrow M : [t/x]G}}{\Gamma; (\Delta_I, u : (\forall x.\ D)) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t' \text{ in } M) : [t/x]G}\ \textbf{il-}\forall\text{L}}{\Gamma; (\Delta_I, u : (\forall x.\ D)) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t' \text{ in } M) \diamond^{\exists} t : \exists x.\ G}\ \textbf{il-}\exists$$

$$\downarrow \ \underline{\text{forallLexists}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, au : ([t'/x]D)) \setminus \Delta_O \longrightarrow M : [t/x]G}}{\Gamma; (\Delta_I, au : ([t'/x]D) \setminus \Delta_O \longrightarrow M \diamond^{\exists} t : \exists x.\ G}\ \textbf{il-}\exists}{\Gamma; (\Delta_I, u : (\forall x.\ D)) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t' \text{ in } (M \diamond^{\exists} t)) : \exists x.\ G}\ \textbf{il-}\forall\text{L}$$

4.3.5.3 Proofs ending with il- $\multimap$

Case il-$\otimes$L$_{01}$:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1, v : D) \setminus \Delta_O \longrightarrow M : G} \qquad lu \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2, v : D) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu {\otimes} ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\text{L}_{01} \qquad v \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow \hat{\lambda} v.(\text{def}_1\ lu {\otimes} ru = u \text{ in } M) : D \multimap G}\ \textbf{il-}\multimap$$

$$\downarrow \ \underline{\text{sconjL01limpR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1, v : D) \setminus \Delta_O \longrightarrow M : G} \qquad v \notin \Delta_O}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow \hat{\lambda} v.M : D \multimap G}\ \textbf{il-}\multimap \qquad lu \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu {\otimes} ru = u \text{ in } (\hat{\lambda} v.M)) : D \multimap G}\ \textbf{il-}\otimes\text{L}_{01}$$

Case il-$\otimes$L$_{10}$:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, ru : D_2, v : D) \setminus \Delta_O \longrightarrow M : G} \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2, v : D) \setminus (\Delta_O, lu : D_1) \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\text{L}_{10} \qquad v \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow \hat{\lambda} v.(\text{def}_2\ lu \otimes ru = u \text{ in } M) : D \multimap G}\ \textbf{il-}\multimap$$

$$\downarrow \ \underline{\text{sconjL10limpR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, ru : D_2, v : D) \setminus \Delta_O \longrightarrow M : G} \qquad v \notin \Delta_O}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow \hat{\lambda} v.M : D \multimap G}\ \textbf{il-}\multimap \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } (\hat{\lambda} v.M)) : D \multimap G}\ \textbf{il-}\otimes\text{L}_{10}$$

Case il-$\otimes$L$_{00}$:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2, v : D) \setminus \Delta_O \longrightarrow M : G} \qquad lu \notin \Delta_O \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2, v : D) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\text{L}_{00} \qquad v \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow \hat{\lambda} v.(\text{let } lu \otimes ru = u \text{ in } M) : D \multimap G}\ \textbf{il-}\multimap$$

$$\downarrow\ \underline{\text{sconjL00limpR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2, v : D) \setminus \Delta_O \longrightarrow M : G} \qquad v \notin \Delta_O}{\Gamma; (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow \hat{\lambda} v.M : D \multimap G}\ \textbf{il-}\multimap \qquad lu \notin \Delta_O \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } (\hat{\lambda} v.M)) : D \multimap G}\ \textbf{il-}\otimes\text{L}_{00}$$

Case il-$\multimap$L:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, v : D_1) \setminus \Delta' \longrightarrow M : G} \qquad \overset{\mathcal{D}}{\Gamma; (\Delta', iu : D) \setminus \Delta_O \longrightarrow N : G_1} \qquad iu \notin \Delta_O}{\Gamma; (\Delta_I, u : G \multimap D, v : D_1) \setminus \Delta_O \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } N) : G_1}\ \textbf{il-}\multimap\text{L} \qquad v \notin \Delta_O}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta_O \longrightarrow \hat{\lambda} v.(\text{let } iu = u\hat{\ }M \text{ in } N) : D_1 \multimap G_1}\ \textbf{il-}\multimap$$

$$\downarrow\ \underline{\text{limpLlimpR}}$$

$$\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I \setminus \Delta' \longrightarrow M : G} \qquad \dfrac{\overset{\mathcal{D}}{\Gamma; (\Delta', iu : D, v : D_1) \setminus \Delta_O \longrightarrow N : G_1} \quad v \notin \Delta_O}{\Gamma; (\Delta', iu : D) \setminus \Delta_O \longrightarrow \hat{\lambda} v.(N) : D_1 \multimap G_1}\ \textbf{il-}\multimap \qquad iu \notin \Delta_O}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta_O - \{iu\} \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } (\hat{\lambda} v.(N))) : D_1 \multimap G_1}\ \textbf{il-}\multimap\text{L}$$

Case il-copy:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, u : D); (\Delta_I, v_1 : D_1, v : D) \setminus \Delta_O \longrightarrow M : G} \qquad v \notin \Delta_O}{(\Gamma, u : D); (\Delta_I, v_1 : D_1) \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } M) : G}\ \textbf{il-}\textit{copy} \qquad v_1 \notin \Delta_O}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow \hat{\lambda} v_1.(\text{let } v = u \text{ in } M) : D_1 \multimap G}\ \textbf{il-}\multimap$$

$$\downarrow\ \underline{\text{copylimpR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, u : D); (\Delta_I, v : D, v_1 : D_1) \setminus \Delta_O \longrightarrow M : G} \quad v_1 \notin \Delta_O}{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow \hat{\lambda} v_1.M : D_1 \multimap G}\ \textbf{il-}\multimap \qquad v \notin \Delta_O}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } (\hat{\lambda} v_1.M)) : D_1 \multimap G}\ \textbf{il-}\textit{copy}$$

Case il-$\forall$L:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ \Gamma; (\Delta_I, au : ([t/x]D), v : D_1) \setminus \Delta_O \longrightarrow M : G\end{array}}{\Gamma; (\Delta_I, u : (\forall x.\ D), v : D_1) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t \text{ in } M) : G}\ \textbf{il-}\forall\text{L} \qquad v \notin \Delta_O}{\Gamma; (\Delta_I, u : (\forall x.\ D)) \setminus \Delta_O \longrightarrow \hat{\lambda} v.(\text{let } au = u \diamond^{\forall} t \text{ in } M) : D_1 \multimap G}\ \textbf{il-}\multimap \\[2ex]
\downarrow\ \underline{\text{forallLlimpR}} \\[2ex]
\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ \Gamma; (\Delta_I, au : ([t/x]D), v : D_1) \setminus \Delta_O \longrightarrow M : G \quad v \notin \Delta_O\end{array}}{\Gamma; (\Delta_I, au : ([t/x]D) \setminus \Delta_O \longrightarrow \hat{\lambda} v.M : D_1 \multimap G}\ \textbf{il-}\multimap}{\Gamma; (\Delta_I, u : (\forall x.\ D)) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t \text{ in } (\hat{\lambda} v.(M))) : D_1 \multimap G}\ \textbf{il-}\forall\text{L}
\end{array}}$$

4.3.5.4 Proofs ending with il-$\supset$

Case il-$\otimes$L$_{01}$:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ (\Gamma, v : D); (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow M : G \qquad lu \notin \Delta_O\end{array}}{(\Gamma, v : D); (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu {\otimes} ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\text{L}_{01}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow \lambda v.(\text{def}_1\ lu {\otimes} ru = u \text{ in } M) : D \supset G}\ \textbf{il-}\supset \\[2ex]
\downarrow\ \underline{\text{sconjL01impR}} \\[2ex]
\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ (\Gamma, v : D); (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow M : G\end{array}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow \lambda v.M : D \supset G}\ \textbf{il-}\supset \qquad lu \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, ru : D_2) \longrightarrow (\text{def}_1\ lu {\otimes} ru = u \text{ in } (\lambda v.M)) : D \supset G}\ \textbf{il-}\otimes\text{L}_{01}
\end{array}}$$

Case il-$\otimes$L$_{10}$:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, v : D); (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow M : G} \qquad ru \notin \Delta_O}{(\Gamma, v : D); (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow (\mathrm{def}_2\ lu \otimes ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\mathrm{L}_{10}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow \lambda v.(\mathrm{def}_2\ lu \otimes ru = u \text{ in } M) : D \supset G}\ \textbf{il-}\supset$$

$$\downarrow\ \underline{\text{sconjL10impR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, v : D); (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow M : G}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow \lambda v.M : D \supset G}\ \textbf{il-}\supset \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus (\Delta_O, lu : D_1) \longrightarrow (\mathrm{def}_2\ lu \otimes ru = u \text{ in } (\lambda v.M)) : D \supset G}\ \textbf{il-}\otimes\mathrm{L}_{10}$$

Case il-$\otimes$L$_{00}$:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, v : D); (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow M : G} \qquad lu \notin \Delta_O \qquad ru \notin \Delta_O}{(\Gamma, v : D); (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } M) : G}\ \textbf{il-}\otimes\mathrm{L}_{00}}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow \lambda v.(\text{let } lu \otimes ru = u \text{ in } M) : D \supset G}\ \textbf{il-}\supset$$

$$\downarrow\ \underline{\text{sconjL00impR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, v : D); (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow M : G}}{\Gamma; (\Delta_I, lu : D_1, ru : D_2) \setminus \Delta_O \longrightarrow \lambda v.M : D \supset G}\ \textbf{il-}\supset \qquad lu \notin \Delta_O \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } (\lambda v.M)) : D \supset G}\ \textbf{il-}\otimes\mathrm{L}_{00}$$

Case il-$\multimap$L:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma, v : D_1); \Delta_I \setminus \Delta' \longrightarrow M : G} \qquad \overset{\mathcal{D}}{(\Gamma, v : D_1); (\Delta', iu : D) \setminus \Delta_O \longrightarrow N : G_1} \qquad iu \notin \Delta_O}{(\Gamma, v : D_1); (\Delta_I, u : G \multimap D) \setminus \Delta_O \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } N) : G_1}\ \textbf{il-}\multimap\mathrm{L}}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta_O \longrightarrow \lambda v.(\text{let } iu = u\hat{\ }M \text{ in } N) : D_1 \supset G_1}\ \textbf{il-}\supset$$

$$\downarrow\ \underline{\text{limpLimpR}}$$

$$\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I \setminus \Delta' \longrightarrow M : G} \qquad \dfrac{\overset{\mathcal{D}}{(\Gamma, v : D_1); (\Delta', iu : D) \setminus \Delta_O \longrightarrow N : G_1}}{\Gamma; (\Delta', iu : D) \setminus \Delta_O \longrightarrow \lambda v.(N) : D_1 \supset G_1}\ \textbf{il-}\supset \qquad iu \notin \Delta_O}{\Gamma; (\Delta_I, u : G \multimap D) \setminus \Delta_O - \{iu\} \longrightarrow (\text{let } iu = u\hat{\ }M \text{ in } (\lambda v.(N))) : D_1 \supset G_1}\ \textbf{il-}\multimap\mathrm{L}$$

Case il-copy:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\(\Gamma, u : D, v_1 : D_1); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow M : G \qquad v \notin \Delta_O\end{array}}{(\Gamma, u : D, v_1 : D_1); \Delta_I \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } M) : G}\;\textbf{il-}\mathit{copy}}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow \lambda v_1.(\text{let } v = u \text{ in } M) : D_1 \supset G}\;\textbf{il-}\supset \\[2ex]
\downarrow\; \underline{\text{copyimpR}} \\[2ex]
\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\(\Gamma, u : D, v_1 : D_1); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow M : G\end{array}}{(\Gamma, u : D); (\Delta_I, v : D) \setminus \Delta_O \longrightarrow \lambda v_1.M : D_1 \supset G}\;\textbf{il-}\supset \qquad v \notin \Delta_O}{(\Gamma, u : D); \Delta_I \setminus \Delta_O \longrightarrow (\text{let } v = u \text{ in } (\lambda v_1.M)) : D_1 \supset G}\;\textbf{il-}\mathit{copy}
\end{array}}$$

Case il-$\forall$L:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\(\Gamma, v : D_1); (\Delta_I, au : ([t/x]D)) \setminus \Delta_O \longrightarrow M : G\end{array}}{(\Gamma, v : D_1); (\Delta_I, u : (\forall x.\, D)) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t \text{ in } M) : G}\;\textbf{il-}\forall\text{L}}{\Gamma; (\Delta_I, u : (\forall x.\, D)) \setminus \Delta_O \longrightarrow \lambda v.(\text{let } au = u \diamond^{\forall} t \text{ in } M) : D_1 \supset G}\;\textbf{il-}\supset \\[2ex]
\downarrow\; \underline{\text{forallLimpR}} \\[2ex]
\dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\(\Gamma, v : D_1); (\Delta_I, au : ([t/x]D)) \setminus \Delta_O \longrightarrow M : G\end{array}}{\Gamma; (\Delta_I, au : ([t/x]D) \setminus \Delta_O \longrightarrow \lambda v.M : D_1 \supset G}\;\textbf{il-}\supset}{\Gamma; (\Delta_I, u : (\forall x.\, D)) \setminus \Delta_O \longrightarrow (\text{let } au = u \diamond^{\forall} t \text{ in } (\lambda v.(M))) : D_1 \supset G}\;\textbf{il-}\forall\text{L}
\end{array}}$$

4.3.5.5 Proofs ending with il-$\otimes$

Similar to earlier examples, possible cases are categorized by the preceding rule.

Case il-$\otimes$L$_{01}$ on the left branch:

Simultaneous conjunctions on the left branches are among the most important cases since they enable us to convert def-forms into let-forms. Depending on whether a partial leftover resource is used by the right branch of the right rule, we have two different rewrite rules, sconjL01sconjRA and sconjL01sconjRB.

The first rewrite rule, sconjL01sconjRA captures the case when the partial resource is left unused by the right branch and is defined as

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I, lu : D_1 \setminus \Delta' \longrightarrow M_1 : G_1} \qquad lu \notin \Delta'}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta', ru : D_2 \longrightarrow \text{def}_1\ lu{\otimes}ru = u \text{ in } M_1 : G_1}\ \textbf{il-}{\otimes}\text{L}_{01} \quad \overset{\mathcal{D}}{\Gamma; \Delta', ru : D_2 \setminus \Delta_O, ru : D_2 \longrightarrow M_2 : G_2}}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta_O, ru : D_2 \longrightarrow (\text{def}_1\ lu{\otimes}ru = u \text{ in } M_1) \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}{\otimes} \\[2ex]
\downarrow\ \underline{\text{sconjL01sconjRA}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta' \longrightarrow M_1 : G_1} \quad \overset{\mathcal{D}'}{\Gamma; \Delta' \setminus \Delta_O \longrightarrow M_2 : G_2}}{\Gamma; (\Delta_I, lu : D_1) \setminus \Delta_O \longrightarrow M_1 \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}{\otimes} \qquad lu \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O, ru : D_2 \longrightarrow (\text{def}_1\ lu{\otimes}ru = u \text{ in } (M_1 \otimes M_2)) : G_1 \otimes G_2}\ \textbf{il-}{\otimes}\text{L}_{01}
\end{array}}$$

where the derivation $\mathcal{D}'$ is obtained from $\mathcal{D}$ by erasing the resource $ru : D_2$ from both input and output contexts of all sequents in it.

In contrast, when the leftover resource $ru : D_2$ is used up in the right branch, the def-form can be changed to a let-form as captured by the rewrite rule $\underline{\text{sconjL01sconjRB}}$, defined as

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I, lu : D_1 \setminus \Delta' \longrightarrow M_1 : G_1} \qquad lu \notin \Delta'}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta', ru : D_2 \longrightarrow \text{def}_1\ lu{\otimes}ru = u \text{ in } M_1 : G_1}\ \textbf{il-}{\otimes}\text{L}_{01} \quad \overset{\mathcal{D}}{\Gamma; \Delta', ru : D_2 \setminus \Delta_O \longrightarrow M_2 : G_2}}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta_O \longrightarrow (\text{def}_1\ lu{\otimes}ru = u \text{ in } M_1) \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}{\otimes} \\[2ex]
\downarrow\ \begin{array}{l} \{ru : D_2 \notin \Delta_O\} \\ \underline{\text{sconjL01sconjRB}} \end{array} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}'}{\Gamma; \Delta_I, lu : D_1, ru : D_2 \setminus \Delta', ru : D_2 \longrightarrow M_1 : G_1} \quad \overset{\mathcal{D}}{\Gamma; \Delta', ru : D_2 \setminus \Delta_O \longrightarrow M_2 : G_2}}{\Gamma; \Delta_I, lu : D_1, ru : D_2 \setminus \Delta_O \longrightarrow M_1 \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}{\otimes} \qquad lu, ru \notin \Delta_O}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } (M_1 \otimes M_2)) : G_1 \otimes G_2}\ \textbf{il-}{\otimes}\text{L}_{00}
\end{array}}$$

where the proof $\mathcal{E}'$ is obtained from $\mathcal{E}$ by adding the resource $ru : D_2$ to input and output contexts of all sequents in it.

Case il-$\otimes$L$_{10}$ on the left branch:

$$\boxed{\begin{array}{c}
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I, ru : D_2 \setminus \Delta' \longrightarrow M_1 : G_1} \qquad ru \notin \Delta'}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta', lu : D_1 \longrightarrow \text{def}_2\ lu \otimes ru = u \text{ in } M_1 : G_1}\ \textbf{il-}{\otimes}\text{L}_{10} \quad \overset{\mathcal{D}}{\Gamma; \Delta', lu : D_1 \setminus \Delta_O, lu : D_1 \longrightarrow M_2 : G_2}}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta_O, lu : D_1 \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } M_1) \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}{\otimes} \\[2ex]
\downarrow\ \underline{\text{sconjL10sconjRA}} \\[2ex]
\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta' \longrightarrow M_1 : G_1} \quad \overset{\mathcal{D}'}{\Gamma; \Delta' \setminus \Delta_O \longrightarrow M_2 : G_2}}{\Gamma; (\Delta_I, ru : D_2) \setminus \Delta_O \longrightarrow M_1 \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}{\otimes} \qquad ru \notin \Delta_O}{\Gamma; (\Delta_I, u : D_1 \otimes D_2) \setminus \Delta_O, lu : D_1 \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } (M_1 \otimes M_2)) : G_1 \otimes G_2}\ \textbf{il-}{\otimes}\text{L}_{01}
\end{array}}$$

where the derivation $\mathcal{D}'$ is obtained from $\mathcal{D}$ by erasing the resource $lu : D_1$ from both input and output contexts of all sequents in it.

In contrast, when the leftover resource $lu : D_1$ is used up in the right branch, the def-form can be changed to a let-form as captured by the rewrite rule <u>sconjL10sconjRB</u>, defined as

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I, ru : D_2 \setminus \Delta' \longrightarrow M_1 : G_1} \qquad ru \notin \Delta'}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta', lu : D_1 \longrightarrow \text{def}_1\ lu \otimes ru = u \text{ in } M_1 : G_1}\ \textbf{il-}\otimes\text{L}_{01} \qquad \overset{\mathcal{D}}{\Gamma; \Delta', lu : D_1 \setminus \Delta_O \longrightarrow M_2 : G_2}}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta_O \longrightarrow (\text{def}_2\ lu \otimes ru = u \text{ in } M_1) \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}\otimes$$

$$\downarrow \quad \begin{array}{l} \{lu : D_1 \notin \Delta_O\} \\ \underline{\text{sconjL10sconjRB}} \end{array}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}'}{\Gamma; \Delta_I, lu : D_1, ru : D_2 \setminus \Delta', lu : D_1 \longrightarrow M_1 : G_1} \quad \overset{\mathcal{D}}{\Gamma; \Delta', lu : D_1 \setminus \Delta_O \longrightarrow M_2 : G_2}}{\Gamma; \Delta_I, lu : D_1, ru : D_2 \setminus \Delta_O \longrightarrow M_1 \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}\otimes \qquad lu, ru \notin \Delta_O}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } (M_1 \otimes M_2)) : G_1 \otimes G_2}\ \textbf{il-}\otimes\text{L}_{00}$$

Case il-$\otimes$L$_{00}$ on the left branch:

Finally, if the rule **il-**$\otimes$L$_{00}$ precedes an instance of **il-**$\otimes$, the associated rewrite rule <u>sconjL00sconjR</u> is straightforward and is defined as

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I, lu : D_1, ru : D_2 \setminus \Delta' \longrightarrow M_1 : G_1} \qquad lu, ru \notin \Delta'}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta' \longrightarrow (\text{let } lu \otimes ru = u \text{ in } M_1) : G_1}\ \textbf{il-}\otimes\text{L}_{00} \qquad \overset{\mathcal{D}}{\Gamma; \Delta' \setminus \Delta_O \longrightarrow M_2 : G_2}}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } M_1) \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}\otimes$$

$$\downarrow \ \underline{\text{sconjL00sconjR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma; \Delta_I, lu : D_1, ru : D_2 \setminus \Delta' \longrightarrow M_1 : G_1} \quad \overset{\mathcal{D}}{\Gamma; \Delta' \setminus \Delta_O \longrightarrow M_2 : G_2}}{\Gamma; \Delta_I, lu : D_1, ru : D_2 \setminus \Delta_O \longrightarrow M_1 \otimes M_2 : G_1 \otimes G_2}\ \textbf{il-}\otimes \qquad lu, ru \notin \Delta_O}{\Gamma; \Delta_I, u : D_1 \otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } lu \otimes ru = u \text{ in } (M_1 \otimes M_2)) : G_1 \otimes G_2}\ \textbf{il-}\otimes\text{L}_{00}$$

Case il-$\multimap$L on the left branch:

This is when a **il-**$\otimes$ is preceded by a **il-** $\multimap$ L on its right branch. This situation occurs frequently when the leftover resource is generated by a sequence of left implication rules.

$$\dfrac{\dfrac{\overset{\mathcal{E}_1}{\Gamma;\Delta_I\setminus\Delta'\longrightarrow N:G}\quad \overset{\mathcal{E}_2}{\Gamma;(\Delta',iu:D)\setminus\Delta'_M\longrightarrow M_1:G_1}\qquad iu\notin\Delta''}{\Gamma;(\Delta_I,u:G\multimap D)\setminus\Delta_M\longrightarrow(\text{let }iu=u\hat{\ }N\text{ in }M_1):G_1}\;\textbf{il-}\multimap\text{L}\qquad \overset{\mathcal{D}}{\Gamma;\Delta_M\setminus\Delta_O\longrightarrow M_2:G_2}}{\Gamma;(\Delta_I,u:G\multimap D)\setminus\Delta_O\longrightarrow(\text{let }iu=u\hat{\ }N\text{ in }M_1)\otimes M_2:G_1\otimes G_2}\;\textbf{il-}\otimes$$

$$\downarrow\ \underline{\text{limpLsconjRA}}$$

$$\dfrac{\overset{\mathcal{E}_1}{\Gamma;\Delta_I\setminus\Delta'\longrightarrow N:G}\qquad \dfrac{\overset{\mathcal{E}_2}{\Gamma;(\Delta',iu:D)\setminus\Delta'_M\longrightarrow M_1:G_1}\quad \overset{\mathcal{D}'}{\Gamma;\Delta'_M\setminus\Delta'_O\longrightarrow M_2:G_2}}{\Gamma;(\Delta',iu:D)\setminus\Delta_O\longrightarrow(M_1\otimes M_2):G_1\otimes G_2}\;\textbf{il-}\otimes}{\Gamma;(\Delta_I,u:G\multimap D)\setminus\Delta_O\longrightarrow(\text{let }iu=u\hat{\ }N\text{ in }(M_1\otimes M_2)):G_1\otimes G_2}\;\textbf{il-}\multimap\text{L}$$

where the derivation $\mathcal{D}'$ is obtained from the derivation $\mathcal{D}$ by removing the additional dependency labels placed by the side condition $\Delta_M = \text{UpdDep}(iu, \Delta' - \Delta_I, \Delta'_M)$ associated with the **il-** $\multimap$ L rule from affected resources in all of its input and output contexts.

Case il-copy on the left branch:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma,u:D);(\Delta_I,v:D)\setminus\Delta'\longrightarrow M_1:G_1}\qquad v\notin\Delta'}{(\Gamma,u:D);\Delta_I\setminus\Delta'\longrightarrow(\text{let }v=u\text{ in }M_1):G_1}\;\textbf{il-}\mathit{copy}\qquad \overset{\mathcal{D}}{(\Gamma,u:D);\Delta'\setminus\Delta_O\longrightarrow M_2:G_2}}{(\Gamma,u:D);\Delta_I\setminus\Delta_O\longrightarrow(\text{let }v=u\text{ in }M_1)\otimes M_2:G_1\otimes G_2}\;\textbf{il-}\otimes$$

$$\downarrow\ \underline{\text{copysconjR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{(\Gamma,u:D);(\Delta_I,v:D)\setminus\Delta'\longrightarrow M_1:G_1}\quad \overset{\mathcal{D}}{(\Gamma,u:D);\Delta'\setminus\Delta_O\longrightarrow M_2:G_2}}{(\Gamma,u:D);(\Delta_I,v:D)\setminus\Delta_O\longrightarrow M_1\otimes M_2:G_1\otimes G_2}\;\textbf{il-}\otimes\qquad v\notin\Delta_O}{(\Gamma,u:D);\Delta_I\setminus\Delta_O\longrightarrow(\text{let }v=u\text{ in }(M_1\otimes M_2)):G_1\otimes G_2}\;\textbf{il-}\mathit{copy}$$

Case il-∀L on the left branch:

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma;(\Delta_I,au:([t/x]D))\setminus\Delta'\longrightarrow M:G}}{\Gamma;(\Delta_I,u:(\forall x.\ D))\setminus\Delta'\longrightarrow(\text{let }au=u\diamond^{\forall}t\text{ in }M):G}\;\textbf{il-}\forall\text{L}\qquad \overset{\mathcal{D}}{\Gamma;\Delta'\setminus\Delta_O\longrightarrow M_2:G_2}}{\Gamma;(\Delta_I,u:(\forall x.\ D))\setminus\Delta_O\longrightarrow(\text{let }au=u\diamond^{\forall}t\text{ in }M_1)\otimes M_2:G_1\otimes G_2}\;\textbf{il-}\otimes$$

$$\downarrow\ \underline{\text{forallLsconjR}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma;(\Delta_I,au:([t/x]D))\setminus\Delta'\longrightarrow M_1:G_1}\quad \overset{\mathcal{D}}{\Gamma;\Delta'\setminus\Delta_O\longrightarrow M_2:G_2}}{\Gamma;(\Delta_I,au:([t/x]D))\setminus\Delta_O\longrightarrow M_1\otimes M_2:G_1\otimes G_2}\;\textbf{il-}\otimes}{\Gamma;(\Delta_I,u:(\forall x.\ D))\setminus\Delta_O\longrightarrow(\text{let }au=u\diamond^{\forall}t\text{ in }(M_1\otimes M_2)):G_1\otimes G_2}\;\textbf{il-}\forall\text{L}$$

Cases il- $\multimap$ **L, il-copy, il-∀L, il-**$\otimes$**L**$_{00}$**, il-**$\otimes$**L**$_{01}$ **and il-**$\otimes$**L**$_{10}$ **on the right**

branch:

These cases are symmetric to given cases on the left branch.

4.3.5.6 Proofs ending with il- $\multimap$ L

Similar to earlier examples, possible cases are categorized by the preceding rule and whether the preceding rule is in the subgoal (left) branch or the decompositional (right) branch. There are also different possibilities based on possible different dependencies among resources. Note that we will never have to shift right rules down in the derivation for the soundness proof, so we only consider cases where the preceding rules are left rules.

Cases il-$\otimes$L$_{00}$, il-$\otimes$L$_{01}$ and il-$\otimes$L$_{10}$ on the subgoal branch:

The rewriting rules in Figure 4.6 illustrate cases where the subgoal branch has either **il-**$\otimes$L$_{01}$ or **il-**$\otimes$L$_{00}$.

The first two rewriting rules correspond to two subcases for **il-**$\otimes$L$_{01}$ depending on whether the partial resource generated by the decomposed resource is utilized in the right branch of the left implication rule or not.

The subcase when the partial resource is left unused, i.e. $ru_2 : D_2 \in \Delta_O$, is captured by the rewrite rule sconjL01limpLGA. Following the transformation, the proof $\mathcal{D}'$ is obtained from $\mathcal{D}$ by removing the resource $ru_2 : D_2$ from all input and output contexts in the derivation, which is possible since $ru_2 : D_2$ also appears in the overall output of the last sequent.

In contrast, when the partial leftover is used in the right branch of the implication with $ru_2 : D_2 \notin \Delta_O$, the def-form has to be transformed into a let-form. This is captured by the rewriting rule sconjL01limpLGA in Figure 4.6. Following the transformation, the derivation $\mathcal{E}'$ is obtained from the derivation $\mathcal{E}$ by recursively adding the resource $ru_2 : D_2$ to all of its input and output contexts, considering global slack counters when initializing the associated slack indicators.

Finally, last rewriting rule sconjL00limpLG in Figure 4.6 shows the subcase where a **il-**$\otimes L_{00}$ precedes a **il-** $\multimap$ L, for which no special considerations are needed.

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma;\Delta_I, lu_2 : D_1 \setminus \Delta' \longrightarrow N_2 : G} \quad lu_2 \notin \Delta'}{\Gamma;\Delta_I, u_2 : D_1\otimes D_2 \setminus \Delta', ru_2 : D_2 \longrightarrow \text{def}_1\ lu_2\otimes ru_2 = u_2 \text{ in } N_2 : G}\ \textbf{il-}\otimes\text{L}_{01} \quad \overset{\mathcal{D}}{\Gamma;\Delta', ru_2 : D_2, iu_1 : D \setminus \Delta'_O \longrightarrow N_1 : C} \quad iu_1 \notin \Delta'_O}{\Gamma;\Delta_I, u_1 : G \multimap D, u_2 : D_1\otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } iu_1 = u_1\hat{}(\text{def}_1\ lu_2\otimes ru_2 = u_2 \text{ in } N_2) \text{ in } N_1) : C}\ \textbf{il-}\multimap\text{L}$$

$$\downarrow \quad \dfrac{\{ru_2 : D_2 \in \Delta_O\}}{\text{sconjL01limpLGA}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma;\Delta_I, lu_2 : D_1 \setminus \Delta' \longrightarrow N_2 : G} \quad \overset{\mathcal{D}'}{\Gamma;\Delta', iu_1 : D \setminus \Delta'_O \longrightarrow N_1 : C} \quad iu_1 \notin \Delta'_O}{\Gamma;\Delta_I, u_1 : G \multimap D, lu_2 : D_1 \setminus \Delta_O - \{ru_2 : D_2\} \longrightarrow (\text{let } iu_1 = u_1\hat{}N_2 \text{ in } N_1) : C}\ \textbf{il-}\multimap\text{L} \quad lu_2 \notin \Delta'_O}{\Gamma;\Delta_I, u_1 : G \multimap D, u_2 : D_1\otimes D_2 \setminus \Delta_O \longrightarrow (\text{def}_1\ lu_2\otimes ru_2 = u_2 \text{ in } (\text{let } iu_1 = u_1\hat{}N_2 \text{ in } N_1)) : C}\ \textbf{il-}\otimes\text{L}_{01}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma;\Delta_I, lu_2 : D_1 \setminus \Delta' \longrightarrow_{i\setminus i'} N_2 : G} \quad lu_2 \notin \Delta'}{\Gamma;\Delta_I, u_2 : D_1\otimes D_2 \setminus \Delta', ru_2 : D_2 \longrightarrow \text{def}_1\ lu_2\otimes ru_2 = u_2 \text{ in } N_2 : G}\ \textbf{il-}\otimes\text{L}_{01} \quad \overset{\mathcal{D}}{\Gamma;\Delta', ru_2 : D_2, iu_1 : D \setminus \Delta'_O \longrightarrow N_1 : C} \quad iu_1 \notin \Delta'_O}{\Gamma;\Delta_I, u_1 : G \multimap D, u_2 : D_1\otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } iu_1 = u_1\hat{}(\text{def}_1\ lu_2\otimes ru_2 = u_2 \text{ in } N_2) \text{ in } N_1) : C}\ \textbf{il-}\multimap\text{L}$$

$$\downarrow \quad \dfrac{\{ru_2 : D_2 \notin \Delta_O\}}{\text{sconjL01limpLGB}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}'}{\Gamma;\Delta_I, lu_2 : D_1, ru_2 : D_2 \setminus \Delta', ru_2 : D_2 \longrightarrow N_2 : G} \quad \overset{\mathcal{D}}{\Gamma;\Delta', ru_2 : D_2, iu_1 : D \setminus \Delta'_O \longrightarrow N_1 : C} \quad iu_1 \notin \Delta_O}{\Gamma;\Delta_I, u_1 : G \multimap D, lu_2 : D_1, ru_2 : D_2 \setminus \Delta_O \longrightarrow (\text{let } iu_1 = u_1\hat{}N_2 \text{ in } N_1) : C}\ \textbf{il-}\multimap\text{L} \quad lu_2, ru_2 \notin \Delta_O}{\Gamma;\Delta_I, u_1 : G \multimap D, u_2 : D_1\otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } lu_2 \otimes ru_2 = u_2 \text{ in } (\text{let } iu_1 = u_1\hat{}N_2 \text{ in } N_1)) : C}\ \textbf{il-}\otimes\text{L}_{00}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma;\Delta_I, lu_2 : D_1, ru_2 : D_2 \setminus \Delta' \longrightarrow N_2 : G} \quad lu_2, ru_2 \notin \Delta'}{\Gamma;\Delta_I, u_2 : D_1\otimes D_2 \setminus \Delta' \longrightarrow \text{let } lu_2 \otimes ru_2 = u_2 \text{ in } N_2 : G}\ \textbf{il-}\otimes\text{L}_{00} \quad \overset{\mathcal{D}}{\Gamma;\Delta', iu_1 : D \setminus \Delta'_O \longrightarrow N_1 : C} \quad iu_1 \notin \Delta_O}{\Gamma;\Delta_I, u_1 : G \multimap D, u_2 : D_1\otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } iu_1 = u_1\hat{}(\text{let } lu_2 \otimes ru_2 = u_2 \text{ in } N_2) \text{ in } N_1) : C}\ \textbf{il-}\multimap\text{L}$$

$$\downarrow \quad \underline{\text{sconjL00limpLG}}$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma;\Delta_I, lu_2 : D_1, ru_2 : D_2 \setminus \Delta' \longrightarrow N_2 : G} \quad \overset{\mathcal{D}}{\Gamma;\Delta', iu_1 : D \setminus \Delta'_O \longrightarrow_{i'\setminus j} N_1 : C} \quad iu_1 \notin \Delta_O}{\Gamma;\Delta_I, u_1 : G \multimap D, lu_2 : D_1, ru_2 : D_2 \setminus \Delta_O \longrightarrow (\text{let } iu_1 = u_1\hat{}N_2 \text{ in } N_1) : C}\ \textbf{il-}\multimap\text{L} \quad lu_2, ru_2 \notin \Delta'_O}{\Gamma;\Delta_I, u_1 : G \multimap D, u_2 : D_1\otimes D_2 \setminus \Delta_O \longrightarrow (\text{let } lu_2 \otimes ru_2 = u_2 \text{ in } (\text{let } iu_1 = u_1\hat{}N_2 \text{ in } N_1)) : C}\ \textbf{il-}\otimes\text{L}_{00}$$

Figure 4.6: Rewriting rules for proofs ending with **il-** $\multimap$ L, preceded by **il-**$\otimes$L$_{01}$ or **il-**$\otimes$L$_{00}$ on the subgoal (left) branch.

Chapter 5

LinGraph: A Graph Based Linear Logic Theorem Prover for Task Planning

In this chapter, we propose our second main contribution, a novel framework for automated generation of plans for STRIPS-based planning problems based on a graph-based theorem prover for a propositional fragment of intuitionistic linear logic [10, 11]. In doing so, we rely on the previously established connection between intuitionistic linear logic and planning problems, which allows us to reduce the complexity of plan construction through proof search while preserving correspondence to logically sound semantics for planning. LinGraph is particularly successful for dealing with concurrent domains having large numbers of resources and agents with identical properties (e.g. robots within a swarm, or parts in a large factory). We first present our graph-based automated planner which we call the Linear Logic Graph Planner (*LinGraph*). Subsequently we illustrate its application for planning the actions of robots within a concurrent manufacturing domain and provide comparisons with four existing automated planners, BlackBox, Symba-2, Metis and the Temporal Fast Downward (TFD), covering a wide range of state-of-the-art automated planning techniques and implementations that are well-known in the literature for their performance on various of problem types and domains. We show that even though LinGraph does not rely on any heuristics, it still outperforms these systems for concurrent domains with large numbers of identical objects and agents, finding feasible plans that they cannot identify. These gains persist even when existing methods on symmetry reduction and numerical fluents are used, with LinGraph capable of handling problems with thousands of objects. Following these results, we also formally show that plan construction with LinGraph is equivalent to multiset rewriting systems, establishing a formal relation between LinGraph and intuitionistic linear logic.

5.1 Linear Logic Graph Planner: The Language

There are currently no automated planner implementations that exploit the connection between intuitionistic linear logic and planning problems that has previously been observed by a number of researchers [12, 54, 58, 82]. In order to present novel contributions of LinGraph in this context, this section introduces the Linear Graph Planning Logic (LGPL) language, the multiplicative exponential fragment of intuitionistic propositional linear logic we use as a basis for LinGraph, mirroring previous uses of linear logic for planning.

We define the LGPL language by the following grammar:

$$\begin{array}{lll}
\text{Resource formulas:} & \mathcal{F}_R ::= & p \mid (p \otimes \mathcal{F}_R) \\
\text{Action formulas:} & \mathcal{F}_A ::= & (\mathcal{F}_R \multimap \mathcal{F}_R) \\
\text{Program formulas:} & \mathcal{F}_P ::= & !\mathcal{F}_A \mid \mathcal{F}_R \\
\text{Goal formulas:} & \mathcal{F}_G ::= & \mathcal{F}_R \mid (\mathcal{F}_P \multimap \mathcal{F}_G)
\end{array}$$

where p denotes atomic resources, $\mathcal{F}_R$ denotes linear formulae that represent resources , $\mathcal{F}_A$ denotes implicative formulae representing actions, $\mathcal{F}_P$ denotes program formulae that can appear as assumptions and $\mathcal{F}_G$ denotes goal formulae.

In the introduced grammar, we have two multiplicative linear connectives, *simultaneous conjunction* $\otimes$ and *linear implication* $\multimap$, that are enough to model dynamic components of the state. Incorporating other linear logic connectives such as disjunction and alternative conjunction can increase expressivity. However, we don't need those connectives to express STRIPS problems. Thereby, for simplicity and efficiency, we focus on this small fragment of linear logic to develop our graph based theorem prover and planner. We only have the modal use of the *bang* operator, !, which is an additional component in LGPL other than the multiplicative connectives. This operator makes it possible to use action definitions more than once, since we shouldn't interpret actions as consumable resources or dynamic state.

In the rest of this section, we will cover proof theoretic semantics for the $LGPL$ language. For this purpose, we will use a sequent calculus formulation which is very similar to the given sequent calculus in Section 2.2.2. Below, we present the corresponding sequent definition

$$\Gamma; \Delta \Rightarrow \mathcal{F}_G \ , \tag{5.1}$$

where Δ is a multiset of program formulae and Γ is a multiset of action formulae. We can interpret this sequent as, we can prove the goal formula $\mathcal{F}_G$ using the single-use resources in Δ and unrestricted propositions (actions) in Γ. We will use this sequent definition to formalize proof construction by introducing *left* and *right* inference rules. Based on the given sequent definition, we give right and left sequent calculus rules for $LGPL$ in Figure 5.26. We should note that, we follow the conventions of [81].

The *init* rule appears at the leaves of the proof tree. It basically associates atomic resources with atomic goals. In the sequent, only a single atomic resource can be on the left hand side, forcing every such resource to be used at least once. On the other hand, the *copy* rule places no restrictions on the use of propositions in Γ, ensuring unrestricted uses for domain actions.

Using the $\otimes R$ rule, we try to achieve a conjunctive goal by splitting the given resources into two in the sequent. This process eliminates the possibility of reusing

Hypotheses

$$\frac{}{\Gamma; p \Rightarrow p}\ init \qquad \frac{(\Gamma, \mathcal{F}_A); (\Delta, \mathcal{F}_A) \Rightarrow \mathcal{F}_G}{(\Gamma, \mathcal{F}_A); \Delta \Rightarrow \mathcal{F}_G}\ copy$$

Simultaneous Conjunction

$$\frac{\Gamma; \Delta_1 \Rightarrow p \quad \Gamma; \Delta_2 \Rightarrow \mathcal{F}_R}{\Gamma; \Delta_1, \Delta_2 \Rightarrow p \otimes \mathcal{F}_R} \otimes R \qquad \frac{\Gamma; \Delta, p, \mathcal{F}_R \Rightarrow \mathcal{F}_G}{\Gamma; \Delta, p \otimes \mathcal{F}_R \Rightarrow \mathcal{F}_G} \otimes L$$

Linear Implication

$$\frac{\Gamma; \Delta, \mathcal{F}_P \Rightarrow \mathcal{F}_G}{\Gamma; \Delta \Rightarrow \mathcal{F}_P \multimap \mathcal{F}_G} \multimap R \qquad \frac{\Gamma; \Delta_1 \Rightarrow \mathcal{F}_{R1} \quad \Gamma; \Delta_2, \mathcal{F}_{R2} \Rightarrow \mathcal{F}_G}{\Gamma; \Delta_1, \Delta_2, \mathcal{F}_{R1} \multimap \mathcal{F}_{R2} \Rightarrow \mathcal{F}_G} \multimap L$$

Unrestricted Resources

$$\frac{(\Gamma, \mathcal{F}_A); \Delta \Rightarrow \mathcal{F}_G}{\Gamma; (\Delta, !\mathcal{F}_A) \Rightarrow \mathcal{F}_P}\ !L$$

Figure 5.1: Sequent calculus proof rules for $LGPL$.

an existing resource more than once. The $\otimes L$ rule decomposes a conjunctive resource and then adds each decomposed resource to the context of consumable resources. The $\multimap R$ rule achieves the conclusion $\mathcal{F}_G$ by adding the program formula $\mathcal{F}_P$ into the available resources Δ. The $\multimap L$ rule first achieves the formula $\mathcal{F}_{R1}$, and then achieves the goal $\mathcal{F}_G$ using the resource $\mathcal{F}_{R2}$ while splitting available resources as necessary. For unrestricted resources, we introduce the $!L$ rule and the *copy* rule, recovering the possibility of using action definitions more than once throughout the proof.

These presented inference rules ensure that all resources must be used exactly once in the consumable resources context Δ. Consumable resource concept together with an *intuitionistic* fragment of linear logic in $LGPL$ provide associating each proof for goal formulae with a valid plan for the associated planning problem. However this association is not our main contribution. Instead of this association, the main contribution is the novel, graph-based theorem prover and planner associated with this language and proof theory.

We use simultaneous conjunction ($\otimes$) to group components of system state at a particular time instant, while we use linear implication ($\multimap$) to model actions in a task planning domain. In LGPL, we also use linear implication ($\multimap$) within goal formulae to define initial states and actions for a domain using currying. Given the goal formula

$$\mathcal{F}_{P1} \multimap (... \multimap (\mathcal{F}_{Pn} \multimap \mathcal{F}_G)...), \tag{5.2}$$

considering the program formulae among $\mathcal{F}_{P1}$ through $\mathcal{F}_{Pn}$, resource formulae define the initial system state while implicative formulae define available actions. This structure closely corresponds to the STRIPS formalism while maintaining the logical semantics of linear logic.

Our introduced theorem prover LinGraph has a connection to Girard's proof nets [83]. Proof nets are geometric representations of proofs. The geometric representation of proofs eliminates different orderings of resources and applies rules in a derivation that only differentiates distinct proofs. Some of our ideas in the LinGraph method are similar, for example, in our group of multiple instances of identical objects within the same graph node.

5.2 Assembly Planning Domain Example

5.2.1 Domain Definition

In this section, we introduce a new class of planning domains, enabling us to provide systematic comparisons of performance between LinGraph and other, state-of-the-art planners. In this domain, there are multiple types of *components* C_i, *manipulators* M_i and *products* P_i. All instances having the same type of object have the same functionality and properties, although they are distinct physical objects. In this context, manipulators are assumed to operate on components to generate products as a result. We give an overview of objects in Figure 5.2.

For instance, a simple example of such a domain can include a single component type C, a single manipulator type M and a single product type P, with an action

Objects:		
$C_1, ..., C_n$	:	Components of type 1 through n
$M_1, ..., M_m$	:	Manipulators of type 1 through m
$P_1, ..., P_o$	:	Products of type 1 through o

Figure 5.2: Objects types and notation for the robotic assembly planning domain.

MakeP transforming a component C into a product P. A more complex example might have different types of components, manipulators and products. And each manipulator might have multiple different actions.

In this domain, we can present a large scenario using these building blocks. Assuming a large factory, there might be hundreds of manipulator robots considered as resources and they are capable of operating in parallel. Using this scenario for our example domain, we can illustrate one of the distinguishing advantages of the LinGraph planner.

5.2.2 Encoding the Assembly Planning Domain in LGPL

We described in Section 5.1 that we can encode STRIPS style domains in LGPL by using linear resources to represent dynamic state components and implicative formulas to represent actions [54]. We now present a particular problem in our assembly planning domain, having a single type of manipulator, M, that takes a component of type C and transforms it into a product of type P. We encode this action in LGPL as below

$$\text{MakeP} := !(C \otimes M \multimap M \otimes P) \, , \tag{5.3}$$

where MakeP is the name of the action. Using the preconditions of the action, a component C and a manipulator M, resulting in the production of a product P and keeping the manipulator M available for later use. We should also note that, in LGPL if a resource is required but is left unaffected by the action, then we need to reintroduce the resource in the effect part of the action such as the manipulator M resource in the example.

We now encode the initial state of a problem example with three components and two manipulators within LGPL, such as

$$\mathcal{F}_{R0} \ := \ (C \otimes C \otimes C \otimes M \otimes M). \tag{5.4}$$

In this definition, there are repeated resources such as three C and two M. We abbreviate such repeated occurrences of resources with an *exponent* notation, defined as

$$(F_R)^n := \underbrace{(F_R \otimes ... \otimes F_R)}_{n \text{ times}} \ . \tag{5.5}$$

We can encode the given initial state using this notation as $\mathcal{F}_{R0} := (C^3 \otimes M^2)$.

Next, we need a final state for a complete problem specification. For example, if we want to produce three components, we can encode the final state as

$$\mathcal{F}_{Gf} := (P^3 \otimes M^2) \tag{5.6}$$

Finally, we combine the initial state and the final state for this simple example to represent the planning problem in LGPL such as

$$\begin{aligned} \mathcal{F}_G \ &:= \ \text{MakeP} \multimap \mathcal{F}_{R0} \multimap \mathcal{F}_{Gf} && (5.7) \\ &= \ !(C \otimes M \multimap P \otimes M) \multimap (C^3 \otimes M^2) \multimap (P^3 \otimes M^2) && (5.8) \end{aligned}$$

Every proof of this formula with the inference rules in Figure 5.26 corresponds to a valid plan. Each plan is a partial ordering of actions, including multiple applications of the action MakeP. We should also note that, multiple proofs might correspond to the same plan.

We can extend the example given above such that it includes a thousand of components and a thousand of manipulators to produce a thousand of products. We can encode the goal formula for this example as

$$!(C \otimes M \multimap P \otimes M) \multimap (C^{1000} \otimes M^{1000}) \multimap (P^{1000} \otimes M^{1000}) \ . \tag{5.9}$$

The solution to this problem is obvious. However, it might be infeasible to find a solution for a linear logic theorem prover, if the prover is not explicitly aware of repeated occurrences of large numbers functionally equivalent resources. For the LinGraph method, this problem is easy to solve by considering functionally equivalent groups of resources as a unit while searching for a valid plan and proof.

5.3 Linear Logic Graph Planner: Plan Construction

In this section, we present the plan construction of *LinGraph*. Before proceeding with the details of plan construction, we review the GraphPlan method which motivated our LinGraph method.

5.3.1 GraphPlan

In [29], the authors introduced GraphPlan as a less expressive but much faster alternative to Partial-Order Planners. GraphPlan is based on a STRIPS representation of states and actions, but only using propositional states. During the encoding of an action in GraphPlan, we encode preconditions as conjunctions of positive propositions, while we encode effects as conjunctions of positive and negative propositions corresponding to the creation and deletion of state components, respectively.

The GraphPlan method builds the reachability graph, having alternating *levels* of propositional states and actions. The first level of the graph includes the initial state of the planning problem. After this level, there are two different steps *graph extension* and *plan search*, that GraphPlan alternates between them. The first step, graph extension, increases the current graph by adding action and state layers. This extension is basically done by matching preconditions of given actions to nodes in the last level and then generating new nodes for effects of related actions. In the plan search step, we try to find a solution using the backchaining method, searching from the last level to the first level. This process alternates between two steps until we find a solution or we reach a preset limit of graph levels.

In a GraphPlan level, if there are conflicts between pairs of actions or pairs of states, then we use binary mutual exclusion (*mutex*) relations. We now give two conditions that are causing mutex between two actions.

1. *Interference*: In the same level, an action deletes a precondition or effect of another action.
2. *Competing Needs*: In the preceding propositional state layer, the preconditions of an action are mutually exclusive with the preconditions of another action.

If all created actions by two propositions are mutually excluded, then there must be a mutual exclusion constraint between those propositions.

After each graph extension step, we check if goal propositions match the propositions at the last level of the graph. Once this condition satisfies, we perform the backchaining search from the last level to the first level, checking if there is a combination of actions and states which is not limited by mutual exclusion constraints. However, if there is not an appropriate combination, this search will fail. Subsequently, we need to expand the graph and continue the plan search. If there is a successful plan, we know that the backchaining search yields a step optimal partial-order plan.

We can say that LinGraph is similar to GraphPlan in its incremental construction of an approximate reachability graph consisting of alternating state and action layers. However, the main difference between these two methods is that LinGraph encodes *linear resources* of the same type together with their multiplicities. This causes the need for *integer inequality constraints* rather than binary mutexes to encode mutual exclusion conditions between nodes.

Another important property of LinGraph is that once we find an appropriate combination of nodes satisfying the goal state at the last level after a forward search, we know that we will definitely find a step optimal plan after plan extraction and we don't need to perform an additional backchaining search. On the other hand, we can say that GraphPlan performs a combination of forward-chaining and backward-chaining in its plan search.

5.3.2 Planning with LinGraph

In this section, we describe the construction and plan extraction algorithm of LinGraph. We use the example domain described in Section 5.2.2 to illustrate all steps of the algorithm. In Figure 5.3, we present the overall structure of LinGraph. The algorithm starts by decomposing the given formula into its linear assumptions which are actually forming the initial state, unrestricted assumptions which are forming action definitions and the goal formula which is forming the goal state. After initializing the first layer and the goal nodes, the goal check performs attempting to match elements of the goal state to elements of the rightmost state layer. Once a successful match is found, LinGraph guarantees exiting of a step optimal plan at the last level, which is a result of the design of node constraints. If there is not a successful match, then we extend the graph by one level (two layers, one for applicable actions and the next for resulting state nodes).

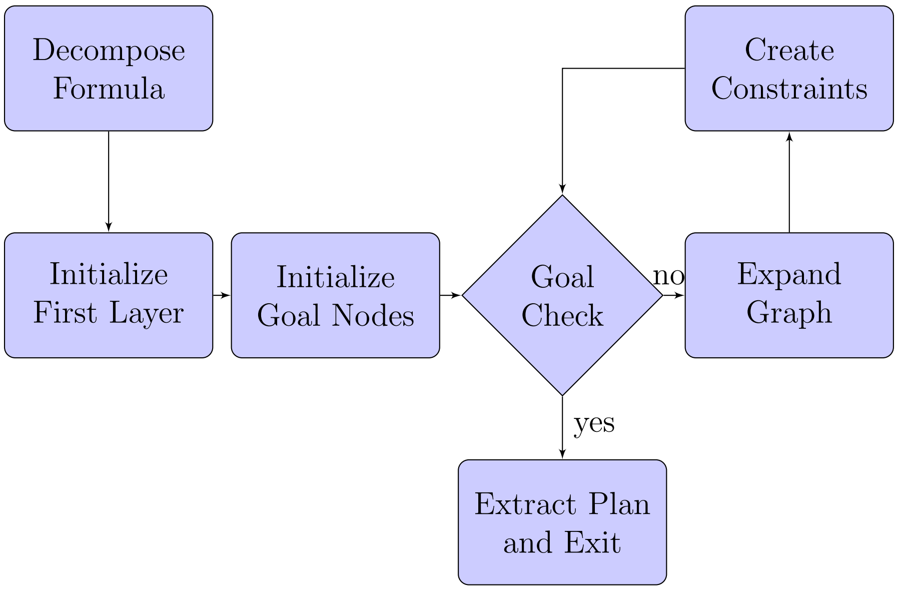

Figure 5.3: An overview of the algorithm for plan search through incremental construction of a LinGraph instance.

We use the components illustrated in Figure 5.4 for visualizing the construction steps of the LinGraph. We use rounded rectangles to show nodes for state layers. p is type of the atomic resource, n is the count that showing how many resources of this type are available and s is a unique label for this node. The second

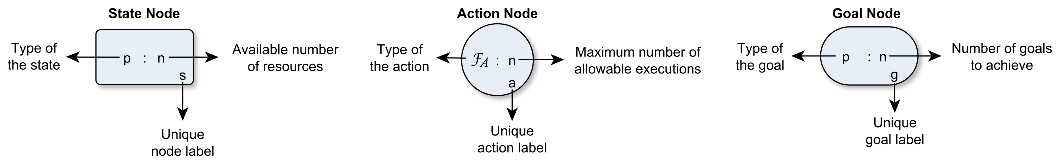

Figure 5.4: Visualization of LinGraph nodes for states, actions and goals, distinguished by their shapes. Each node shows its corresponding LGPL formula, the number of its instances n and its unique label.

component is a circle for nodes in action layers. $\mathcal{F}_A$ is the implicative formula for the action, n is the maximum number of allowable executions for this action and a is a unique action label. The last component is an oval shape for goal nodes. p is showing type of the goal, n is the number of goals to achieve and g is a unique goal label.

We now define a number of notations to use in the rest of this chapter. $n(s_i)$, $n(a_i)$ and $n(g_i)$ denote the number of available instances for state, action and goal nodes, respectively. $lvl(s_i)$ and $lvl(a_i)$ denote the graph level for state and action nodes, respectively. Finally, $frm(s_i)$, $frm(a_i)$ and $frm(g_i)$ denote the LGPL formula associated with state, action and goal nodes, respectively.

5.3.2.1 LinGraph Constraints

In LinGraph, we create *constraints* on the number of instances that can be used for each state node, generalizing the mutual exclusion concept of GraphPlan. Constraints guarantee that extracted plans from LinGraph correspond to associated LGPL proofs. We should also note that constraints take the form of integer linear equality and inequality conditions on the number of instances consumed from each state node, which we denote by overloading the unique labels s_i for state nodes, constrained to be integers in the range $0 \leq s_i \leq n(s_i)$. We formally define constraints in LinGraph as

$$C_j := \sum_{i=0}^{N} \alpha_j^i s_i = \beta_j \quad \text{or} \quad C_j := \sum_{i=0}^{N} \alpha_j^i s_i \geq \gamma_j$$

where N is the total number of state nodes, α_j^i are integer constraint coefficients and β_j, γ_j are constants. In LinGraph, we use constraints for finding a corresponding valid LGPL proof without violating linearity properties of the underlying logic, if there is a match between the goal nodes and the last level state nodes.

Another purpose of using LinGraph constraints is reducing the size of the LinGraph by eliminating unnecessary creation of new actions and nodes during graph expansion step.

To find a feasible solution for a given set of constraint equations, we use an existing constraint solver, *Minion*, [84]. This constraint solver returns the number of instances s_i consumed for all state nodes in LinGraph.

5.3.2.2 Decomposing the Goal Formula and Initializing LinGraph

We now give a simple planning problem to show decomposing the goal formula and initializing LinGraph. Assuming the LGPL formula

$$C^2 \multimap M^2 \multimap !(C \otimes M \multimap M \otimes P) \multimap (P^2 \otimes M^2) \; , \tag{5.10}$$

we will have linear resources $\Delta = \{C^2, M^2\}$, unrestricted resources $\Gamma = \{C \otimes M \multimap M \otimes P\}$ and the goal formula $\mathcal{F}_G = P^2 \otimes M^2$, after parsing this formula using the LGPL grammar in Section 5.1. Subsequently, we will have the LGPL sequent

$$(C \otimes M \multimap M \otimes P); (C^2, M^2) \Rightarrow P^2 \otimes M^2 \; .$$

Afterwards, we decompose the goal formula into its atomic resources as P^2 and M^2 to create goal nodes.

After the decomposition step for the given formula, we initialize first level initial nodes and goal nodes. We create individual state nodes s_i in the first level corresponding to each unique type of initial resource, with $n(s_i)$ initialized with the total number of such resources. For the given example formula, initializing the first level creates two nodes, for type C and for type M, as shown in Figure 5.5.

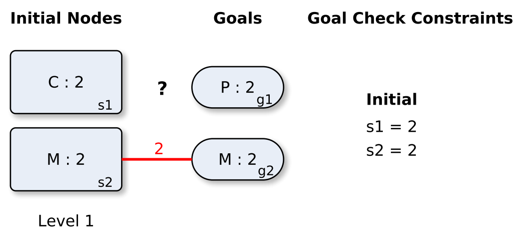

Figure 5.5: Visualization of the LinGraph following the creation of nodes for the initial state and the goal for the planning problem of (5.10). This figure also illustrates initial constraints and the first goal check attempt, which fails since no state nodes match the goal node P.

Similarly, there are two goal nodes for each type of goal resource, one for type P and another for type M. As shown in the figure, each node has a unique identifying label upon creation.

Finally, we complete the initialization of the LinGraph by adding an equality constraint for each initial node. These constraints guarantee that all initial resources will be completely consumed. For the given example, we have two constraints, $s_1 = 2$ and $s_2 = 2$.

5.3.2.3 Goal Check

After each expansion step for building the LinGraph, we check if all desired goals are satisfied with the last level resources and also all constraints should be satisfied. We can define a set of state nodes in the last level l_{max} of the LinGraph for each goal node g_i as

$$S[g_i] := \{s_j \mid (lvl(s_j) = l_{max}) \land (frm(s_j) = frm(g_i))\} \ ,$$

where formulae for the goal node and the state node are matching. We define a constraint for each goal. This constraint is in the form of

$$(\sum_{s_j \in S[g_i]} s_j) = n(g_i) \ .$$

Finally, we use a solver to solve these constraints to find a valid solution, such that each consumed instances of s_i for all state nodes are within the bounds

$0 \leq s_i \leq n(s_i)$.

Recalling the given example above, we find a match for g_2 such that $S[g_2] = \{s_2\}$. We can create the initial constraint for this node as $s_2 = n(g_2) = 2$. On the other hand, we can not find any matches for g_1, yielding $S[g_1] = \{\}$ corresponding to an unsatisfiable constraint $0 = n(g_1) = 2$. Thereby, goal check fails after this step, requiring the expansion of the LinGraph (see Section 5.3.2.4), followed by creation of new constraints as described in Section 5.3.2.5.

5.3.2.4 LinGraph Expansion: Creating New Nodes

When the goal check step fails to find a feasible plan, we need to explore more levels to find a plan. To this end, in the LinGraph expansion step, we create a new level by adding new nodes that are reachable by actions consuming resources in the last level of the existing graph. We denote the last level of the graph with k, while we denote the level after expansion with $(k+1)^{st}$.

We copy all nodes in the level k to the level $(k+1)^{st}$, carrying complete information to the next level. Similar to GraphPlan, we achieve this through the use of *copy* actions. Each copy action has a single precondition and a single effect. with the maximum number of allowable executions matching the number of instances for the precondition. We define the parameterized notation

$$\mathrm{Copy}(x) := x \multimap x$$

to denote formulas associated with copy actions in LinGraph illustrations. We should note that, copy actions are not explicitly added in the original problem formula.

For the given example in Figure 5.5, we show the expansion of the LinGraph in Figure 5.6 from the first level to the second level. Using copy actions a_1 and a_2, we first copy s_1 and s_2 nodes from first level to the second level, creating two new state nodes s_3 and s_4, with the same number of maximum usable instances.

After copying all state nodes to the new level, we consider all applicable action

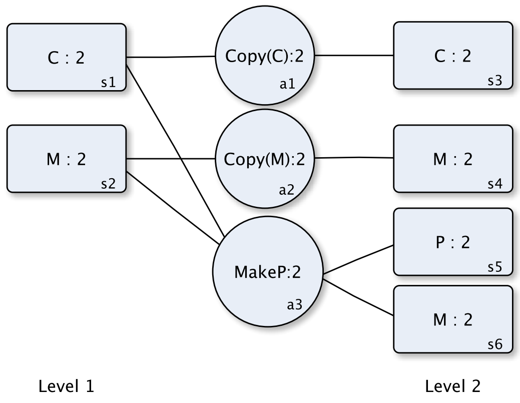

Figure 5.6: LinGraph expansion example illustrating second level nodes created both through copying (s_3 and s_4) from the first level as well as the application of a new instance of the MakeP action (s_5 and s_6).

$F_A \in \Gamma$. An example action has the form

$$F_A := e_1^{c_1} \otimes ... \otimes e_m^{c_m} \multimap f_1^{d_1} \otimes ... \otimes f_n^{d_n}$$

with atomic preconditions and effects. For each such action, LinGraph expansion first attempts to match each precondition e_i to a corresponding state node s_{e_i} in level k, such that $lvl(s_{e_i}) = k$, $frm(s_{e_i}) = e_i$, and $n(s_{e_i}) \geq c_i$. If nodes satisfying these conditions are found for all preconditions, a new action node a_{new} is created with $n(a_{new}) = \min_{i=1}^{m}(\lfloor n(s_{e_i})/c_i \rfloor))$ and $frm(a_{new}) = F_A$. Subsequently, new state nodes are created in level $k+1$ with fresh labels $s_{new,j}$ such that $lvl(s_{new,j}) = k+1$, $lvl(s_{new,j}) = f_i$ and $n(s_{new,j}) = n(a_{new}) * d_j$ with $j = 1, ..., n$.

In Figure 5.6, we create the action MakeP. Since the preconditions of this action match nodes s_1 and s_2 in the first level, we create a new action node a_3 and two state nodes s_5 and s_6 in the second level, each having two available number of resources.

When we finish creating all new actions in the expansion step, the LinGraph proceeds with the creation of dependency constraints for the effects of actions with shared preconditions as described in Section 5.3.2.5.

5.3.2.5 LinGraph Expansion: Sibling and Dependency Constraints

After creating new nodes in the LinGraph expansion step, we create two more types of constraints in addition to the *initial node constraints* of Section 5.3.2.2 and the *goal checking constraints* of Section 5.3.2.3. We use *sibling constraints* to ensure that the effects of a single action are used in accordance with their cardinality in the action definition and *dependency constraints* to enforce limitations arising from shared preconditions between actions within the same level.

We begin with describing sibling constraints. If a created action has two or more different types of effects, then we create a sibling constraint for those effects. Consider, for instance, an action

$$F_A := e_1^{c_1} \otimes ... \otimes e_m^{c_m} \multimap f_1^{d_1} \otimes ... \otimes f_n^{d_n}$$

created during the expansion step from level k to level $k+1$ with $n > 1$. Every instance of this action used within the final plan will create d_1 instances of the resource f_1, d_2 instances of the resource f_2 and so on, all of which must be consumed by subsequent actions. Assuming that state nodes s_1 through s_n are created in level $k+1$ after the expansion step, corresponding to the effects f_1 through f_n for this action, this requires that

$$\frac{s_1}{d_1} = \frac{s_2}{d_2} = ... = \frac{s_n}{d_n} \ .$$

Denoting the least common multiple of d_1 through d_n with $lcm(d_1, ..., d_n)$, this expands into $n-1$ integer equality constraints

$$\begin{aligned} \frac{lcm(d_1,...,d_n)}{d_1} s_1 &= \frac{lcm(d_1,...,d_n)}{d_2} s_2 \\ &... \\ \frac{lcm(d_1,...,d_n)}{d_{n-1}} s_{n-1} &= \frac{lcm(d_1,...,d_n)}{d_n} s_n \end{aligned}$$

added to the current set of LinGraph constraints to be solved during the the goal check. For the example in Figure 5.6, this results in the constraint

$$s_5 = s_6$$

being added to the set of existing constraints as shown in Figure 5.7.

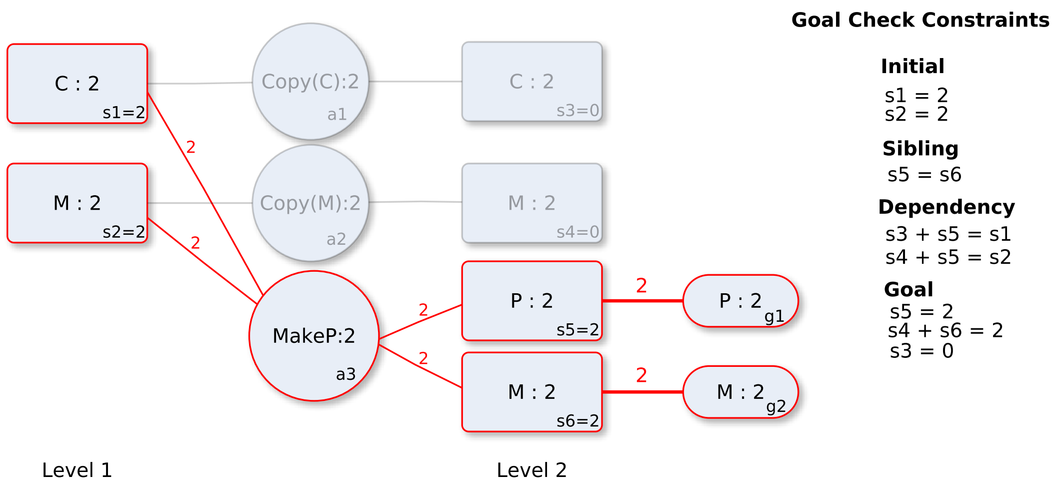

Figure 5.7: Final state of the LinGraph for the planning problem of (5.10), showing all constraints used for the last successful goal check step with all goals satisfied.

In contrast, dependency constraints are introduced following the expansion step from level k to level $k+1$ between effects of actions and their common preconditions in level k. Suppose that a state node s_{cm} with $frm(s_{cm}) = e_{cm}$ in level k appears as a precondition with different cardinalities to multiple actions (possibly including copy actions) $a_1, ..., a_j$, taking the form

$$\begin{aligned} frm(a_1) &= \ldots \otimes e_{cm}^{c_1} \otimes \ldots \multimap f_{1,1}^{d_{1,1}} \otimes \ldots \otimes f_{1,n_1}^{d_{1,n_1}} \\ &\ldots \\ frm(a_j) &= \ldots \otimes e_{cm}^{c_j} \otimes \ldots \multimap f_{j,1}^{d_{j,1}} \otimes \ldots \otimes f_{j,n_j}^{d_{j,n_j}} \end{aligned}$$

The expansion step will then have created new state nodes $s_{r,t}$ in level $k+1$ corresponding to the effects $f_{r,t}$ of these actions with $r = 1, ..., j$ and $t = 1, ..., n_r$. However, there are only $n(s_{cm})$ instances available for the node s_{cm}, meaning that only a certain subset of these actions can be used in the final proof. A constraint must be introduced to ensure that these state nodes in level $k+1$ are used no more or less than what is allowed by the availability of the state node s_{cm} in level k, and that usage counts for successive levels are consistent. Since sibling constraints described above ensure consistency of $s_{r,1}$ through s_{r,n_r} in level $k+1$, it will be sufficient to impose the dependency constraint on a single effect node for each action, which we choose to be their first effects, $s_{r,1}$. Based on these observations, the dependency constraint for s_{cm} takes the form

$$\sum_{r=1}^{j} c_r \frac{s_{r,1}}{d_{r,1}} = s_{cm}$$

which can be transformed into an integer linear equality as

$$\sum_{r=1}^{j} c_r \frac{lcm(d_{1,1},...,d_{j,1})}{d_{r,1}} s_{r,1} = lcm(d_{1,1},...,d_{j,1}) s_{cm}$$

Satisfaction of this constraint ensures that the state node s_{cm} is used within its allowable limits, and no more or less than necessary to support the application of as many instances of the actions a_1 through a_j as necessary for a successful goal check.

In the example of Figure 5.6, the state node s_1 is shared by a_1 and a_3, and the state node s_2 is shared by a_2 and a_3. These dependencies result in the creation of the constraints

$$\begin{aligned} s_3 + s_5 &= s_1, \\ s_4 + s_5 &= s_2 . \end{aligned}$$

as shown in the final LinGraph of Figure 5.7.

Having created both the sibling and dependency constraints for newly created state nodes in level $k+1$, our algorithm proceeds to perform another goal check. To this end, new goal check constraints are created as

$$\begin{aligned} s_5 &= 2, \\ s_4 + s_6 &= 2, \\ s_3 &= 0, \end{aligned}$$

as shown in Figure 5.7, after which the constraint solver is invoked. In this case, a valid solution is found as

$$s_1 = 2,\ s_2 = 2,\ s_3 = 0,\ s_4 = 0,\ s_5 = 2,\ s_6 = 2\ , \tag{5.11}$$

corresponding to a valid solution for the planning problem of (5.10). We can extract the corresponding plan using this solution.

5.3.2.6 LinGraph Expansion: Pruning Unnecessary Actions

One of the main sources of complexity in the LinGraph method is caused by the expansion step while exploring all possible matches for the preconditions of an action to state nodes in the last level. We use two strategies to prevent creating actions that are either redundant, or guaranteed to be inapplicable. Since these strategies decrease the size of the constructed LinGraph, they also increase the performance of plan search.

Suppose that an action

$$F_A := e_1^{c_1} \otimes ... \otimes e_m^{c_m} \multimap f_1^{d_1} \otimes ... \otimes f_n^{d_n}$$

is being considered for expansion and a particular set of matching nodes s_{e_i} have been identified in the level k of the current LinGraph. The first strategy seeks to prevent redundant replication of actions in successive levels. If all preconditions, s_{e_i}, have been created as effects of *copy actions* in level $k-1$, we prevent the creation of a new action since all nodes s_{e_i} have, by construction, identical corresponding nodes in level k, which would have matched the preconditions of F_A, thereby creating its effects. Introducing another action node and its effects would be fully redundant, adding no new plan alternatives.

In the second strategy, we locally check if simultaneous use of all preconditions for a single application of an action is feasible under the current set of constraints. However, such a single, isolated action application will naturally result in a *partial consumption* of available resources. However, to preserve correspondence to the semantics of linear logic, initial, sibling and dependency constraints enforce all available resources to be entirely consumed, and hence do not allow such a partial check.

Fortunately, we can use *inequality constraints* to check for partial resource consumption. Thereby, we replace equality constraints of Section 5.3.2.2 with inequalities, which is setting an upper bound on the number of initial resources to be used. Similarly, we replace equality constraints of dependency constraints into inequality constraints, which is setting a lower bound on resources to be used in level k, based on the needs of level $k+1$. Sibling constraints are kept as equality conditions to ensure that limitations imposed on a node are properly reflected in constraints which might be formulated as a function of its siblings. Finally, usage counts for nodes in level $k+1$ that are neither among the preconditions or for the action under consideration nor among their siblings are forced to be zero for efficiency.

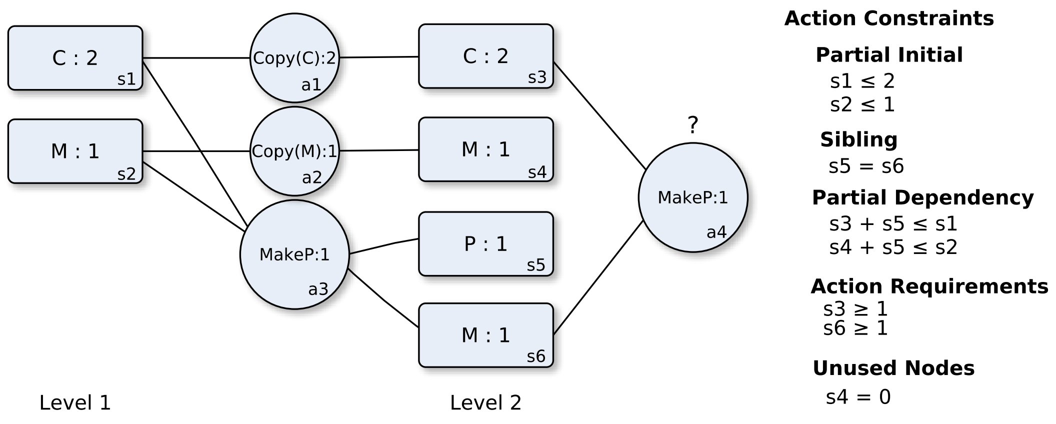

Figure 5.8: LinGraph expansion step checking for feasibility of a new instance of the MakeP action through *action constraints*.

We illustrate this strategy with the example problem

$$C^2 \multimap M \multimap !(C \otimes M \multimap M \otimes P) \multimap (P^2 \otimes M) \ , \tag{5.12}$$

which only has one manipulator. In this example, we can find a plan at third level. In Figure 5.8, we show the picture of LinGraph at the second level, having only a single product available. Thereby, we need to expand the graph one level and consider the MakeP action, yielding the updated action constraints for this problem shown in Figure 5.8. In this case, the resulting set of equality and inequality constraints are satisfiable, resulting in the successful instantiation of the action node.

We should note that the absence of these two strategies for pruning the LinGraph does not impair soundness. We use these strategies for increasing efficiency of plan search.

5.3.2.7 Plan Extraction

If the constraint solver finds a solution during the goal check step, then we finish the LinGraph construction. For the previous example problem, we will extract the final plan from the solution exemplified by (5.11). In Figure 5.9, we show the final graph for the example of (5.12). In this figure, we also show the utilization counts s_i for each node identified by the constraint solver.

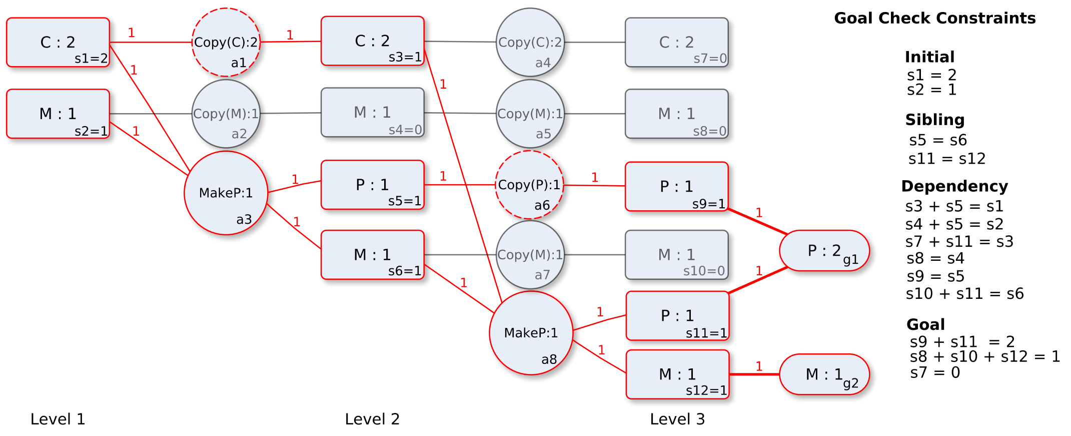

Figure 5.9: Plan extraction using the solution obtained from the constraint solver. Each state node shows the number of utilized resources in the lower right corner. Unused state nodes are faded out for clarity.

During the plan extraction, the LinGraph algorithm generates a multiset of actions to be executed (possibly in parallel) within each level. For the given example, we show the extracted plan in Figure 5.10. This plan includes two successive applications of the MakeP action.

After giving details of the LinGraph algorithm, we make two important observations. The first observation is that we do not include *copy actions* in the final plan, since these type of actions only carry resources from previous level to the next level. If we would include copy actions in the plan, then they would mean

Step 1:	MakeP
Step 2:	MakeP

Figure 5.10: Solution to the modified planning problem of (5.12)

nothing in the planning problem. The second observation is that we can apply multiple instances of an action node within a single level. Subsequent sections will show examples for this feature.

5.3.2.8 Algorithmic Properties of LinGraph

The first component contributing to the complexity of LinGraph comes from the space required to represent its graph structure. Given a LinGraph with i levels, and N_i nodes in its last level, a actions, with a maximum of e postconditions and a maximum of k different ways each action can be applied, the graph expansion phase requires the creation of $N_{i+1} \leq k{*}a{*}e{*}N_i$ new nodes. In the worst case, this means that the LinGraph size is exponential in the number of graph levels, with space complexity $O((kae)^i)$ in a way that is compatible with the EXPSPACE-hard complexity of multiset rewriting [62]. In particularly problematic cases with many nodes having identical propositions, k may also be proportional to N_i, resulting in doubly exponential complexity. In general, however, object symmetry will reduce both e and k since identical propositions will be aggregated in the same node and our constraint based pruning in Section 5.3.2.6 will prevent infeasible actions from being considered.

An important detail to note is that the exponential space complexity of LinGraph is in the number of graph levels. In our examples, we focused on how performance scales with the number of functionally identical object instances, which is the axis along which our algorithm outperforms existing approaches. Naturally, as the number of graph levels (and hence plan length) increases, LinGraph in its current form will become infeasible due to this space complexity. In this direction, two aspects of LinGraph allow delaying the forbidding nature of its exponential space complexity associated with multiset rewriting. First and foremost, LinGraph aggregates identical propositions within the postconditions of an action into a single node, decreasing exponential buildup of space requirements for representing multisets that include identical propositions. Second, domains that encode physical planning problems, in general, include actions that transform physical resources, which inherently disallow an exponential buildup in the number of generated objects and resources. In the long term, the exponential space complexity of LinGraph can be improved on by combining multiple node instances with the same proposition into a single node without, but this requires a careful reconsideration of how node constraints are formulated to maintain semantic validity. We leave this extension for future work.

The second computationally expensive component in LinGraph is the goal check. In the worst case, with g goals, having maximum multiplicities n, each matching a maximum of k, mutually independent nodes in the last level, LinGraph will generate g constraints, each involving at most k variables with their right hand side equal to m. Assuming an uninformed, brute force constraint solver, this result in the consideration of $N_g = (k^m)^g$ combinatorial assignments to constraint variables. Once goal constraints are satisfied, the dependency constraints easily follow since their left hand sides are fixed with initial node constraints determining the feasibility of a particular assignment. It is important to note that the complexity of the goal check is complementary to the complexity of graph construction in that, the worst case for graph size, with each node and goal having a multiplicity of one, corresponds to the best case for the goal check. In contrast, when node multiplicities increase, the graph size decreases but constraints have greater freedom with larger values of g and larger ranges for the individual free variables s_i.

During the goal check, k may also be proportional to the number of nodes in the last LinGraph level in the worst case, meaning that the goal check may also require time exponential in the number of graph levels. Fortunately, most planning problems have considerably more structure in their goals and dependencies within the last level in the associated LinGraph, reducing this complexity in practice through our use of a modern constraint solver and our pruning methods in Section 5.3.2.6. Moreover, the ability of LinGraph to automatically aggregate identical resources allows its average performance to be beyond other currently available methods for domains wherein such functional equivalences are present. The identification of such symmetries is, in general, a nontrivial problem for linear logic and its theorem provers. Proper and automatic handling of these equivalencies is among LinGraph's novel properties. As such, LinGraph's successful elimination of symmetries arising from functionally identical objects does not immediately follow from its foundations in linear logic, but rather as a result of our carefully constructed graph structure and aggregation of identical resources through node multiplicities.

It is important to note that LinGraph does not yet use any heuristics. Therefore, its average performance on sequential problems is expected to be far below existing planners that perform better pruning of the search space. As a result, we have limited the scope of this thesis to present the basic ideas behind LinGraph and its ability to successfully and automatically recognize symmetries in a domain, and left extensions to incorporate heuristics and better search methods to improve its performance on sequential, longer plans for future work. These extensions would require relaxing the layered structure of the graph, tighter integration with the constraint solver to reduce space requirements and work on finding heuristics in the presence of concurrency.

In summary, if a planning problem has a solution, LinGraph will terminate but may require time and space exponential in the number of concurrent plan steps. As we show in our experimental results, the internal structure and symmetries of the planning problem enable LinGraph to successfully identify valid plans, improving on the performance of existing planners on specific domains. Finally, despite the completeness of LinGraph with respect to the LGPL proof theory we show in Section 5.5, in the absence of facilities to detect plan loops, LinGraph may also not be able to successfully conclude the absence of a plan, but this is no worse than the general undecidability of problems with intermediate creation of objects (creation planning) wherein the number of ground symbols may not be bounded.

5.3.3 A More Challenging Assembly Planning Example

We described the assembly planning domain in Section 5.2. In this section, we now use the LinGraph planner on a more complex problem instance within this planning domain. For this example, we show the object types and actions in

Figure 5.11, wherein two types of components are first transformed into two separate sub-products (through actions MakeS$_1$ and MakeS$_2$), and then they are assembled in pairs into a single product (with the action MakeP). Finally, we use the action MakeFP and then assemble four products into a final product to finish the process.

Objects:

C_1, C_2	:	Component 1 and Component 2 objects
S_1, S_2	:	Sub-product 1 and Sub-product 2 objects
M	:	Manipulator object
P	:	Product object
FP	:	Final Product object

Actions:

MakeS$_1$	:	Manipulator processes a C_1 and produces a S_1.
MakeS$_2$	:	Manipulator processes a C_2 and produces a S_2.
MakeP	:	Manipulator processes a S_1 and a S_2, finally produces a P.
MakeFP	:	Manipulator processes 4 of P and produces a FP.

Figure 5.11: Objects and actions encoding of the real world planning example.

Even though the given example is structurally similar to our previous examples, in this example, we increase problem complexity by including multiple intermediate products which requires more levels for the solution. In Figure 5.12, we encode the linear logic formulae corresponding to the actions described in Figure 5.11.

MakeS$_1$	:	$(C_1 \otimes M) \multimap (S_1 \otimes M)$
MakeS$_2$	:	$(C_2 \otimes M) \multimap (S_2 \otimes M)$
MakeP	:	$(S_1 \otimes S_2 \otimes M) \multimap (P \otimes M)$
MakeFP	:	$(P^4 \otimes M) \multimap (FP \otimes M)$

Figure 5.12: Linear logical encodings of actions within the more complex assembly planning problem domain.

Below, we give a challenging example problem. We initially have 32 components of type C_1, 32 components of type C_2 and 48 manipulators M. We want to produce 16 instances of product P and 4 instances of the product FP, using the initial components. We encode this example by the LinGraph goal formula

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap !\mathrm{MakeFP} \multimap \; (C_1^{32} \otimes C_2^{32} \otimes M^{48}) \multimap (P^{16} \otimes FP^4 \otimes M^{48}) \tag{5.13}$$

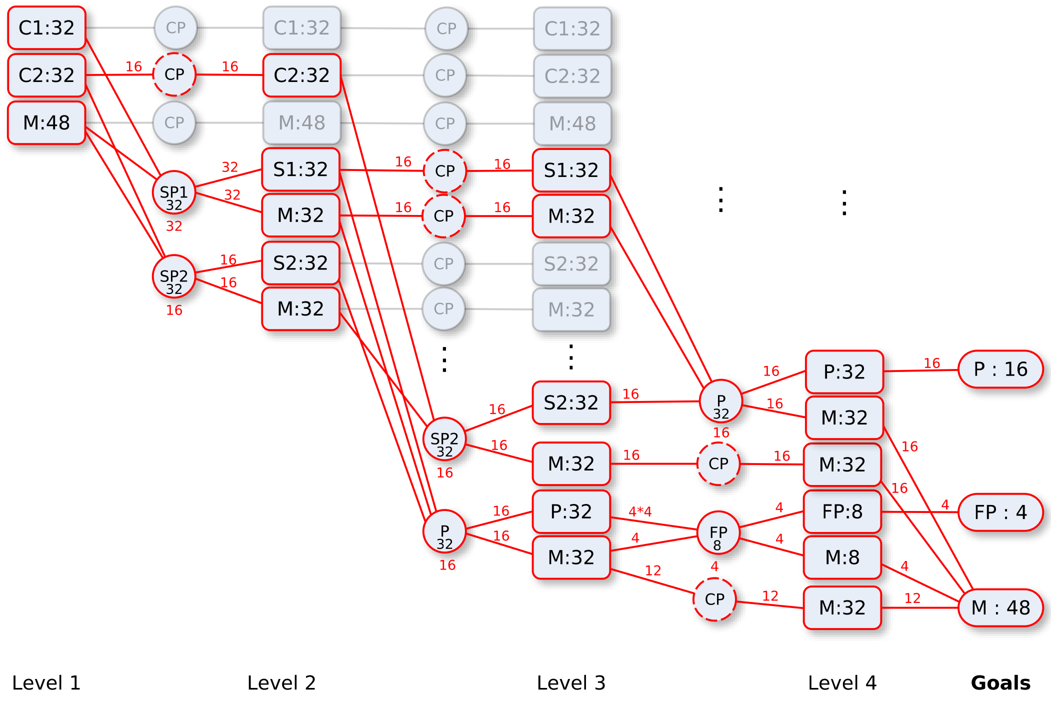

Figure 5.13: Final LinGraph for the complex assembly planning example of (5.13). Some actions and nodes for levels larger than 2 were omitted for visual clarity. Nodes that are part of the final plan are shown in red.

We choose the initial number of available components and manipulators for the purpose of both illustrating parallel execution of actions, as well as the extension of the plan to several steps due to the relatively limited number of manipulators. In Figure 5.13, we show the LinGraph generated for this example. We should note that, for clarity we removed certain parts in levels higher than two. This example is good to show that LinGraph can find a valid plan even for complex examples including large numbers of components and manipulators. We show the extracted plan from the constructed LinGraph in Figure 5.14, with multiple instances of each action executed in parallel within each step.

Step 1: MakeSP_1^{32}, MakeSP_2^{16}
Step 2: MakeSP_2^{16}, MakeP^{16}
Step 3: MakeP^{16}, MakeFP^{4}

Figure 5.14: Solution to the complex assembly planning problem defined by (5.13)

5.4 Experimental Results and the Performance of Plan Search

In this section, we evaluate the performance of LinGraph on increasingly difficult, concurrent planning problems and show that it outperforms modern, state-of-the-art automated planners for problems in domains with substantial symmetry, large numbers of functionally equivalent objects, as well as intermediate creation of objects throughout the plan. Currently, LinGraph performs blind search and does not use any heuristics to guide plan search. As a result, its performance on structurally simple, general planning problems, particularly sequential domains with long action sequences, cannot challenge existing planners. Even though we believe that relaxing the layered structure of LinGraph to support heuristic search and actions with non-unit durations will be possible to bring its performance on general planning problems to match modern planners for temporal problems, we have left these extensions for future work and focused the scope of the thesis to provide a clear description of LinGraph's novel representation of object equivalence, its correspondence to linear logic and its performance on domains with large numbers of functionally identical objects that might be created or destroyed throughout the progression of the plan.

Our evaluation focuses on comparing LinGraph's execution time to several existing planners from the literature for specific problems with the characteristics described above. First, we consider Blackbox, one of the earlier yet successful planners based on a combination of GraphPlan and SAT planning with support for concurrency and make-span optimality. Our comparisons also include Symba-2 [85], which is one of the recent high performance planners for sequential domains, having won the sequential optimal track for the 2014 IPC [86]. Next, we consider the Metis planner [87] since it implements symmetry reduction techniques that are uniquely relevant to our approach and our example domains. Due to their exclusive focus on sequential problems, these last two planners do not generate concurrent plans and hence can be considered to operate on easier instances of our domains. Finally, we include the Temporal Fast Downward (TFD) planner [88], which is among the best modern planners for temporal problems with explicit support for concurrency and make-span optimality, as well as numerical fluents which we consider to offer the best expressivity for encodings that are closest to LinGraph's approach. We have not considered YAHSP3-MT, the winner of the temporal track for IPC 2014, for comparison in this last category since it was not able to construct concurrent plans.

In considering the execution times in subsequent sections, it should be noted that our prototype LinGraph planner was implemented in SML, without any explicit optimizations for execution efficiency. In contrast, the latest, optimized implementations were used for Blackbox, Symba-2, Metis and TFD. In all cases, experiments were performed on a 2.93GHz Intel Pentium Dual-Core CPU E6500 Processor and 2 GB of RAM.

All planning examples we describe in this section were fed to our LinGraph planner in the form of LGPL goal formulas. In contrast, Blackbox, Symba-2, Metis and TFD implementations were given problem inputs encoded in PDDL. Additional PDDL features we relied on for encoding example problems included type specifications for objects and durative actions for temporal planners, as well as numerical fluents to explicitly encode object multiplicity. It is also important to remember that plans generated by the Symba-2 and Metis planners are sequential and fail to capture concurrency features of any domain.

5.4.1 Domain-1: Assembly Planning with Two Types of Components

We first start with an instance of the assembly planning domain described in Section 5.2. This domain has two distinct types of components, C_1 and C_2, first transformed into two different types of sub-products S_1 and S_2, which are then combined to make a product. In Figure 5.15, we show encoding of three actions in LGPL.

$$\boxed{\begin{array}{lcl} MakeS_1 & : & (C_1 \otimes M) \multimap (S_1 \otimes M) \\ MakeS_2 & : & (C_2 \otimes M) \multimap (S_2 \otimes M) \\ MakeP & : & (S_1 \otimes S_2 \otimes M) \multimap (P \otimes M) \end{array}}$$

Figure 5.15: Encodings of actions for Domain-1 in LGPL.

We need to give a bit more effort to encode the same problem in PDDL. In PDDL, we need to divide these encodings into two files, one for encoding the domain and its actions (the *domain file*), and one for describing the problem instance (the *problem file*) with the initial and goal states. In Figure 5.16 we show the domain file for the example in this section.

```
(define (domain assembly-domain1) (:requirements :strips :equality :typing)
 (:types COM1 COM2 SUB1 SUB2 MAN PRO)
 (:predicates (comp1 ?c1) (comp2 ?c2) (subprod1 ?s1) (subprod2 ?s2) (manip ?m) (prod ?p))

 (:action MakeSP1
   :parameters (?c1 - COM1 ?m - MAN ?s1 - SUB1)
   :precondition (and (comp1 ?c1) (manip ?m) )
   :effect (and (subprod1 ?s1) (not (comp1 ?c1)) (not (manip ?m)) (manip ?m) ))

 (:action MakeSP2
   :parameters (?c2 - COM2 ?m - MAN ?s2 - SUB2)
   :precondition (and (comp2 ?c2) (manip ?m) )
   :effect (and (subprod2 ?s2) (not (comp2 ?c2)) (not (manip ?m)) (manip ?m)))

 (:action MakeP
   :parameters (?s1 - SUB1 ?s2 - SUB2 ?m - MAN ?p - PRO)
   :precondition (and (subprod1 ?s1) (subprod2 ?s2) (manip ?m))
   :effect (and (prod ?p) (not (subprod1 ?s1)) (not (subprod2 ?s2))
         (not (manip ?m)) (manip ?m))))
```

Figure 5.16: PDDL domain file encoding Domain-1, an assembly planning example with two types of components.

In the first part of the domain file defines object types and available predicates as well as required PDDL features. The rest of the domain file has action definitions. In the PDDL encoding, the predicate (*manip* ?*m*) is a precondition to each action similar to the LGPL encoding of Figure 5.16. However, the effects of each action includes both (*manip* ?*m*) and (*not*(*manip* ?*m*)), which might seem unintuitive at first. However, we need such an encoding to ensure that the same manipulator instance is not used simultaneously by two actions within the same level by the planner by enforcing the creation of associated mutexes. The need for such *tricks* in STRIPS encodings of planning problems are a byproduct of the informal semantics associated with PDDL encodings, wherein implementation details for a particular planner may result in different semantics for the domain. We do not need such implementation specific details in LinGraph for encoding planning problems, due to its semantic correspondence to LGPL formulas.

We can describe a problem instance for this domain by specifying initial and goal states. For example, if we initially have four instances of each component and we want to produce four final products, then we can give an LGPL encoding of this problem by the goal formula

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap (C_1^4 \otimes C_2^4 \otimes M^4) \multimap (P^4 \otimes M^4)\ . \qquad (5.14)$$

```
(define (problem domain1-four-components)
(:domain assembly-domain1)
(:objects   c11 - COM1 c12 - COM1 c13 - COM1 c14 - COM1
            c21 - COM2 c22 - COM2 c23 - COM2 c24 - COM2
            m1 - MAN m2 - MAN m3 - MAN m4 - MAN
            s11 - SUB1 s12 - SUB1 s13 - SUB1 s14 - SUB1
            s21 - SUB2 s22 - SUB2 s23 - SUB2 s24 - SUB2
            p1 - PRO p2 - PRO p3 - PRO p4 - PRO)
(:goal (and (prod p1) (prod p2) (prod p3) (prod p4)
            (manip m1) (manip m2) (manip m3) (manip m4)))
(:init (comp1 c11) (comp1 c12) (comp1 c13) (comp1 c14)
       (comp2 c21) (comp2 c22) (comp2 c23) (comp2 c24)
       (manip m1) (manip m2) (manip m3) (manip m4) ))
```

Figure 5.17: Encoding the problem with four components for Domain-1 in PDDL.

In Figure 5.17, we give the *problem file* which is showing the PDDL encoding of the same example. For the PDDL problem definition as encoded in the example, we need to define all objects that may be needed throughout the plan, including any number of intermediate objects. This is not an appropriate way, since it is often difficult to know these beforehand, highlighting another advantage of the LGPL encoding that generates objects as resources whenever they are created.

```
(:durative-action MakeS1
 :parameters (?c1 - COM1 ?m - MAN ?s1 - SUB1)
 :duration (= ?duration 1)
 :condition (and (at start (comp1 ?c1))
                 (at start (manip ?m)) )
 :effect (and (at end (subprod1 ?s1))
              (at start (not (comp1 ?c1)))
              (at start (not (manip ?m)))
              (at end (manip ?m)) ))
```

Figure 5.18: PDDL encoding of the durative version of the MakeS1 action for Domain-1

The PDDL encodings of Figures 5.16 and 5.17 are appropriate for sequential planners. Temporal planners, including TFD, require support for *durative actions*. For instance, the durative version of the MakeS1 action is shown in Figure 5.18. The problem file for TFD is similar to Figure 5.17, except a new directive to request minimization of the plan's make-span.

In addition to these two, we have also used a third PDDL encoding to support a more concise and efficient representation with numerical fluents and the PDDL feature *fluents*. This feature is supported by the TFD planner, allowing us provide a fair comparison with a modern planner and an encoding that is similar in spirit to LinGraph's aggregation of functionally identical resources. Figure 5.19 illustrates the main structure of this encoding, focusing on the MakeS1 action.

In this definition, several numerical functions are defined to keep track of the available instances for the components, subcomponents and manipulators, as well as a special function `num` to allow the planner to explore using different multiplicities from each object. The definition for the MakeS1 action checks the availability of manipulators as a precondition, picks the number

```
(define (domain assembly-domain1-fluent)
 (:requirements :strips :equality :typing
                :durative-actions :fluents)
 (:functions (cnt-c1) (cnt-c2) (cnt-m) (cnt-s1)
             (cnt-s2) (cnt-p) (num ?n))
 (:durative-action MakeS1
  :parameters (?u)   :duration (= ?duration 1)
  :condition (and (at start (>= (cnt-c1) (num ?u)))
                  (at start (>= (cnt-m) (num ?u))))
  :effect (and (at start (decrease (cnt-c1) (num ?u)))
               (at start (decrease (cnt-m) (num ?u)))
               (at end (increase (cnt-s1) (num ?u)))
               (at end (increase (cnt-m) (num ?u)))))
```

Figure 5.19: PDDL encoding of Domain-1 using numerical fluents to capture object multiplicity. This figure only shows the definition of the MakeS1 action

of manipulators to use through `(num ?u)` and adjusts the postconditions accordingly. It is assumed that the problem file will define the predicate `(num ?u)` for as many positive integers as there are manipulators. Figure 5.20 shows an example problem file with six manipulators and three of each component to produce 3 final products. In all of our experimental results, we use TFD-NF to refer to this type of PDDL encoding with numerical fluents used with the TFD planner.

```
(define (problem prob-three-comp-six-manip)
(:domain assembly-domain1-fluent)
(:objects n1 n2 n3 n4 n5 n6)
(:init (= (num n1) 1) (= (num n2) 2) (= (num n3) 3)
       (= (num n4) 4) (= (num n5) 5) (= (num n6) 6)
       (= (cnt-c1) 3) (= (cnt-c2) 3)
       (= (cnt-s1) 0) (= (cnt-s2) 0)
       (= (cnt-m) 6) (= (cnt-p) 0) )
(:goal (and (= (cnt-p) 3)))
            (:metric minimize (total-time)) )
```

Figure 5.20: An example PDDL problem definition for Domain-1 using numerical fluents, featuring six manipulators to produce a total of three products

We now compare the performance of LinGraph with the performance of other planners on increasingly difficult problem instances. We first start with a problem type in Domain-1, where there are enough manipulators available to transform

n	**BB**	**S-2**	**M**	**T**	**T-N**	**LG**
1	0.004	0.292	0.144	0.022	0.096	0.056
2	0.008	0.672	0.204	7.788	0.112	0.055
3	0.021	1.104	0.604	Error	0.220	0.058
4	3.136	1.544	1.756	-	0.896	0.059
8	Error	11.88	166.2	-	257.7	0.059
32	-	Error	Error	-	Error	0.055
1000	-	-	-	-	-	0.061

Table 5.1: Execution times (in seconds) for Domain-1 with twice the number of initially available manipulators as the number of initial components n. LinGraph solutions have 4 levels

all components to sub-components in the first level. In this type of problem, LinGraph finds a solution in three levels. We give the general form of the LGPL encoding for this problem as

$$!\text{MakeS}_1 \multimap !\text{MakeS}_2 \multimap !\text{MakeP} \multimap \ (C_1^n \otimes C_2^n \otimes M^{2n}) \multimap (P^n \otimes M^{2n})\ . \quad (5.15)$$

We show resulting execution times for all planners for different values of n in Table 5.1. As we expected, optimized implementations for Blackbox, Symba-2, Metis, TFD and TFD-NF outperform LinGraph for small values of n. On the other hand, these planners have combinatorial choices for different instances of components of the same type, which quickly increase the search space, resulting in termination with a (possibly out-of-memory) errors for Blackbox, Symba-2, Metis, TFD and TFD-NF. However, LinGraph can solve arbitrarily large instances of the same problem, which is a result of aggregation of identical components. An important additional observation is that TFD-NF, despite its use of numerical fluents, fails to find valid plans after a certain problem size, whereas LinGraph performance stays constant independent of problem size.

We now describe another problem type in Domain-1, which involves a smaller number of manipulators in the initial state, requiring LinGraph plans to extend to level 4, with two new levels to produce all subproducts and a final level of actions to produce the final products. We encode the general form for this problem in LGPL as

$$!\text{MakeS}_1 \multimap !\text{MakeS}_2 \multimap !\text{MakeP} \multimap \ (C_1^n \otimes C_2^n \otimes M^{n}) \multimap (P^n \otimes M^{n})\ . \quad (5.16)$$

n	**BB**	**S-2**	**M**	**T**	**T-N**	**LG**
1	0.004	0.284	0.124	0.076	0.080	0.198
2	0.008	0.688	0.146	0.218	0.092	0.678
3	0.556	1.098	0.348	99m	0.156	0.775
4	43m	1.348	1.016	Error	0.520	0.788
5	$>$ 1d	1.964	2.616	-	2.716	0.752
1000	-	Error	Error	-	Error	0.794
10000	-	-	-	-	-	2.497

Table 5.2: Execution times (in seconds) for Domain-1 with the same number n of initially available components and manipulators. LinGraph solutions have 4 levels

We show resulting execution times for all planners for different values of n in Table 5.2.

From these results of two comparisons, we can conclude that LinGraph can successfully handle problem instances with large numbers of functionally identical objects without increasing the search space. Sequential planners, Symba-2 and Metis, perform better in this second problem instance than the first since the smaller number of manipulators decrease the available number of actions in each step. TFD experiences similar performance gains since its underlying search relies on intermediate parallelization of sequential plans. In contrast, LinGraph and Blackbox natively support concurrency, exhibiting degraded performance due to the longer make-span required for the optimal solution.

n/m	**BB**	**S-2**	**M**	**T**	**T-N**	**LG**
2/3	0.012	0.696	0.192	1.936	0.120	0.774
3/5	1.580	1.148	0.548	7.248	0.240	0.739
4/6	47 m	1.512	1.348	13.855	2.016	0.783
5/8	Error	2.112	3.912	46.272	9.300	0.777
32/48	-	Error	Error	Error	Error	0.742
1000/1500	-	-	-	-	-	0.781

Table 5.3: Execution times (in seconds) for Domain-1 with the number m of initially available manipulators a non-integer multiple of the number n of initially available components. LinGraph solutions have 4 levels

We now extend the simple problem instance above, with the initial number of manipulators chosen to be a non-integer multiple of initially available components. This eliminates the easily parallelizable nature of the previous two problem instances. We show execution times for this problem in Table 5.3. We give the general form of the LGPL encoding for this problem as

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap (C_1^n \otimes C_2^n \otimes M^m) \multimap (P^n \otimes M^m)\ , \quad (5.17)$$

which is also successfully solved by LinGraph for large values of m and n.

n	m	**Levels**	**(Last Level/All) Nodes**	**Pruned**	**Dependency**	**Sibling**
1	1	4	15/36	70	21	6
2	2	4	87/116	136	29	42
3	3	4	185/216	60	31	91
4	4	4	221/252	24	31	109
5	5	4	223/254	22	31	110
1000	1000	4	223/254	22	31	110
32	48	4	223/254	22	31	110
32	64	3	21/31	4	10	9

Table 5.4: LinGraph level, node and constraint statistics for different instances of the planning problem in (5.17).

In Table 5.4, we show node and constraint statistics for LinGraph solutions of Domain-1 problems, which are using the problem definition of (5.17). We should also note that, even when problem sizes are increased dramatically for Domain-1 problem instances, the execution times for LinGraph remain the same in Tables 5.1, 5.2 and 5.3. This is a result of a fixed number of possible combinations for different types of objects and actions in the problem.

Level	**Nodes**	**Pruned**	**Dependency**	**Sibling**
1	3	0	0	0
2	7	0	3	2
3	21	4	7	7
4	223	18	21	101

Table 5.5: Node and constraint statistics for each level of the LinGraph solution for a Domain-1 problem instance with $n = 10000$ and $m = 10000$.

In Table 5.5, we also show node and constraint statistics for the expansion of each intermediate level as we create the LinGraph for the Domain-1 problem instance with $n = 10000$ and $m = 10000$. When the LinGraph reaches level 4, the number of created nodes increase more then the previous levels, since many new combinations becoming available for how actions can be applied to nodes in level 3. However, all problem domains do not entail such exponential increase in the number of nodes, as we will show in subsequent sections.

One of the reasons for LinGraph's difficulty handling larger limits for plan size is that its encoding of this problem instance includes manipulator objects both as preconditions and effects of actions. Consequently, the pruning check during action creation cannot immediately detect repeated uses of the same manipulator on copied node instances from previous levels, resulting in the inability of the planner to detect redundant applications of the same action on copied instances of resources. We can use an additional heuristic identifying recreated instances of reusable resources such as manipulators to eliminate this complexity, but this is left for future work. Nevertheless, despite decreased performance for these worst case examples in the absence of this added heuristic, the ability of LinGraph to quickly identify solutions for different scenarios justifies its use as an alternative planner.

5.4.2 Domain-2: Assembly Domain Without Trivial Parallelism

In this section, we introduce a new domain. In addition to those actions listed in Figure 5.15 for Domain-1, MakeS$_1$, MakeS$_2$ and MakeP, we add a new action MakeFP to this domain, enforcing that multiple Product instances are assembled in a final step into a so called *Final Product.* We define the new action as

$$\text{MakeFP} : P \otimes P \otimes P \otimes P \otimes M \multimap FP \otimes M \ .$$

We also add this action to the PDDL domain definition file. We show the additional action definition in Figure 5.21. We should note that, the PDDL encoding of the problem, similar to the handling of preconditions for manipulators, requires the *trick* of including the negation of product instances to prevent the planner from using the same object and its predicate to satisfy all four preconditions.

On the other hand, we do not need such an additional countermeasure for LinGraph encoding of Domain-2, since preconditions in LinGraph are considered to be single-use resources.

```
(:action MakeFP
   :parameters (?p1 - PRO ?p2 - PRO ?p3 - PRO ?p4 - PRO ?m - MAN ?fp - FPRO)
   :precondition (and (prod ?p1) (prod ?p2) (prod ?p3) (prod ?p4) (manip ?m))
   :effect (and (finalprod ?fp) (not (prod ?p1)) (not (prod ?p2))
        (not (prod ?p3)) (not (prod ?p4)) (not (manip ?m)) (manip ?m)))
```

Figure 5.21: New action definition in the PDDL domain file encoding of Domain-2.

One of instances of this domain is where we convert all products into final products. Another instance of this domain is where there are leftover products that are insufficient in number to be converted into final products. We start with encoding the first problem instance by the LGPL goal formula

$$!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap !\mathrm{MakeFP} \multimap (C_1^{4n} \otimes C_2^{4n} \otimes M^{8n}) \multimap (FP^n \otimes M^{8n}) \;, \tag{5.18}$$

wherein we convert all initial $4n$ instances of components into $4n$ products, and then we assemble them into n final products. We show execution times for all planners solving this problem instance in Table 5.6.

n	**BB**	**S-2**	**M**	**T**	**T-N**	**LG**
1	101.484	1.94	2.544	1.416	1.572	1.907
2	Error	78.696	27 min	> 1 d	Error	1.902
4	-	Error	Error	-	-	1.968
250	-	-	-	-	-	2.151
2500	-	-	-	-	-	8.087

Table 5.6: Execution times (in seconds) for the problem of (5.18). LinGraph plans have 4 levels

We encode the second problem instance in this domain by the LGPL formula

$$\begin{aligned} &!\mathrm{MakeS}_1 \multimap !\mathrm{MakeS}_2 \multimap !\mathrm{MakeP} \multimap !\mathrm{MakeFP} \\ &\qquad \multimap (C_1^n \otimes C_2^n \otimes M^{2n}) \multimap (P^m \otimes FP^r \otimes M^{2n}) \;, \end{aligned} \tag{5.19}$$

where from the initial n available components, and the resulting n products, not all are converted into final products, resulting in m leftover products P such that

$n/m/r$	**BB**	**S-2**	**M**	**T**	**T-N**	**LG**
5/1/1	Error	3.908	8.884	Error	13.69	1.847
6/2/1	-	8.676	49.84	-	203.1	1.910
9/1/2	-	243.7	> 1d	-	Error	1.997
19/3/4	-	Error	-	-	-	2.054
1200/400/200	-	-	-	-	-	4.804

Table 5.7: Execution times (in seconds) for the problem of (5.19). LinGraph plans have 4 levels

$4r + m = n$. We show execution times for all planners solving increasing sizes of this problem instance in Table 5.7.

We can conclude that when the problem domain does not allow trivial parallelization for multiple object instances of the same type, LinGraph is still capable of generating feasible plans whereas the computational complexity of planners that consider different instances of such objects as distinct individuals quickly becomes impractical.

5.4.3 RHex hexapod robot domain

In our previous work [89], we introduced a novel logic language and theorem prover, *Linear Planning Logic (LPL)*, for robotic task planning and we presented a real world planning domain with the RHex hexapod robot. The example domain consists of a mobile robot, the RHex hexapod in this case, navigating in an environment populated with uniquely identifiable colored landmarks. Paths between each pair of landmarks can have surfaces with different traversability properties appropriate for different locomotory gaits of the robot. Moreover, the robot is assumed to have the capability of *tagging* landmarks, corresponding to a particular action carried out at that location. We capture this aspect of the domain with a persistent binary state, *tagged* or *untagged*, associated with each landmark. Even though LinGraph has a more restricted language than *LPL*, we can still express the presented domain with LinGraph. Generating a long sequence of actions (14 total) for this example takes 60 seconds in *LPL* while it takes 2 seconds in LinGraph.

Levels	Actions	Created Nodes	Pruned Nodes	Dependency Constraints	Sibling Constraints
1	$Seek\_b_1$	24	0	0	0
2	$Walk\_b_1$	1	0	24	0
3	$Seek\_b_0$	4	1	25	1
4	$Walk\_b_0$	2	17	29	0
5	$Seek\_b_3$	5	19	31	1
6	$Walk\_b_3$	3	42	36	0
7	Tag_b_3	6	45	39	1
8	$Seek\_b_4$	6	84	45	1
9	Run_b_4	6	126	51	1
10	$Seek\_b_2$	6	189	57	1
11	Run_b_2	6	270	63	1
12	$Seek\_b_5$	5	351	69	1
13	Run_b_5	4	449	74	1
14	Tag_b_5	6	549	78	2
15		3	726	84	1

Table 5.8: Created nodes and constraints at each level for the Rhex example in LinGraph.

5.4.4 Domain-3: Cooperative Multi-Robot Assembly

In this section, we introduce a new domain for assembling bicycles. This domain includes the coordination of multiple functionally identical robots for the transportation and assembly of a different types of bicycle parts. We transport wheel and body parts into a central station, and then assemble them into bicycles. In this example, there are three stations, two supply stations l_1 and l_2 containing wheel and body parts respectively, and a third base station l_0 for hosting transportation robots and performing bicycle assembly. We show an illustration of the initial state for this domain in Figure 5.22.

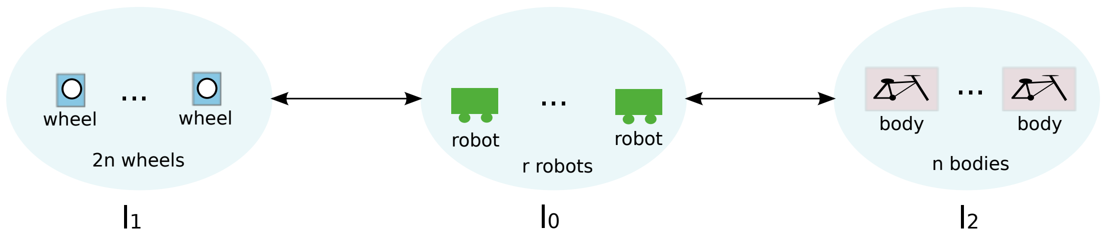

Figure 5.22: The cooperative multi-robot domain consisting of two supply stations, l_1 and l_2 and a base station l_0. This figure also illustrates the initial state for our examples, wherein $2n$ wheels, n body parts and r robots are available in their corresponding stations. Arrows indicate connections between locations traversable by robots.

In this domain, robots can move between different locations connected with traversable paths. Robots can move objects between locations by picking up and then carrying objects. We begin the LGPL encoding for this domain with defining propositional atoms encoding components of the problem state, and individual actions for transforming state. We summarize these components in Figure 5.23, together with LGPL definitions and descriptions for all actions.

Objects:		
$wheelat_x$	:	There is a wheel in location l_x
$bodyat_x$	:	There is a body in location l_x
$bicycleat_x$	:	There is a bicycle in location l_x
$rbtat_x$	:	There is a robot in location l_x
$wheelteamat_x$	:	A two robot team holding a wheel are in location l_x
$bodyteamat_x$	:	A three robot team holding a body are in location l_x
Actions:		
Move_{xy}	:	$rbtat_x \multimap rbtat_y$
		(Robot moves from l_x to l_y)
$\mathrm{HoldWheel}_x$	:	$(rbtat_x)^2 \otimes wheelat_x \multimap (wheelteamat_x)$
		(Two robots pick up a wheel in location l_x)
$\mathrm{HoldBody}_x$	:	$(rbtat_x)^3 \otimes bodyat_x \multimap (bodyteamat_x)$
		(Three robots pick up a body in location l_x)
$\mathrm{ReleaseWheel}_x$	:	$wheelteamat_x \multimap (rbtat_x)^2 \otimes wheelat_x$
		(Two robots release a wheel in location l_x)
$\mathrm{ReleaseBody}_x$	:	$bodyteamat_x \multimap (rbtat_x)^3 \otimes bodyat_x$
		(Three robots release a body in location l_x)
$\mathrm{CarryWheel}_{xy}$	:	$wheelteamat_x \multimap wheelteamat_y$
		(Two robots carry a wheel from location l_x to location l_y)
$\mathrm{CarryBody}_{xy}$	:	$bodyteamat_x \multimap bodyteamat_y$
		(Three robots carry a body from location l_x to location l_y)
$\mathrm{AssembleBicycle}_x$	:	$bodyat_x \otimes wheelat_x^2 \multimap bicycleat_x$
		(A bicycle is assembled in location l_x)

Figure 5.23: LGPL encoding of the example for cooperative multi-robot assembly of bicycles. Propositions encoding components of the state, together with LGPL action definitions and their descriptions are given.

We illustrate the final state for our example in Figure 5.24, wherein the base station contains n fully assembled bicycles and r robots. Below, we give the general form of the LGPL goal formula encoding this example problem by

$$\begin{aligned}&!\mathrm{Move}_{xy} \multimap !\mathrm{HoldWheel}_x \multimap !\mathrm{HoldBody}_x \multimap !\mathrm{ReleaseWheel}_x \multimap \\ &!\mathrm{ReleaseBody}_x \multimap !\mathrm{CarryWheel}_{xy} \multimap !\mathrm{CarryBody}_{xy} \multimap !\mathrm{AssembleBicycle}_x \multimap \\ &({rbtat_0}^r \otimes {wheelat_1}^{2n} \otimes {bodyat_2}^n) \multimap ({rbtat_0}^r \otimes {bicycleat_0}^n)\ . \qquad (5.20)\end{aligned}$$

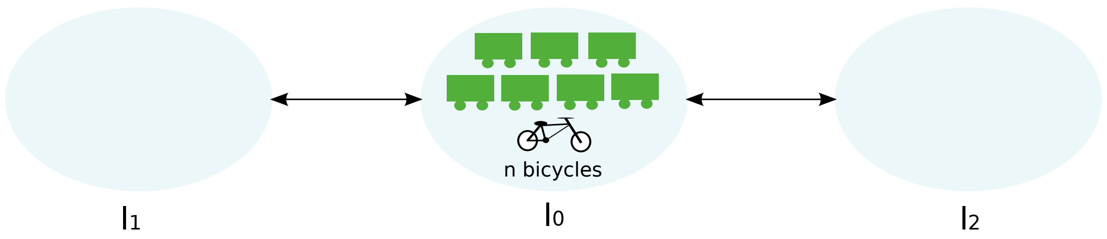

Figure 5.24: The desired final state for the example multi-robot cooperative assembly domain. n fully assembled bicycles and r robots should be located in the base station l_0.

Since LGPL currently lacks quantifiers, we must explicitly instantiate actions defined in Figure 5.23 for each location or location pair in the goal statement. We particularly instantiate $Move_{xy}$ actions only for pairs of locations connected by traversable paths. For clarity, we have kept the quantified versions of the action labels in the goal formula above.

Step 1:	Move^4_{01},	Move^3_{02}
Step 2:	$\mathrm{HoldWheel}^2_1$,	$\mathrm{HoldBody}_2$
Step 3:	$\mathrm{CarryWheel}^2_{10}$,	$\mathrm{CarryBody}_{20}$
Step 4:	$\mathrm{ReleaseWheel}^2_0$,	$\mathrm{ReleaseBody}_0$
Step 5:	$\mathrm{AssembleBicycle}_0$	

Figure 5.25: Solution to the multi-robot example encoded by (5.20) with $n = 1$ and $r = 7$, generated by the LinGraph planner.

When we assign $n = 1$ and $r = 7$, LinGraph finds the solution as we show in Figure 5.25. Even though this is the simplest instance of the problem with only a small number of functionally identical objects, only BlackBox and sequential planners Symba-2 and Metis were capable of finding feasible solutions, with TFD failing with errors. As we show in Table 5.9, beyond $n = 3$ and $r = 21$, only LinGraph continues to find solutions as a result of its aggregation of functionally identical objects.

We now extend the plan length to 9 steps from 5 steps by decreasing the number of available robots. All temporal planners fail, whereas the sequantial planners Symba-2 and Metis can only find solutions for the simplest case, with execution times substantially larger than those of LinGraph. In Table 5.10, the execution times are detailed. In summary, these examples show that LinGraph

n/r	**BB**	**S-2**	**M**	**T**	**T-N**	**LG**
1/7	38 min	50 min	445.6	Error	Error	0.130
2/14	Error	Error	Error	-	-	0.130
3/21	-	-	-	-	-	0.131
4/28	-	-	-	-	-	0.130
128/896	-	-	-	-	-	0.130

Table 5.9: Execution times (in seconds) for different planners in solving increasingly large instances of the planning problem encoded by (5.20). LinGraph extends to level 6 in all cases, corresponding to 5 steps. Other planners are unable to solve any of the problem instances

is much more capable than existing automated planners in exploiting symmetries in concurrent planning problems with large numbers of functionally identical objects in ways that are not possible with current methods for symmetry reduction including the use of numerical fluents.

n/r	**BB**	**S-2**	**M**	**T**	**T-N**	**LG**
1/5	> 1 d	30.986	41.15	Error	Error	1.524
2/10	-	> 1 day	Error	-	-	3.846
4/20	-	-	-	-	-	3.846
128/640	-	-	-	-	-	4.350

Table 5.10: Execution times (in seconds) for different planners on the planning problem of (5.20) with fewer manipulators. LinGraph extends to level 10 in all cases (9 steps). Other planners abort with errors beyond a certain problem size

5.5 LinGraph, Multiset Rewriting and Linear Logic

In this section, we present a formalization of the connection between LinGraph and multiset rewriting, enabling us to prove soundness and completeness with respect to the LGPL fragment of linear logic. In doing so, we also establish LinGraph as a theorem prover for its underlying fragment of intuitionistic linear logic. In earlier sections, our focus has been the use of LinGraph as an automated, model-based, domain-independent planner. However, LinGraph is also closely connected to theorem proving in the multiplicative exponential fragment of propositional, intuitionistic linear logic defined by the grammar in Section 5.1. In this section, we show that LinGraph construction provides a sound and complete method for proof construction in this logic by first interpreting LinGraph as a multiset rewriting system, then using the well-known correspondence between such systems and theorem proving in linear logic [62, 90, 91]. In this context, it is important to note that LinGraph does not simply encode a simple multiset rewriting process. A partially constructed LinGraph in fact captures *all possible* multiset rewriting sequences associated with the problem in its graph structure. This ensures completeness, allowing LinGraph to find all valid plans for a problem by exploring different solutions to its cumulative constraints. The main benefit provided by LinGraph is the reduction in the complexity of plan (and proof) search through the aggregation of functionally identical resources in individual nodes.

We begin by adapting and reviewing key definitions on multiset rewriting systems from [90]. Given a set of propositions P, a finite *multiset* over P is a function $M : P \to \mathbb{N}$ such that $M(p)$ gives the multiplicity of the proposition p in the multiset. We write $p \in M$ if $M(p) \neq 0$. As noted in [92], a multiset $[p_1, ..., p_k]$ over P can also be associated with the product expression $(p_1 \otimes ... \otimes p_k)$. As such, multiplicative conjunction of two product expressions corresponds to the *additive multiset union* of two multisets M_1 and M_2 over P, which is the multiset defined by $(M_1 \uplus M_2)(p) := M_1(p) + M_2(p)$.

A *multiset rewriting rule* R is an ordered pair of multisets, $R = (M_1, M_2)$ over the set of propositions P, where M_1 and M_2 are called the *preset* and the *postset*, respectively. A rule $R = (M_1, M_2)$ is said to be *applicable* on a multiset M if $\forall p \in P, M_1(p) \leq M(p)$ encoding the requirement that the preset M_1 of R is a submultiset of M. In such cases, the application of the rule R on the multiset M *generates* a new multiset $M_{new} = (M \setminus M_1) \uplus M_2$ where $\setminus$ denotes multiset difference. Based on this definition, a *multiset rewriting system* (MRS) is simply a set of rewriting rules, $\mathcal{R} = \{R_1, ..., R_N\}$ which, in the context of problems in task planning, can be used to represent possible actions in a domain to transform system state.

Definition 3 *(Multiset of a LinGraph Node) Given a LinGraph L with r nodes, and a particular solution vector* $[s_1, ..., s_r]$ *satisfying all of its cumulative constraints, each node* n_i *having the type encoded by the propositional atom* P_i *uniquely defines an associated multiset* $M_{nd}(n_i) := \{P_i, ..., P_i\}$ *consisting of* s_i *copies of the proposition* P_i*.*

Definition 4 *(Multiset of a LinGraph level) Given a LinGraph L with r nodes, l levels and a particular solution vector* $[s_1, ..., s_r]$ *satisfying all of its cumulative constraints, every level* $j \leq l$ *in L uniquely defines an associated multiset* $M_{lvl}(j) := M_{nd}(n_{i_1}) \uplus ... \uplus M_{nd}(n_{i_m})$*, where* i_1 *through* i_m *denote indices of all LinGraph nodes that belong to level j.*

Note, also, that since every action formula $F_a = p_1 \otimes ... \otimes p_k \multimap q_1 \otimes ... \otimes q_t$ can be associated with a corresponding multiset rewriting rule $R_{F_a} = (\{p_1, ..., p_k\}, \{q_1, ..., q_t\})$, every LinGraph instance also defines a corresponding MRS, consisting of rewriting rules associated with all of its actions, together with rules corresponding to trivial "copy" actions $p \multimap p$ for all propositions p. We will denote this MRS with $\mathcal{R}_L$. The following theorem is a key step in establishing that every LinGraph, when considered together with a particular solution with its constraints, encodes a valid sequence of rewriting rule applications starting from its initial level, ending with its last level.

Theorem 8 *(LinGraph Expansion) Given a LinGraph L with r nodes, two successive levels i and* $i + 1$ *and a particular solution vector* $[s_1, ..., s_r]$ *satisfying all of its cumulative constraints, there exists a (not necessarily unique) sequence of rewriting rules,* $[R_1, ..., R_u]$ *with* $\forall j \leq u, R_j \in \mathcal{R}_L$*, with an associated sequence of intermediate multisets* $[M_0, ..., M_u]$ *such that* $M_0 = M_{lvl}(i)$*,* $\forall j \leq u, M_j$ *is generated by* R_j *from* M_{j-1} *and* $M_u = M_{lvl}(i + 1)$*.*

Proof 8 *The proof proceeds by iteration through the set of action nodes connecting level i to level* $i + 1$ *in the LinGraph L. Consider a particular ordering of these actions,* $\mathcal{A} = [F_{A_1}, ..., F_{A_k}]$*. First of all, as a result of sibling constraints* *defined in Section 5.3.2.5 and the structure and construction of the LinGraph,* all resources *in the final multiset $M_{lvl}(i+1)$ can be associated with their corresponding generating actions, including copy actions. Consequently, if a proposition $p \in M_{lvl}(i+1)$, then the corresponding node count satisfies $s_p > 0$ and by construction, there exists an action $F_A \in \mathcal{A}$ has p in its postconditions. Moreover, as a result of the sibling constrains being satisfied, $M_{lvl}(i+1)$ is also guaranteed to have a consistent number of propositions corresponding to all the postconditions of this action. This ensures that $M_{lvl}(i+1) \subseteq M_u$.*

On the other hand, dependency constraints between the node multiplicities ensure that exactly the number of available resources in level i are consumed by the collective application of all actions connecting level i to level $i+1$. Consequently, the multiset $M_{lvl}(i)$ is entirely replaced by the postconditions of all the actions in A. This ensures that only nodes in level $i+1$ can be in the corresponding multiset, meaning that $M_u \subseteq M_{lvl}(i+1)$. This proves that $M_u = M_{lvl}(i+1)$. All the intermediate multisets M_1 through M_{u-1} are then generated by successive application of multiset rewriting rules associated with the actions in A.

In order to establish the soundness of LinGraph with respect to LGPL semantics, we first review standard proof theoretic semantics for the $LGPL$ language, adapted from [81]. We use a sequent calculus formulation, with the corresponding sequent definition

$$\Gamma; \Delta \Rightarrow \mathcal{F}_G \ , \tag{5.21}$$

where Δ is a multiset of atomic resources and Γ is a multiset of action formulae. This sequent states that the goal formula $\mathcal{F}_G$ can be proven using the single-use resources in Δ and unlimited use propositions in Γ. Sequent calculus is generally used to formalize proof construction for logical languages, wherein “left” and “right” inference rules capture how occurrences of connectives as assumptions or goals, respectively, can be refined to recursively construct a valid proof. Sequent calculus rules for $LGPL$ are detailed in Figure 5.26, where we also follow the conventions of [81]. We provide these standard sequent rules for the LGPL fragment of linear logic to provide a basis for LinGraph’s soundness properties and will omit detailed descriptions for space considerations. Interested readers may refer to the literature on proof theoretic semantics for intuitionistic linear logic for further details.

$$\frac{}{\Gamma; p \Rightarrow p}\ init \qquad \frac{(\Gamma, \mathcal{F}_A); (\Delta, \mathcal{F}_A) \Rightarrow \mathcal{F}_G}{(\Gamma, \mathcal{F}_A); \Delta \Rightarrow \mathcal{F}_G}\ copy$$

$$\frac{(\Gamma, \mathcal{F}_A); \Delta \Rightarrow \mathcal{F}_G}{\Gamma; (\Delta, !\mathcal{F}_A) \Rightarrow \mathcal{F}_G}\ !L \qquad \frac{\Gamma; \Delta_1 \Rightarrow p \quad \Gamma; \Delta_2 \Rightarrow \mathcal{F}_R}{\Gamma; \Delta_1, \Delta_2 \Rightarrow p \otimes \mathcal{F}_R}\ \otimes R$$

$$\frac{\Gamma; \Delta, p, \mathcal{F}_R \Rightarrow \mathcal{F}_G}{\Gamma; \Delta, p \otimes \mathcal{F}_R \Rightarrow \mathcal{F}_G}\ \otimes L \qquad \frac{\Gamma; \Delta, \mathcal{F}_P \Rightarrow \mathcal{F}_G}{\Gamma; \Delta \Rightarrow \mathcal{F}_P \multimap \mathcal{F}_G}\ \multimap R$$

$$\frac{\Gamma; \Delta_1 \Rightarrow \mathcal{F}_{R1} \quad \Gamma; \Delta_2, \mathcal{F}_{R2} \Rightarrow \mathcal{F}_G}{\Gamma; \Delta_1, \Delta_2, \mathcal{F}_{R1} \multimap \mathcal{F}_{R2} \Rightarrow \mathcal{F}_G}\ \multimap L$$

Figure 5.26: Sequent calculus proof rules for $LGPL$

All together, these inference rules enforce the requirement that all resources appearing in Δ on the left hand side of a sequent must be used exactly once in the associated proof, thereby allowing their interpretation as consumable resources [54]. This property, combined with our choice of an *intuitionistic* fragment of linear logic make it possible to associate each proof for $LGPL$ goal formulae with a valid plan.

Theorem 9 *(Soundness of LinGraph) Given a goal formula F_G in LGPL, the corresponding completed LinGraph L with r nodes, its last level l matching the decomposed goal from Section 5.3.2.2 and a valid solution vector $[s_1, ..., s_r]$ satisfying all of its cumulative constraints, there exists a corresponding proof in the sequent calculus system of Figure 5.26 for F_G.*

Proof 9 *LGPL formulae have the form shown in (5.2). Proving the validity of this formula corresponds to proving the sequent $(F_{A_1}, ..., F_{A_a}); (F_{R_1}, ..., F_{R_b}) \Rightarrow F_{R_g}$ in the system of Figure 5.26, following straightforward elimination of implications and exponentials using the rules $\multimap L$, $\otimes L$ and $!L$. In the LinGraph initialization phase, all resource formulae in the linear context, F_{R_1} through F_{R_b} are converted* *into nodes, forming the first layer of the graph. Together with associated initial constraints described in Section 5.3.2.2, these ensure that the multiset $M_{lvl}(1)$ exactly corresponds to the linear context in this sequent. Similarly, the goal formula F_{R_g} is converted into a set of goal nodes which, through the use of goal check constraints satisfied in the last stage of LinGraph construction ensure that the multiset $M_{lvl}(l)$ associated with the last level of L exactly corresponds to the desired goal formula F_{R_g}. Iterative application of Theorem 8, shows that there is a finite sequence of multiset rewriting rule applications connecting the initial multiset to the final multiset associated with the goal formula. Finally, this sequence of rule applications can be transformed into a proof in the multiplicative Horn fragment of linear logic, establishing the soundness of LinGraph construction [90].*

Soundness is of course a necessary property for any automated planner. Equally important, however, is completeness with respect to the underlying logic. To prove completeness, we rely on the ability of LinGraph to capture all possible sequences of multiset rewriting rule applications through different solutions to its set of constraints.

Theorem 10 *(Completeness of LinGraph) Given a goal formula F_G in LGPL, provable in the system of Figure 5.26, the algorithm described in Section 5.3.2 is guaranteed to terminate and produce a valid LinGraph for F_G.*

Proof 10 *Without loss of generality, the proof of the formula F_G, having the form in(5.2), can be assumed to start with the elimination of implications and exponentials through the associated invertible rules, thereby having a subproof for the sequent*

$$(F_{A_1}, ..., F_{A_a}); (F_{R_1}, ..., F_{R_b}) \Rightarrow F_{R_g} \ , \tag{5.22}$$

Since this sequent is free of exponentials, the corresponding proof will be in the multiplicative Horn fragment of linear logic, and hence can be transformed into a sequence $S = [R_1, ..., R_u]$ of multiset rewriting rule applications corresponding to left rules for actions, $(F_{A_1}, ..., F_{A_a})$, starting from the initial multiset associated *with the initial context $I = (F_{R_1}, ..., F_{R_b})$ [90]. By construction, action nodes in a particular LinGraph level capture* all possible applications *of available multiset rewrite rules, each of which corresponding to a particular solution for the cumulative set of constraints associated with the LinGraph.*

Even though the multiset rewriting rules defined above are applied sequentially, they naturally encode concurrency through the independence of their preset and postsets [91, 93]. Consequently, the sequence S defined above can be permuted to collect all mutually independent applications related to the initial multiset I to the beginning of the sequence. By construction, each such rule applied to the initial multiset, will then have a corresponding action node in the LinGraph, with the subsequent level having nodes associated with the elements in the postset of the rule. Once all rules applicable to the initial set are exhausted, the second level of the LinGraph will be formed, and the construction will proceed recursively until all the rules in S are covered. Finally, since the last level in the LinGraph constructed in this fashion will have nodes corresponding to all components of F_{R_g}, the goal check is guaranteed to succeed, ensuring termination of LinGraph construction, with a valid solution for all initial, dependency and goal check constraints in the LinGraph.

From a planning perspective, LinGraph semantics are hence sound and complete based on existing work in planning and linear logic [82, 94]. From a theorem proving perspective, LinGraph allows efficient proof construction in the corresponding fragment of linear logic when there are considerable object symmetries.

Chapter 6

Conclusion and Future Work

In this thesis, we introduce two different languages and associated theorem provers for domain independent task planning using linear logic. The first language that we introduce is Linear Planning Logic (*LPL*), a fragment of linear logic sufficiently expressive and suitable for representing task planning problems. We presented the LPL grammar, sequent definitions and the associated backchaining proof theory through informative examples. One of the primary novel contributions of our work is the design and implementation of a backchaining theorem prover in the presence of program formulae with non-atomic conclusions that are necessary for efficient representations of planning problems. Subsequently, we have demonstrated both the expressivity and feasibility of the LPL language and our associated theorem prover on an application domain that involves a mobile robot visually navigating through an environment populated with colored landmarks. We have presented both a formal encoding of this domain in LPL, as well as an example scenario and associated experiments with the RHex hexapod platform. Possible future directions for this work include more efficient implementations of the LPL theorem prover, extensions to incorporate nondetermistic actions in the domain through disjunctive and additive connectives as well as integration with continuous constraint expressions to enable more accurate modeling of physical properties of robotic behaviors and reactive components and provide support for dynamic environments.

Our second introduced language is Linear Logic Graph Planner (*LinGraph*), a new, model-based, domain-independent automated planner based on encodings of planning problems within a fragment of propositional intuitionistic Linear Logic (LGPL), and a graph-based strategy for plan generation. Our planner benefits from the non-monotonicity of linear logic in addressing the frame problem, with linear assumptions interpreted as resources to allow simple and effective encoding of dynamic state. In this context, our novel graph-based method, inspired from GraphPlan, implements forward proof-search within the LGPL fragment of linear logic, providing an effective means of automated plan construction. The intuitionistic nature of LGPL preserves strict correspondence between valid proofs and plans, admitting the use of this system as an automated planner. Finally, the effectiveness of linear logic in capturing and representing concurrency allows LinGraph to perform well for concurrent planning problems with multiple agents.

A distinguishing feature of LinGraph is its ability to eliminate irrelevant combinatorial nondeterminism in plan search when there are multiple, functionally identical objects within a problem. Our graph-based search strategy aggregates multiple instances of such indistinguishable components of the state, admitting efficient plan search for problems that are otherwise intractable. We illustrate both the basic operation of the planner, as well as its performance on increasingly complex instances of a simple assembly planning domain incorporating multiple identical instances of different types of components and manipulators. In this context, we provide a comparison of execution times for LinGraph with four planners: Blackbox, Symba-2, Metis and the Temporal Fast Downward (TFD) system, using the TFD system first with a simple encoding treating objects as unique, and a second using numerical fluents to more efficiently encode equivalent object multiplicity. We show that even though our unoptimized implementation of LinGraph, with its current blind search strategy, does not necessarily outperform existing planners for small concurrent problems or long sequential problems, it is much more scalable and maintains its ability to identify feasible plans for increased problem sizes, particularly in the presence of larger numbers of functionally identical objects. We also investigated the performance of LinGraph in the context of a multi-robot domain that involves coordinated manipulation and transportation of objects and show that it can handle much larger problems than other planners.

In addition to a presentation of LinGraph and these experimental results, we provided a brief analysis of algorithmic properties of LinGraph, followed by theoretical results establishing the soundness and completeness of LinGraph's plan construction with respect to a standard proof theory for the LGPL fragment of intuitionistic linear logic. This was accomplished by interpreting LinGraph instances as multiset rewriting systems, which are known to correspond to Petri Nets and also proofs in Horn fragments of propositional linear logic. These results provide a formal connection between LinGraph generated plans and the semantics of goal formulae in the LGPL fragment of linear logic.

Possible extensions to our work can be categorized in three directions. First, different sources of complexity in LinGraph construction can be addressed. In particular, defining a concept of state node equivalence, and introducing the ability to combine equivalent nodes during graph expansion might substantially reduce LinGraph size and allow longer plans to be generated. This could be extended with the detection of loops wherein the effects of an action are completely reversed by a subsequent action. Further expansion of such looping states could be suppressed, resulting in further reductions in graph size. Methods in existing planning literature focusing on state equivalence and loop detection could be applicable to LinGraph to address these issues. The second category of improvements involve the action creation and goal check stages. Currently, LinGraph expands the graph in a strict layered structure, performing blind, breadth-first search. Incorporating effective heuristics to properly guide graph generation would substantially improve performance on sequential problems, wherein LinGraph is far from being practical. This is one of the very first extensions that should be considered for LinGraph to be applicable to general planning problems. Similarly, the current LinGraph implementation uses an uninformed constraint solver to identify a feasible assignment of usage counts for all state nodes during the goal check, and partial usage counts for action creation. A new instance of the constraint solver is invoked for every check, discarding previous work that might have helped eliminate assignments that are altogether infeasible. A tighter integration with graph generation and an incremental constraint solver could improve the performance of both graph generation and the goal check, allowing, once again, handling of larger planning problems. The third and final direction for future work involves extensions to the expressivity of the language underlying the semantics of LinGraph. The LGPL language only allows discrete actions with no explicit support for conditional statements or nondeterminism. On the other hand, linear logic introduces additive connectives an other components that could potentially provide richer expressivity for encoding planning problems. Such more expressive uses of linear logic for planning problems has not been sufficiently explored in previous literature, but could allow more uniform modeling of more advanced planning structures such as hierarchical plans, dynamic management of available actions and nondeterminism. In addition to the semantic formulation of the connection between these logical connectives and the planning domain, both the structure and the construction of LinGraph would need to be extended to correctly handle proof construction in the presence of new connectives. Other possible extensions include the incorporation of continuous constraint expressions into linear logic as was proposed in [95], allowing native modeling of hybrid systems.

Appendices

Appendix A

Formal Representation of LinGraph

In this section, we introduce the formal representation of LinGraph. We also give details of each element used in LinGraph. We recall the grammar of the LGPL language in Section 5.1,

$$\begin{array}{lrl}
\text{Resource formulas:} & \mathcal{F}_R ::= & p \mid (p \otimes \mathcal{F}_R) \\
\text{Action formulas:} & \mathcal{F}_A ::= & (\mathcal{F}_R \multimap \mathcal{F}_R) \\
\text{Program formulas:} & \mathcal{F}_P ::= & !\mathcal{F}_A \mid \mathcal{F}_R \\
\text{Goal formulas:} & \mathcal{F}_G ::= & \mathcal{F}_R \mid (\mathcal{F}_P \multimap \mathcal{F}_G)\ ,
\end{array}$$

where p denotes atomic resources, $\mathcal{F}_R$ denotes linear formulae that can be used as resources, $\mathcal{F}_A$ denotes implicative formulae that represent actions, $\mathcal{F}_P$ captures program formulae that can appear as assumptions and $\mathcal{F}_G$ denotes goal formulae. Using these definitions, we introduce $\mathcal{L}_\mathcal{N}, \mathcal{L}_\mathcal{A}, \mathcal{L}_\mathcal{G}$ which are sets of labels such that

$$\mathcal{L}_\mathcal{N} \cap \mathcal{L}_\mathcal{A} = \emptyset$$
$$\mathcal{L}_\mathcal{N} \cap \mathcal{L}_\mathcal{G} = \emptyset$$
$$\mathcal{L}_\mathcal{A} \cap \mathcal{L}_\mathcal{G} = \emptyset$$

where $\mathcal{L}_{\mathcal{N}}$ stands for Node Labels, $\mathcal{L}_{\mathcal{A}}$ stands for Action Labels and $\mathcal{L}_{\mathcal{G}}$ stands for Goal Labels. Next, we present the formal definition of each LinGraph element.

Definition 5 *A node element, N, is defined as*

$$N \in \mathcal{L}_{\mathcal{N}} \times (p \ \cup \ \mathcal{F}_A) \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}.$$

Given a node $N = (l_n,\ p,\ l,\ c_a,\ c_u)$, l_n denotes the node label, p is proposition of the node, the level is $l \geq 1$, c_a is total number of available resources, c_u is total number of used resources.

Definition 6 *A goal element, G, is defined as*

$$G \in \mathcal{L}_{\mathcal{G}} \times p \times \mathbb{N}.$$

Given a goal $g = (l_g, p, c)$, l_g is the goal label, p is the goal proposition and c is the goal count. We denote $g = (l_g, p, c)$ with $(l_g : (p, c))$.

Definition 7 *An implication action element, A_I, is defined as*

$$A_I \in \mathcal{L}_{\mathcal{A}} \times \mathcal{F}_A \times \mathbb{N} \times \mathcal{P}(\mathcal{L}_{\mathcal{N}} \times \mathbb{N}).$$

Given an implication action $a_i = (l_a,\ p_i,\ c,\ d_l)$, l_a is the action label, p_i is proposition of the action, c is number of applicable actions and d_l holds labels and counts for used resources for the precondition of the action where we denote each d_l element as (l_d, c_d). We must also note that, $\mathcal{P}(\mathcal{L}_{\mathcal{N}} \times \mathbb{N})$ denotes power set of $\mathcal{L}_{\mathcal{N}} \times \mathbb{N}$, which is $2^{\mathcal{L}_{\mathcal{N}} \times \mathbb{N}}$.

Definition 8 *A copy action element, A_C, is defined as*

$$A_C \in \mathcal{L}_{\mathcal{A}} \times (p \ \cup \ \mathcal{F}_A) \times \mathbb{N} \times \mathcal{P}(\mathcal{L}_{\mathcal{N}} \times \mathbb{N}).$$

Given a copy action $a_c = (l_a,\ p,\ c,\ d_l)$, l_a is the action label, p is proposition of the action, c is number of applicable actions and d_l holds labels and counts for used resources for the precondition of the action where we denote each d_l element as (l_d, c_d).

Definition 9 *An action element, A, is defined as*

$$A \in (A_I \cup A_C).$$

Definition 10 *A node to action link element, L_{NA}, holds a link from node to action defined as*

$$L_{NA} \in \mathcal{N} \times \mathcal{A},$$

where $\mathcal{N}$ is set of node elements and $\mathcal{A}$ is set of action elements.

Definition 11 *An action to node link element, L_{AN}, holds a link from action to node defined as*

$$L_{AN} \in \mathcal{A} \times \mathcal{N}.$$

Definition 12 *A node to goal link element, L_{NG}, holds a link from node to goal defined as*

$$L_{NG} \in \mathcal{N} \times \mathcal{G},$$

where $\mathcal{G}$ is set of goal elements.

Definition 13 *A projection notation, $\pi_x(E)$, returns the value of x in the given element E.*

Definition 14 *Given a LinGraph L, the initial level nodes of L, $\mathcal{N}_{\mathcal{I}}(L)$, are defined as*

$$\mathcal{N}_{\mathcal{I}}(L) := \{N \in \mathcal{N} \mid \pi_l(N) = 1\},$$

where $\mathcal{N}$ is set of node elements of L.

Definition 15 *Given a LinGraph L, the last level nodes of L, $\mathcal{N}_{\mathcal{L}}(L)$, are defined as*

$$\mathcal{N}_{\mathcal{L}}(L) := \{N \in \mathcal{N} \mid \forall N_2 \in \mathcal{N}, (\pi_l(N) \geq \pi_l(N_2))\},$$

where $\mathcal{N}$ is set of node elements of L.

Definition 16 *Given a LinGraph L, an initial constraint element c_i of L is defined as*

$$c_i \in \mathcal{N}_{\mathcal{I}}(L) \times \mathbb{N}.$$

Given an initial constraint $c_i = (N, \beta)$, the constraint formula is

$$\pi_{l_n}(N) = \beta,$$

where $\pi_{l_n}(N)$ denotes the label of the given initial node and the value β equals to the available number of the node N, $\pi_{c_a}(N)$.

Definition 17 *A sibling constraint element, c_s, with the arity$(c_s) = n$, is defined as*

$$c_s \in (N)^n.$$

Given a sibling constraint $c_s = (N_1, N_2, ..., N_n)$, the constraint formula is

$$\pi_{l_n}(N_1) = \pi_{l_n}(N_2) = ... = \pi_{l_n}(N_n).$$

Definition 18 *A dependency constraint element, c_d, with the arity$(c_d) = n$, is defined as*

$$c_d \in N \times (N)^n.$$

Given a dependency constraint $c_d = (N, (N_1, N_2, ..., N_n))$, the constraint formula is

$$\pi_{l_n}(N) = \pi_{l_n}(N_1) + \pi_{l_n}(N_2) + ... + \pi_{l_n}(N_n).$$

Definition 19 *A goal constraint element, c_g, with the arity$(c_g) = n$, is defined as*

$$c_g \in (N)^n \times \mathbb{N}.$$

Given a goal constraint $c_g = ((N_1, N_2, ..., N_n), \beta)$, the constraint formula is

$$\pi_{l_n}(N_1) + \pi_{l_n}(N_2) + ... + \pi_{l_n}(N_n) = \beta.$$

Definition 20 *Given that $\mathcal{C}_i$ is a set of initial constraints, $\mathcal{C}_s$ is a set of sibling constraints, $\mathcal{C}_d$ is a set of dependency constraints, $\mathcal{C}_g$ is a set of goal constraints, a set of constraints for checking goals, $\mathcal{C}_{cg}$, is defined as*

$$\mathcal{C}_{cg} \in (\mathcal{C}_i \cup \mathcal{C}_s \cup \mathcal{C}_d \cup \mathcal{C}_g).$$

Definition 21 *A partial initial constraint element for precondition check, c_{pi} is defined as*

$$c_{pi} \in N \times \mathbb{N}.$$

Given a partial initial constraint for precondition check $c_{pi} = (N, \beta)$, the constraint formula is

$$\pi_{l_n}(N) \leq \beta.$$

Definition 22 *A partial dependency constraint element for precondition check, c_{pd}, with the arity$(c_{pd}) = n$, is defined as*

$$c_{pd} \in (N)^n \times N.$$

Given a partial dependency constraint for precondition check $c_{pd} = ((N_1, N_2, ..., N_n), N)$, the constraint formula is

$$\pi_{l_n}(N_1) + \pi_{l_n}(N_2) + ... + \pi_{l_n}(N_n) \leq \pi_{l_n}(N).$$

Definition 23 *An action requirement constraint element for precondition check, c_{ar}, is defined as*

$$c_{ar} \in N \times \mathbb{N}.$$

Given an action requirement constraint $c_{ar} = (N, \beta)$, the constraint formula is

$$\pi_{l_n}(N) \geq \beta.$$

Definition 24 *An unused constraint element for precondition check, c_u, is defined as*

$$c_u \in N.$$

Given an action requirement constraint $c_u = N$, the constraint formula is

$$\pi_{l_n}(N) = 0.$$

Definition 25 *Given that $\mathcal{C}_{pi}$ is a set of partial initial constraints, $\mathcal{C}_s$ is a set of sibling constraints, $\mathcal{C}_{pd}$ is a set of partial dependency constraints, $\mathcal{C}_{ar}$ is a set of action requirement constraints, $\mathcal{C}_u$ is a set of unused constraints, a set of constraints for precondition of action check, $\mathcal{C}_{cp}$, is defined as*

$$\mathcal{C}_{cp} \in (\mathcal{C}_{pi} \cup \mathcal{C}_s \cup \mathcal{C}_{pd} \cup \mathcal{C}_{ar} \cup \mathcal{C}_u).$$

Definition 26 *Given that $\mathcal{L}_{\mathcal{N}}$ is a set of node labels, $\mathcal{L}_{\mathcal{A}}$ is a set of action labels, $\mathcal{L}_{\mathcal{G}}$ is a set of goal labels, $\mathcal{F}_G$ is a set of proposition languages, $\mathcal{N}$ is a set of Nodes, $\mathcal{G}$ is a set of Goals, $\mathcal{A}$ is a set of implication actions and copy actions, $\mathcal{L}_{\mathcal{NA}}$ is a set of links between nodes and actions, $\mathcal{L}_{\mathcal{AN}}$ is a set of links between actions and nodes, $\mathcal{L}_{\mathcal{NG}}$ is a set of links between nodes and goals, $\mathcal{C}_{cg}$ is a set of constraints for checking goals and $\mathcal{C}_{cp}$ is a set of constraints for checking preconditions, a planning graph for linear logic is formally represented as*

$$\mathit{LinGraph} := [\mathcal{L}_{\mathcal{N}}, \mathcal{L}_{\mathcal{A}}, \mathcal{L}_{\mathcal{G}}, \mathcal{F}_G, \mathcal{N}, \mathcal{G}, \mathcal{A}, \mathcal{L}_{\mathcal{NA}}, \mathcal{L}_{\mathcal{AN}}, \mathcal{L}_{\mathcal{NG}}, \mathcal{C}_{cg}, \mathcal{C}_{cp}],$$

where $\mathcal{L}_{\mathcal{NA}} \subseteq \mathcal{N} \times \mathcal{A}$, $\mathcal{L}_{\mathcal{NA}} \subseteq \mathcal{N} \times \mathcal{A}$ and $\mathcal{L}_{\mathcal{NG}} \subseteq \mathcal{N} \times \mathcal{G}$, such that having properties:

1. All levels of L are greater than or equal to 1,

$$\forall N \in \mathcal{N}, \pi_l(N) \geq 1.$$

2. Each node can have only one incoming action link, denoted as

$$\forall N \in \mathcal{N}, \exists A \in \mathcal{A}, \forall A' \in \mathcal{A} \ s.t. \ ((A, N) \in \mathcal{L}_{\mathcal{AN}}, [(A \neq A') \rightarrow (A', N) \notin \mathcal{L}_{\mathcal{AN}}], (\pi_l(N) \geq 2)).$$

3. An action's all predecessor nodes must be at the same level, denoted as

$$\forall A \in \mathcal{A}, \forall N, N' \in \mathcal{N}, ((N, A) \in \mathcal{L}_{\mathcal{NA}} \wedge (N', A) \in \mathcal{L}_{\mathcal{NA}} \rightarrow \pi_l(N) = \pi_l(N'))$$

4. An action's all successor nodes must be at the same level, denoted as

$$\forall A \in \mathcal{A}, \forall N, N' \in \mathcal{N}, ((A, N) \in \mathcal{L}_{\mathcal{AN}} \wedge (A, N') \in \mathcal{L}_{\mathcal{AN}} \rightarrow \pi_l(N) = \pi_l(N'))$$

Definition 27 *Given a LinGraph L, a path from $n_1 \in \mathcal{N}$ to $n_k \in \mathcal{N}$ for L, $P_{(n_1,n_k)}$, is a sequence of nodes and actions ending with a node, denoted as*

$$P_{(n_1,n_k)} = \langle n_1, a_1, n_2, a_2 ..., n_k \rangle,$$

s.t. $\forall i \leq k, n_i \in \mathcal{N}, a_i \in \mathcal{A}$.

Definition 28 *Given a LinGraph L, a path from $n_1 \in \mathcal{N}$ to $n_k \in \mathcal{N}$ for L is valid, denoted with $V(P_{(n_1,n_k)})$, if*

1. *There exists a node to action link connecting each node to the next action and there exists an action to node link connecting each action to the next node, such as*

$$\forall i < k, (n_i, a_i) \in \mathcal{L}_{\mathcal{N}\mathcal{A}}, (a_i, n_{i+1}) \in \mathcal{L}_{\mathcal{A}\mathcal{N}},$$

2. *Level of sequential nodes increases one by one, denoted as*

$$\forall i < k, \pi_l(n_{i+1}) = \pi_l(n_i) + 1,$$

where $\mathcal{L}_{\mathcal{N}\mathcal{A}}$ is the set of links from actions to nodes and $\mathcal{L}_{\mathcal{A}\mathcal{N}}$ is the set of links from nodes to actions.

Definition 29 *Given a LinGraph L, the predecessor for the node $N \in \mathcal{N}$ denoted with $pred(N)$, is defined as*

$$pred(N) := \{A \in \mathcal{A} \mid (A, N) \in \mathcal{L}_{\mathcal{A}\mathcal{N}}\},$$

where $\mathcal{N}$ is the set of node elements and $\mathcal{A}$ is the set of action elements for L.

Definition 30 *Given a LinGraph L, the set of predecessors for the action $A \in \mathcal{A}$ denoted with $pred(A)$, is defined as*

$$pred(A) := \{N \in \mathcal{N} \mid (N, A) \in \mathcal{L}_{\mathcal{N}\mathcal{A}}\},$$

where $\mathcal{N}$ is the set of node elements and $\mathcal{A}$ is the set of action elements for L.

Definition 31 *Given a LinGraph L, the set of successors for the node $N \in \mathcal{N}$ denoted with succ(N), is defined as*

$$succ(N) := \{A \in \mathcal{A} \mid (N, A) \in \mathcal{L}_{\mathcal{N}\mathcal{A}}\},$$

where $\mathcal{N}$ is the set of node elements and $\mathcal{A}$ is the set of action elements for L.

Definition 32 *Given a LinGraph L, the set of successors for the action $A \in \mathcal{A}$ denoted with succ(A), is defined as*

$$succ(A) := \{N \in \mathcal{N} \mid (A, N) \in \mathcal{L}_{\mathcal{A}\mathcal{N}}\}.$$

Definition 33 *Given a LinGraph*

$$L := [\mathcal{L}_{\mathcal{N}}, \mathcal{L}_{\mathcal{A}}, \mathcal{L}_{\mathcal{G}}, \mathcal{F}_G, \mathcal{N}, \mathcal{G}, \mathcal{A}, \mathcal{L}_{\mathcal{N}\mathcal{A}}, \mathcal{L}_{\mathcal{A}\mathcal{N}}, \mathcal{L}_{\mathcal{N}\mathcal{G}}, \mathcal{C}_{cg}, \mathcal{C}_{cp}],$$

as given in Def. 26, L is an empty LinGraph if

$$L := [\mathcal{L}_{\mathcal{N}}, \mathcal{L}_{\mathcal{A}}, \mathcal{L}_{\mathcal{G}}, \mathcal{F}_G, \emptyset, \emptyset, \emptyset, \mathcal{L}_{\mathcal{N}\mathcal{A}}, \mathcal{L}_{\mathcal{A}\mathcal{N}}, \mathcal{L}_{\mathcal{N}\mathcal{G}}, \mathcal{C}_{cg}, \mathcal{C}_{cp}].$$

Definition 34 *Given a LinGraph L, cardinality of outgoing links for the last level node $N \in \mathcal{N}_{\mathcal{L}}(L)$, $OL(N)$, is defined as,*

$$OL(N) := |\{G \ s.t.\ (N, G) \in \mathcal{L}_{\mathcal{N}\mathcal{G}}\}|.$$

Definition 35 *Given a LinGraph L, cardinality of incoming links for the goal node $G \in \mathcal{G}$, $IL(G)$, is defined as,*

$$IL(G) := |\{G \ s.t.\ (N, G) \in \mathcal{L}_{\mathcal{N}\mathcal{G}}\}|.$$

Definition 36 *Given a multiset M, multiplicity of an element $m \in M$, $Mul(M, m)$, is defined as,*

$$Mul(M, m) := m \in^n M,$$

where multiplicity n is a nonnegative integer. If $M = \{\}$ or $m \notin M$, then $n = 0$.

Definition 37 *Given a resource formula F_R, we decompose the given formula and return a multiset of all atomic propositions using the function $DecRes(F_R)$, defined as*

$$DecRes(F_R) := \begin{cases} \{a\} & \text{if } F_R = a \\ \{a\} \cup DecRes(F_{R_2}) & \text{if } F_R = a \otimes F_{R_2}, \end{cases}$$

where union operation is a multiset operation and a is an atomic proposition.

Definition 38 *Given a goal formula G, we extract goals and return a multiset of all goals using the function $ExtGoal(G)$, defined as*

$$ExtGoal(G) := \begin{cases} DecRes(F_R) & \text{if } G = F_R \\ ExtGoal(G_2) & \text{if } G = F_P \multimap G_2, \end{cases}$$

where union operation is a multiset operation, a is an atomic proposition and F_P is a program formula.

Definition 39 *Given a goal formula G, we decompose the given formula and return a multiset of all resources using the function $ExtRes(G)$, defined as*

$$ExtRes(G) := \begin{cases} \{\} & \text{if } G = F_R \\ \{a\} \cup ExtRes(G_2) & \text{if } G = a \multimap G_2 \\ \{a\} \cup DecRes(F_R) \cup ExtRes(G_2) & \text{if } G = (a \otimes F_R) \multimap G_2 \\ \{(F_{R_1} \multimap F_{R_2})\} \cup ExtRes(G_2) & \text{if } G = !(F_{R_1} \multimap F_{R_2}) \multimap G_2, \end{cases}$$

where union operation is a multiset operation and a is an atomic proposition.

Definition 40 *Given a multiset M and a proposition p, if p is an atomic proposition, $Cnt(M,p)$ returns the number of p in the given multiset M, otherwise 1, defined as*

$$Cnt(M,p) := \begin{cases} 1 & \text{if } p = F_{R_1} \multimap F_{R_2} \\ Mul(M,p) & \text{if } p = a, \end{cases}$$

where a is an atomic proposition.

Definition 41 *Given a formula F and its LinGraph L as given in Def. 26, Def. 14 and Def. 15, L is a* ***valid LinGraph*** *iff:*

1. *Number of resources for the initial level nodes are equal to the corresponding*

decomposed propositions, defined as

$$\forall N \in \mathcal{N}_{\mathcal{I}}(\mathcal{L}), (\pi_p(N) \neq (F_{R_1} \multimap F_{R_2}), \pi_p(N) \in^k ExtRes(F)) \rightarrow \pi_{c_a}(N) = k.$$

2. *Number of action resources for the initial level nodes are equal to infinite, defined as*

$$\forall N \in \mathcal{N}_{\mathcal{I}}(\mathcal{L}), (\pi_p(N) = (F_{R_1} \multimap F_{R_2})) \rightarrow \pi_{c_a}(N) = \infty.$$

3. *Each goal node (l_g, p, c) in $\mathcal{G}$ corresponds to aggregating the same type of decomposed goals together from the initial formula F, denoted as*

$$\forall G \in \mathcal{G}, \forall G' \in \mathcal{G} \ s.t. \ (\pi_p(G) \in^k ExtGoal(F), \pi_p(G) = \pi_p(G')) \rightarrow (k = c, G = G').$$

4. *Number of incoming links to each goal node (l_g, p, c) equals to the count of the goal node, defined as*

$$\forall G \in \mathcal{G}, IL(G) = \pi_c(G).$$

5. *Count used (c_u) values are zero for any nodes except last level nodes, denoted as*

$$\forall N \in \mathcal{N} \ s.t. \ (N \notin \mathcal{N}_{\mathcal{L}}(L) \rightarrow (\pi_{c_u}(N) = 0).$$

6. *There exists at least one valid path from each first level node to last level nodes, defined as*

   (a) $\forall N_I \in \mathcal{N}_{\mathcal{I}}(L), \exists N_L \in \mathcal{N}_{\mathcal{L}}(L) \ s.t. \ P_{(N_I, N_L)}$ *is a valid path,*

   (b) $\forall N_L \in \mathcal{N}_{\mathcal{L}}(L), \exists N_I \in \mathcal{N}_{\mathcal{I}}(L) \ s.t. \ P_{(N_I, N_L)}$ *is a valid path.*

7. *All initial constraints are satisfied,*

$$\forall c_i \in \mathcal{C}_i, \pi_{l_n}(\pi_N(c_i)) = \pi_\beta(c_i).$$

8. *All sibling constraints are satisfied,*

$$\forall c_s \in \mathcal{C}_s, \pi_{l_n}(N_1) = \pi_{l_n}(N_2) = ... = \pi_{l_n}(N_{arity(c_s)})$$

9. *All dependency constraints of checking goals are satisfied,*

$$\forall c_d \in \mathcal{C}_d, \pi_{l_n}(\pi_N(c_d)) = \sum_{i=1}^{arity(c_d)} (\pi_{l_n}(\pi_{N_i}(c_d))).$$

10. *All goal constraints are satisfied,*

$$\forall c_g \in \mathcal{C}_g, \sum_{i=1}^{arity(c_g)} (\pi_{l_n}(\pi_{N_i}(c_g))) = \pi_\beta(c_g)$$

11. *All goals are satisfied,*

    (a) *Number of links going to each goal node must be equal to the goal node's goal count.*

    $$\forall G \in \mathcal{G}, \pi_c(G) = IL(G)$$

    (b) *Number of outgoing links from each last level node must be equal to the number of used resources, denoted as*

    $$\forall N \in \mathcal{N}_{\mathcal{L}}(L), \pi_{c_u}(N) = OL(N)$$

12. *Each node created by a copy action has the same available number of resources with the predecessor node, denoted as*

$$\forall N \in \mathcal{N}, \forall Succ(N) \in \mathcal{A}_{\mathcal{C}}, (N \notin \mathcal{N}_{\mathcal{L}}(L) \rightarrow \pi_{c_a}(Succ(Succ(N))) = \pi_{c_a}(N).$$

13. *Available number of resources for created nodes of an action equals to a proportion including count of decomposed formulas, defined as*

$$\forall A \in \mathcal{A}_{\mathcal{I}}, \forall N \in succ(A) \ if \ \pi_{p_i}(A) = F_R \multimap F_{R_2} \ then \ \pi_{c_a}(N) = \pi_c(A) \cdot Cnt(DecRes(F_{R_2}), \pi_p(N)).$$

Bibliography

[1] T. Estlin, R. Castano, R. Anderson, D. Gaines, F. Fisher, and M. Judd, "Learning and planning for mars rover science," in *Proc. of the Int. J. Conf. on Artificial Intelligence*, 2003.

[2] M. Maurette, "Mars rover autonomous navigation," *Autonomous Robots*, vol. 14, no. 2-3, pp. 199–208, 2003.

[3] J. L. Bresina, A. K. Jónsson, P. H. Morris, and K. Rajan, "Activity planning for the mars exploration rovers," in *Proc. of the Int. Conf. on Automated Planning and Scheduling*, pp. 40–49, AAAI, 2005.

[4] T. A. Estlin, D. M. Gaines, C. Chouinard, R. Castao, B. J. Bornstein, M. Judd, I. A. D. Nesnas, and R. C. Anderson, "Increased mars rover autonomy using ai planning, scheduling and execution," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 4911–4918, 2007.

[5] J. P. Grotzinger, J. Crisp, A. R. Vasavada, R. C. Anderson, C. J. Baker, R. Barry, D. F. Blake, P. Conrad, K. S. Edgett, B. Ferdowski, *et al.*, "Mars science laboratory mission and science investigation," *Space science reviews*, vol. 170, no. 1-4, pp. 5–56, 2012.

[6] T. Lozano-Pérez, J. L. Jones, E. Mazer, and P. A. O'Donnell, *HANDEY: A Robot Task Planner.* MIT Press, 1992.

[7] M. Brady, J. M. Hollerbach, T. L. Johnson, M. T. Mason, and T. Lozano-Perez, *Robot Motion: Planning and Control.* Cambridge, MA, USA: MIT Press, 1983.

[8] T. Lozano-Perez, "Task planning," *Robot motion: planning and control*, pp. 463–489, 1982.

[9] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, second ed., 2003.

[10] J.-Y. Girard, "Linear logic," *Theoretical Computer Science*, vol. 50, no. 1, pp. 1–102, 1987.

[11] J.-Y. Girard, "A new constructive logic: classic logic," *Mathematical Structures in Computer Science*, vol. 1, pp. 255–296, 11 1991.

[12] L. Chrpa, "Linear logic in planning," in *Proc. of the The Int. Conf. on Automated Planning and Scheduling*, pp. 26–29, 2006.

[13] R. Reiter, "The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression," *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy*, vol. 27, pp. 359–380, 1991.

[14] J. Lee and R. Palla, "Reformulating the situation calculus and the event calculus in the general theory of stable models and in answer set programming," *Journal of Artificial Intelligence Research*, vol. 43, pp. 571–620, 2012.

[15] P. Kungas, "Linear logic programming for AI planning," M.Sc., Institute of Cybernetic at Tallinn Technical University, May 2002.

[16] E. Jacopin, "Classical AI planning as theorem proving: The case of a fragment of linear logic," in *AAAI Fall Symp. on Automated Deduction in Nonstandard Logics*, (Palo Alto, CA), pp. 62–66, 1993.

[17] M. Kanovich and J. Vauzeilles, "The classical AI planning problems in the mirror of horn linear logic: semantics, expressibility, complexity," *Mathematical Structures in Computer Science*, vol. 11, pp. 689–716, December 2001.

[18] C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. Pappas, "Symbolic planning and control of robot motion - grand challenges of robotics," *IEEE Robotics and Automation Magazine*, vol. 14, pp. 61–70, March 2007.

[19] J. S. Hodas and D. Miller, "Logic programming in a fragment of intuitionistic linear logic," *Information and Computation*, vol. 110, no. 2, pp. 327–365, 1994.

[20] U. Saranli, M. Buehler, and D. E. Koditschek, "RHex: A simple and highly mobile robot," *International Journal of Robotics Research*, vol. 20, pp. 616–631, July 2001.

[21] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, pp. 189–208, 1971.

[22] E. P. Pednault, "Formulating multiagent dynamic-world problems in the classical planning framework," in *Reasoning About Actions and Plans* (M. P. Georgeff and A. L. Lansky, eds.), pp. 47–82, Morgan Kaufmann, 1987.

[23] M. Gelfond and V. Lifschitz, "Action languages," *Electronic Transactions on Artificial Intelligence*, vol. 3, pp. 193–210, 1998.

[24] M. Ghallab, C. Knoblock, D. Wilkins, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. E. Smith, *et al.*, "PDDL-the planning domain definition language," 1998.

[25] D. S. Nau, "Current trends in automated planning," *AI Magazine*, vol. 28, no. 4, p. 43, 2007.

[26] D. S. Weld, "An introduction to least commitment planning.," *AI Magazine*, vol. 15, no. 4, pp. 27–61, 1994.

[27] D. McAllester and D. Rosenblitt, "Systematic nonlinear planning," in *Proc. of the National Conf. of the American Association for Artificial Intelligence (AAAI-91)*, pp. 634–639, 1991.

[28] J. S. Penberthy and D. S. Weld, "UCPOP: A sound, complete, partial order planner for adl," in *Proc. of the Int. Conf. on Knowledge Representation and Reasoning*, pp. 103–114, 1992.

[29] A. L. Blum and M. L. Furst, "Fast planning through planning graph analysis," *Artificial Intelligence*, vol. 90, no. 1, pp. 1636–1642, 1995.

[30] B. C. Gazen and C. A. Knoblock, “Combining the expressivity of UCPOP with the efficiency of GraphPlan,” in *Proc. of the European Conf. on Planning*, pp. 221–233, 1997.

[31] D. S. Weld, C. R. Anderson, and D. E. Smith, “Extending graphplan to handle uncertainty and sensing actions,” in *Proc. of the National Conf. of the American Association for Artificial Intelligence (AAAI-98)*, (Menlo Park, CA, USA), pp. 897–904, 1998.

[32] A. Blum and J. Langford, “Probabilistic planning in the graphplan framework,” in *Recent Advances in AI Planning* (S. Biundo and M. Fox, eds.), vol. 1809 of *Lecture Notes in Computer Science*, pp. 319–332, Springer Berlin Heidelberg, 2000.

[33] J. Fu, F. B. Bastani, V. Ng, I.-L. Yen, and Y. Zhang, “FIP: A fast planning-graph-based iterative planner,” in *Proc. of the IEEE Int. Conf. on Tools in Artificial Intelligence*, pp. 419–426, 2008.

[34] A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, “Weak, strong, and strong cyclic planning via symbolic model checking,” *Artificial Intelligence*, vol. 147, no. 1–2, pp. 35–84, 2003.

[35] H. Jiang, X. Hu, D. Zuo, and F.-Q. Song, “A novel extension to graphplan on a dynamic set of objects,” in *Proc. of the IEEE Int. Conf. on Machine Learning and Cybernetics*, vol. 4, pp. 1830–1834, 2008.

[36] A. Lopez and F. Bacchus, “Generalizing GraphPlan by formulating planning as a CSP,” in *Proc. of the Int. J. Conf. on Artificial Intelligence*, (San Francisco, CA, USA), pp. 954–960, Morgan Kaufmann Publishers Inc., 2003.

[37] H. Kautz and B. Selman, “Blackbox: A new approach to the application of theorem proving to problem solving,” in *AIPS98 Workshop on Planning as Combinatorial Search*, pp. 58–60, 1998.

[38] H. Kautz and B. Selman, “Unifying sat-based and graph-based planning,” in *Proc. of the Int. J. Conf. on Artificial Intelligence*, pp. 318–325, Morgan Kaufmann, 1999.

[39] H. A. Kautz and B. Selman, “Pushing the envelope: Planning, propositional logic and stochastic search,” in *Proc. of the National Conf. of the American Association for Artificial Intelligence (AAAI-96)*, vol. 2, pp. 1194–1201, 1996.

[40] H. Kautz, D. Mcallester, and B. Selman, “Encoding plans in propositional logic,” in *Proc. of the Int. Conf. on Knowledge Representation and Reasoning*, pp. 374–384, Morgan Kaufmann, 1996.

[41] H. A. Kautz, “Deconstructing planning as satisfiability,” in *Proc. of the National Conf. of the American Association for Artificial Intelligence (AAAI-06)*, (Boston, MA), 2006.

[42] H. Kautz and B. Selman, “Planning as satisfiability,” in *Proc. of the European Conf. on Artificial Intelligence*, pp. 359–363, Wiley, 1992.

[43] T. Bylander, “The computational complexity of propositional STRIPS planning,” *Artificial Intelligence*, vol. 69, no. 1-2, pp. 165–204, 1994.

[44] B. Bonet and H. Geffner, “Planning as heuristic search,” *Artificial Intelligence*, vol. 129, no. 1-2, pp. 5–33, 2001.

[45] B. Bonet, G. Loerincs, and H. Geffner, “A robust and fast action selection mechanism for planning,” in *Proc. of the National Conf. of the American Association for Artificial Intelligence (AAAI-97)*, pp. 714–719, MIT Press, 1997.

[46] B. Bonet and H. Geffner, “HSP heuristic search planner,” in *AIPS-98 Planning Competition*, (Pittsburgh, PA), 1998.

[47] J. Hoffmann and B. Nebel, “The FF planning system: Fast plan generation through heuristic search,” *Journal of Artificial Intelligence Research*, vol. 14, pp. 253–302, May 2001.

[48] J. Rintanen, K. Heljanko, and I. Niemelä, “Planning as satisfiability: parallel plans and algorithms for plan search,” *Artificial Intelligence*, vol. 170, no. 12, pp. 1031–1080, 2006.

[49] J. Rintanen, "Planning as satisfiability: Heuristics," *Artificial Intelligence*, vol. 193, pp. 45–86, 2012.

[50] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[51] I. Georgievski and M. Aiello, "HTN planning: Overview, comparison, and beyond," *Artificial Intelligence*, vol. 222, pp. 124–156, 2015.

[52] J. McCarthy, "Situations, actions, and causal laws," Tech. Rep. AD0785031, Stanford University, Dept. of Computer Science, 1963.

[53] J. McCarthy, "Some philosophical problems from the stand-point of artificial intelligence," *Machine Intelligence*, vol. 4, pp. 463–502, 1969.

[54] L. Dixon, A. Smaill, and T. Tsang, "Plans, actions and dialogue using linear logic," *Journal of Logic, Language and Information*, vol. 18, pp. 251–289, 2009.

[55] Y. Dimopoulos, B. Nebel, and J. Koehler, "Encoding planning problems in nonmonotonic logic programs," in *Recent Advances in AI Planning* (S. Steel and R. Alami, eds.), vol. 1348 of *Lecture Notes in Computer Science*, pp. 169–181, Springer Berlin Heidelberg, 1997.

[56] V. Lifschitz, "Answer set planning," in *Logic Programming and Nonmonotonic Reasoning* (M. Gelfond, N. Leone, and G. Pfeifer, eds.), vol. 1730 of *Lecture Notes in Computer Science*, pp. 373–374, Springer Berlin Heidelberg, 1999.

[57] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres, "Answer set planning under action costs," *Journal of Artificial Intelligence Research*, vol. 19, pp. 25–71, 2003.

[58] M. Masseron, C. Tollu, and J. Vauzeilles, "Generating plans in linear logic I. Actions as proofs.," *Theoretical Computer Science*, vol. 113, no. 2, pp. 349–370, 1993.

[59] M. Masseron, "Generating plans in linear logic II. A geometry of conjunctive actions," *Theoretical Computer Science*, vol. 113, no. 2, pp. 371–375, 1993.

[60] O. Kahramanogullari, "Towards planning as concurrency," in *Artificial Intelligence and Applications*, pp. 387–393, 2005.

[61] O. Kahramanoğulları, "On linear logic planning and concurrency," in *Language and Automata Theory and Applications*, pp. 250–262, Springer, 2008.

[62] N. Mart-Oliet and J. Meseguer, "From petri nets to linear logic," in *Category Theory and Computer Science* (D. Pitt, D. Rydeheard, P. Dybjer, A. Pitts, and A. Poign, eds.), vol. 389 of *Lecture Notes in Computer Science*, pp. 313–340, Springer Berlin Heidelberg, 1989.

[63] F. Silva, M. A. Castilho, and L. A. Künzle, "Petriplan: a new algorithm for plan generation (preliminary report)," in *Advances in Artificial Intelligence*, pp. 86–95, Springer, 2000.

[64] S. L. Hickmott, J. Rintanen, S. Thiébaux, and L. B. White, "Planning via petri net unfolding," in *Proc. of the Int. J. Conf. on Artificial Intelligence*, vol. 7, pp. 1904–1911, 2007.

[65] H. Costelha and P. Lima, "Robot task plan representation by petri nets: modelling, identification, analysis and execution," *Autonomous Robots*, vol. 33, no. 4, pp. 337–360, 2012.

[66] D. Berenson, S. Srinivasa, D. Ferguson, and J. Kuffner, "Manipulation planning on constraint manifolds," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 625–632, 2009.

[67] N. Dantam, P. Kolhe, and M. Stilman, "The motion grammar for physical human-robot games," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 2011.

[68] E. Klavins, "Automatic compilation of concurrent hybrid factories from product assembly specifications," in *Hybrid Systems: Computation and Control* (N. Lynch and B. Krogh, eds.), vol. 1790 of *Lecture Notes in Computer Science*, pp. 174–187, Springer, 2000.

[69] T. Lutovac and J. Harland, "Detecting loops during proof search in propositional affine logic," *Journal of Logic and Computation*, vol. 16, pp. 61–133, February 2006.

[70] H. J. Levesque, "Planning with loops," in *Proc. of the Int. J. Conf. on Artificial Intelligence*, pp. 509–515, 2005.

[71] Y. Hu and H. J. Levesque, "Planning with loops: Some new results," in *Proc. of the Int. Conf. on Automated Planning and Scheduling*, 2009.

[72] S. Thompson, *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[73] G. Gentzen and M. Szabo, *The collected papers of Gerhard Gentzen.* Studies in logic and the foundations of mathematics, Amsterdam: North-Holland, 1969.

[74] N. Gupta and D. S. Nau, "On the complexity of blocks-world planning," *Artificial Intelligence*, vol. 56, no. 2-3, pp. 223–254, 1992.

[75] J.-M. Andreoli, "Logic programming with focusing proofs in linear logic," *Journal of Logic and Computation*, vol. 2, pp. 297–347, 1992.

[76] I. Cervesato, J. S. Hodas, and F. Pfenning, "Efficient resource management for linear logic proof search," *Theoretical Computer Science*, vol. 232, pp. 133–163, Feb. 2000.

[77] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, "Uniform proofs as a foundation for logic programming," *Annals of Pure and Applied Logic*, vol. 51, no. 1–2, pp. 125–157, 1991.

[78] T. Tammet, "Resolution, inverse method and the sequent calculus.," in *Kurt Gödel Colloquium on Computational Logic and Proof Theory*, vol. 1289 of *Lecture Notes in Computer Science*, pp. 65–83, Springer, 1997.

[79] K. Chaudhuri and F. Pfenning, "Focusing the inverse method for linear logic.," in *International Workshop on Computer Science Logic*, vol. 3634 of *Lecture Notes in Computer Science*, pp. 200–215, Springer, 2005.

[80] C. C. L. nán, “cvBlob.” http://cvblob.googlecode.com.

[81] F. Pfenning, “Lecture notes on linear logic,” tech. rep., Carnegie Mellon University, 2002.

[82] P. Kungas, “Linear logic for domain-independent AI planning,” in *Proc. of Doct. Cons. at the Int. Conf. on Automated Planning and Scheduling*, (Trento, Italy), pp. 68–72, 2003.

[83] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*. Cambridge University Press, 1989.

[84] I. P. Gent, C. Jefferson, and I. Miguel, “MINION: A fast, scalable, constraint solver,” in *Proc. of the European Conf. on Artificial Intelligence*, (Riva Del Garda, Italy), pp. 98–102, 2006.

[85] A. Torralba, V. Alcazar, D. Borrajo, P. Kissmann, and S. Edelkamp, “Symba: A symbolic bidirectional a planner,” in *Eighth Int. Planning Competition (IPC 2014), Deterministic Part*, pp. 105–108, 2014.

[86] M. Vallati, L. Chrpa, M. Grzes, T. L. McCluskey, M. Roberts, and S. Sanner, “The 2014 international planning competition: Progress and trends,” *AI Magazine*, vol. 36, no. 3, pp. 90–98, 2015.

[87] Y. Alkhazraji, M. Katz, R. Matmüller, F. Pommerening, A. Shleyfman, and M. Wehrle, “Metis: Arming fast downward with pruning and incremental computation,” in *Eighth Int. Planning Competition, Deterministic Part*, pp. 88–92, 2014.

[88] P. Eyerich, T. Keller, J. Aldinger, and C. Dornhege, “Preferring preferred operators in temporal fast downward,” in *Eighth Int. Planning Competition (IPC 2014), Deterministic Part*, pp. 88–92, 2014.

[89] S. Kortik and U. Saranli, “Linear planning logic: An efficient language and theorem prover for robotic task planning,” in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 3764–3770, 2014.

[90] A. Tzouvaras, “The linear logic of multisets,” *Logic Journal of the IGPL*, vol. 6, no. 6, pp. 901–916, 1998.

[91] I. Cervesato and A. Scedrov, "Relating state-based and process-based concurrency through linear logic," *Information and Computation*, vol. 207, no. 10, pp. 1044 – 1077, 2009.

[92] M. I. Kanovich, "Linear logic as a logic of computations," *Annals of Pure and Applied Logic*, vol. 67, no. 1, pp. 183 – 212, 1994.

[93] I. Cervesato and E. S. L. Lam, "Modular multiset rewriting," in *Proc. of the Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, pp. 515–531, 2015.

[94] O. Kahramanogullari, "On linear logic planning and concurrency," *Information and Computation*, vol. 207, no. 11, pp. 1229–1258, 2009.

[95] U. Saranli and F. Pfenning, "Using constrained intuitionistic linear logic for hybrid robotic planning problems," in *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pp. 3705–3710, 2007.