EXPERIMENTS IN INTEGRATING CONSTRAINTS WITH LOGICAL REASONING FOR ROBOTIC PLANNING WITHIN THE TWELF LOGICAL FRAMEWORK AND THE PROLOG LANGUAGE

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Mert Duatepe

September, 2008

ABSTRACT

EXPERIMENTS IN INTEGRATING CONSTRAINTS WITH LOGICAL REASONING FOR ROBOTIC PLANNING WITHIN THE TWELF LOGICAL FRAMEWORK AND THE PROLOG LANGUAGE

Mert Duatepe
M.S. in Computer Engineering
Supervisor: Asst. Prof. Dr. Uluç Saranlı
September, 2008

The underlying domain of various application areas, especially real-time systems and robotic applications, generally includes a combination of both discrete and continuous properties. In robotic applications, a large amount of different approaches are introduced to solve either a discrete planning or control theoretic problem. Only a few methods exist to solve the combination of them. Moreover, these methods fail to ensure a uniform treatment of both aspects of the domain. Therefore, there is need for a uniform framework to represent and solve such problems. A new formalism, the Constrained Intuitionistic Linear Logic (CILL), combines continuous constraint solvers with linear logic. Linear logic has a great property to handle hypotheses as resources, easily solving state transition problems. On the other hand, constraint solvers deal well with continuous problems defined as constraints. Both properties of CILL gives us powerful ways to express and reason about the robotics domain. In this thesis, we focus on the implementation of CILL in both the Twelf Logical Framework and Prolog. The reader of this thesis can find answers of why classical aspects are not proper for the robotics domain, what advantages one can gain from intuitionism and linearity, how one can define a simple robotic domain in a logical formalism, how a proof in logical system corresponds to a plan in the robotic domain, what the advantages and disadvantages of logical frameworks and Prolog have and how the implementation of CILL can or cannot be done using both Twelf Logical Framework and Prolog.

Keywords: constrained intuitionistic linear logic, automated theorem proving, intuitionism, planning in robotics, logical framework, prolog.

ÖZET

ROBOTİK PLANLAMAYI GERÇEKLEŞTİRMEK İÇİN MANTIKSAL MUHAKEME İLE KISITLAMALARIN TWELF MANTIKSAL ÇATISI VE PROLOG DİLİ İÇERİSİNDE BİRLEŞTİRİLMESİNE YÖNELİK DENEMELER

Mert Duatepe
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Yrd. Doç. Dr. Uluç Saranlı
Eylül, 2008

Birçok uygulama alanı, özellikle de gerçek zamanlı sistemler ve robotik uygulamalar, çoğunlukla hem kesikli hem de sürekli özelliklerini içerirler. Robotik uygulamalarda, kesikli planlama ya da kontrol teorisi problemlerini çözmek için birçok farklı yaklaşım ortaya atılmıştır. Ayrıca, bunların kombinasyonunu barındıran problemleri çözmek için farklı yöntemler mevcuttur. Fakat, bu yöntemler uygulama alanının hem kesikli hem sürekli sorunlarını çözmek için bütün bir sistem oluşturamazlar. Bu yüzden, bu tür problemleri betimeleyebileceğimiz ve çözebileceğimiz bütün bir sisteme önemli ölçüde ihtiyaç vardır. Yeni bir biçim olan Kısıtlı Sezgisel Doğrusal Mantık(KSDM), sürekli kısıtlama çözücülerle doğrusal mantığı birleştirir. Doğrusal mantık, varsayımları kaynak olarak kullanarak durum geçiş problemlerini rahatlıkla çözebilecek çok önemli bir özelliğe sahiptir. Başka bir taraftan, kısıtlama çözücü kısıtlama olarak tanımlanmış sürekli problemleri çözecektir. KSDM'nin bu iki özelliği robotik alan uygulamalarının tanımlanmasını ve çözümlenmesini güçlü bir şekilde yapacaktır. Bu tezde, KSDM'nin hem Twelf Mantıksal çatısı hem de Prolog kullanarak gerçekleştirilmesinin üzerine odaklanılmıştır. Bu tezi okuyan okuyucu, klasik görüşün robotik alan uygulamaları için hangi eksiklikleri içerdiği, sezgiselcilikten ve doğrusalcılıktan nasıl kazançlar elde edileceği, bir mantıksal biçimin içinde basit bir robotik alan uygulamasının nasıl ifade edileceği, mantıksal sistemdeki bir kanıtın nasıl robotik alanda bir plana karşılık geleceği, mantıksal çatı ve prologun hangi artıları ve eksileri olduğu ile KSDM'nin gerçekleştiriminin hem mantıksal çatı hem de prolog çerçevesinde nasıl olacağı gibi önemli sorulara yanıt bulacaktır.

Anahtar sözcükler: kısıtlı sezgisel doğrusal mantık, özdevinimli kuram ispatlama, sezgicilik, robotikte planlama, mantıksal çatı, prolog.

Acknowledgement

My most sincere thanks go to my advisor, Uluç Saranlı, for giving me encouragement and for being patient and supportive. He always shows great determination on me. He introduced me the field of logical systems and robotic planning and I believe that this work would not have been possible without the technical and moral support of him.

I would like to thank my thesis committee, Varol Akman and Pınar Şenkul for their participation and giving useful advices to me. Furthermore, I would like to thank Ferda Nur Alpaslan for her vital advices.

I am also very grateful to Frank Pfenning for his invaluable class notes and assignments, Karl Crary for his great case studies and Jens Otten for his support and implementation of ileanSeP.

I must also thank all of my colleagues in ASELSAN for their moral support. I am also very grateful to my special friends, Murat Soyupak, Orkide Başkaya, Dikmen Işık, Hüseyin Kaval and Seval Karslı for their permanent support and help.

My debt to my parents, Gül and Mustafa, my sister, Meltem, and my aunt, Mevlüde, is immeasurable. They have always offered me their unconditional support in all of my endeavors. They instilled in me the value of hard work and a job well done, and gave me the confidence in my own abilities that made this thesis possible. Especially, my special thanks go to my mom. She is deserving of much of the credit for this thesis. Her boundless love and support have been unfailing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Today, robots are used in a large variety areas, such as military, space researches, factories and even homes. Since they are doing jobs that are impractical and time-consuming for people, they make human's life easier. The problem is how a robot can understand the features of a job and an environment while doing its job. The answer is quite simple: Declaring rules that the robot has to obey. For this purpose, the robot has to know its own capabilities and characteristics. For instance, the robot Phoenix [48] in Mars has capabilities of landing onto the Mars surface, walking on an unknown environment, decomposing soil and taking photos of the environment. Moreover, the Spike system [25] of Hubble telescope has a capability of making ground and orbiting telescope scheduling under several constraints.

Since a robot will have to make a plan according to the aforementioned rules, these rules have to be declared by taking into account of its behaviours and characteristics. Furthermore, this plan should correspond to action sequences and strategies. For example, if we have an environment that has some obstacles and a robot has a capability of moving only forward and right directions, then one has to find an obstacle-free shortest path under these conditions. This is called path planning in robotics. Moreover, if we have a robot arm that has three degrees of freedom, with a given robot arm position one can reach a requested robot arm position under the conditions of torque applied onto each degree of freedom and angles between each degree of freedom.

All of these kind of different planning problems which do not include continuous aspects can be reduced into one that can take three inputs. These are a description of the initial state of the world, a description of the desired goal, and a set of possible actions. A planner can find output action sequences that directly corresponds to a plan starting from initial state to the desired goal. One of the known planners in artificial intelligence is STRIPS [14] which is a formal language to express automated planning instances. However, the structure of this language fits only state-changing problems. Nevertheless, robotic planning problems has to deal with continuous problems of an environment. For example, in Blocks World problem one has to consider the problem of properly placing a block onto another block without falling any blocks, while in robot arm problem, one has to ponder torques applied on each degree of freedom.

In robotics and artificial intelligence, one of the main goals has been to build systems that are capable of finding autonomous plans according to given specifications. Several researchers have been trying to address planning problems in artificial intelligence. Techniques such as probabilistic planning [3], conditional planning [39] or utility planning [6] have been introduced. Some other methods degrade planning problems into constraint satisfaction problem in propositional logic [26, 11] while others find planning by applying model checking techniques [37, 7]. Moreover, some techniques represent planning problems in temporal logic [2] and others utilize heuristic methods [4]. Another kind of planning techniques try to represent plans with description logics and ontologies [15, 30].

In the context of robotics, while some discrete planning perspectives based on action sequences employs techniques mentioned before, others from control theory use specific methods for application domain [8, 13, 41, 51]. This is because current methods in artificial intelligence have been specialized for finding action sequences for discrete domains and thus fail to find plans for control theoretic robotic problems. Therefore, it is essential to build a uniform framework for the representation and solution of robotic planning problems including both discrete and continuous aspects of the underlying domain. Constrained Intuitionistic Linear Logic (CILL) [45] proposes a new formalism to represent and reason effectively for application domains that have both discrete and continuous capabilities.

Explaining each word in CILL will make clear for a reader to understand whole meaning of it. Logic gives a great expressive power to represent and reason with an application domain in terms of logical syntax and semantics because of two major properties. First, it is a formal and consistent language and thus planning problems for different applications can be represented in a consistent way. Second, finding plans for applications can be reduced into checking logical truth or provability of logical expressions. This is why several previous approaches have selected logic to represent planning problems. Linearity in logic [16, 17, 36] directly avoids the frame problem without any additional representational overhead and is explained in Chapter 4 in detail. Additional representational overhead is not required because linear logic treats hypotheses as consumable resources. Therefore, linear logic provides a natural way to deal with the state changing problem, frequently seen as one of the major problems in the field of planning. Intuitionism has been fairly discussed by philosophers and mathematicians since it was first suggested. In intuitionistic logic, proofs are described as true only when they are verified. Moreover, its nature ensures that each proof in intuitionistic logic directly corresponds to a program. Integration between linear logic and intuitionistic logic is important for planning problems because intuitionistic linear logic makes possible to find solution for planning problems in terms of programs without the additional requirement of dealing with the frame problem. However, it is still not possible to find a clean way to overcome robotic problems that include continuous properties. Combining the efficiency of domain specific constraint solvers with the expressive power of linear logic gives us a great feature to represent and reason about both discrete and continuous aspects of robotic behaviour in terms of powerful syntax of Constrained Intuitionistic Linear Logic [45]. Doing so allows us to express constraints of application domain within the logical formalism.

CILL offers a uniform formalism to find plans in robotic applications by combining both constraint solver and linear logic. The constraint solvers within linear logic solves mathematical equality or inequality problems and linear logic tries to find plans by using decision trees in logic. The real goal of this thesis is to automate decision procedures of CILL by using Twelf Logical Framework and Prolog language. However, as it is explained in last chapters, implementing CILL in both Twelf Logical Framework and Prolog is problematic. Therefore, we give theoretical way of finding plans in CILL, and instead we implement Constrained Intuitionistic Logic within both Twelf Logical Framework and Prolog language. Moreover, we show a basic robotic application for finding plans by using CIL in Prolog. We also give comparison of advantages and disadvantages of handling constraints in Twelf Logical Framework and Prolog.

In the second chapter, we will give some fundamental information about what we need for reading rest of the chapters. In Chapter 3, we will discuss the new example domain of Blocks World that has both discrete and continuous aspects. In the fourth chapter, we will give the formal definition of Intuitionistic Linear Logic (ILL) and its connectives and the last section of this chapter will show an extension of ILL with constraints and we will encode previously constructed Blocks World domain within CILL. In the following chapter, we give basic information about Twelf Logical Framework and Prolog Language. Chapter 6 gives major reasons why CILL could not be implemented within Twelf Logical Framework and instead Constrained Intuitionistic Logic (CIL) is implemented and some examples will be given. Same chapter shows Prolog implementation of CIL and then, we will finish up with future work and conclusion in last chapter.

Chapter 2

Background

2.1 Logic

Logic comes from the Greek word *logos* which means thought, idea, reason or principle. Logic studies the laws of valid inferences that means the act or process to derive a true conclusion from the base knowledge. The formal definition for logic is the study of the way of reasoning, deduction from hypotheses, and demonstration. In mathematics, logic is concerned with providing symbolic models of acceptable reasoning [12]. However, in philosophy, logic is a more general concept that it is concerned with human thought and ways to analyse them.

Logic has been a branch of philosophy and mathematics for a long time. Initially, logic was studied in philosophy. The usage of logic under the foundations of mathematics was much later than studies in philosophy and this new branch is referred to as formal logic or symbolic logic in the context of mathematics. In artificial intelligence, there are some applications of logic, such as a knowledge representation formalism and method of reasoning, and a programming language [29].

2.1.1 Propositional Logic

Propositional logic is the branch of logic that studies ways of combining propositions or statements to form more complicated propositions or statements [20, 43]. A *statement* can be defined as a declarative sentence that can either be true or false. The following is an example statement:

Mustafa Kemal Atatürk is the founder of the Turkish Republic.

The terms *proposition* and *statement* are generally used interchangeably. However, a proposition sometimes refers to two different statements that have the same meaning. For instance, “It is raining” and “Il pleut” are two different statements with same meaning and can be described with the same proposition. Since the distinction of these two terms is a matter of philosophy, these two terms are used interchangeably throughout this thesis. The upper-case letters, P, Q, R, ..., are mostly used as symbols for simple propositions. Joining two propositions with the words “and” and “or” is one common way of combining propositions.

Propositions will assume an important role in describing basic concepts and states of robotic application domains.

2.1.2 Predicate Logic

Predicate logic (also known as *first-order logic*) is an extension of propositional logic in which formulas contain variables that can be quantified [49, 5]. Two common quantifiers are the existential $\exists$ and universal $\forall$ quantifiers. Variables can be elements in a universe, as well as relations over the universe.

A variable frequently refers to a parameter within a proposition. Moreover, it creates a connection between propositions. In terms of robotic environments, one can define actions using quantifiers and variables such as the examples presented in Chapter 3.

2.1.3 Classical and Intuitionistic Logic

Before describing the differences between classical and intuitionistic logic, it would be better to give different viewpoints that correspond to these logical formalisms. From a classical point of view, each statement has to be true or false independent from whether the knowledge about such a statement is established or not and all quantifiers are assumed to range over a well-defined domain [1]. For instance, the statement that there are no odd perfect numbers is one of the famous conjectures in mathematics. This statement has neither been proved nor disproved, but classically speaking, it should be either true or false.

In the constructive perspective (the words constructive and intuitionistic are used interchangeably), a statement is asserted as true only when it has been verified to be true, and a statement is asserted as false only when it has been verified to be false. From a constructive viewpoint, it could not be asserted that the example about perfect numbers is true or false, since this conjecture could not be verified.

Formally, true propositions that are provable in classical logic, such as $A \vee \neg A$ called the excluded middle are not derivable in intuitionistic logic. In addition to this, intuitionistic logic rejects proof by contradiction because we need to verify the proposition itself rather than verifying its negation. Moreover, the double negation rule is also not derivable. The most important feature for intuitionistic logic is its restriction to allow only a single formula in the conclusion. However, classical logic allows multiple formula conclusions. These feature will be described in later sections in detail.

2.2 Proof Theory

Proof theory is the study and representation of proofs as formal mathematical objects, simplifying their analysis by mathematical techniques. Proofs are presented as data structures such as lists or trees, constructed according to the axioms and rules of a logical system [18].

Although there were some studies of famous mathematicians and logicians such as Gottlob Frege, Bertrand Russell [31] and Giuseppe Peano [50] who were advanced in formalising proofs in mathematical theory, the real story was started by David Hilbert. Contemporary proof theory was established by Hilbert who initiated his program in the foundations of mathematics. However, Gödel's work on proof theory then refuted this program. His incompleteness theorems showed that Hilbert's program was naive and simply a failure [9].

In parallel with the proof theoretic work of Gödel, Gerhard Gentzen introduced the new concept known as structural proof theory [18]. In short years, Gentzen introduced the formalisms of natural deduction and sequent calculus. Moreover, he made fundamental advances in the formalization of intuitionistic logic and introduced the important idea of analytic proofs.

Two of three types of proof calculus which are natural deduction calculus and sequent calculus will be described in the following sections, however the other type of proof calculus namely the Hilbert system, will be skipped since it is outside the scope of this thesis.

2.2.1 Natural Deduction

Gentzen planned and constructed a system called Natural Deduction to formalize proofs in terms of mathematical reasoning after dissatisfaction with Hilbert's system in capturing mathematical reasoning. In this paper, he said that he intended to construct a formalism which would be as close as possible to actual reasoning. The system of Gentzen's approach was devised by Prawitz [38]. In natural deduction, valid deductions are described through inference rules only. Furthermore, the meaning of the logical quantifiers and connectives is also described in terms of their inference rules. Here, the inference rule refers to a function from sets of formulae to formulae. The formulae in the argument are called *premises* and the return value is called *conclusion*. In other words, it forms a relation between premises and a conclusion, where the conclusion is said to be *derivable* from the premises. If the premise is empty, then the conclusion is said to be an axiom. Inference rules are given in the following standard form:

$$\frac{Premise1 \qquad Premise2}{Conclusion}\; Name$$

This expression states that whenever the given premises are obtained, the specified conclusion can be derived as well.

The basic notion in natural deduction is a *judgement* based on an *evidence*. In our normal life, we make judgements based on some evidence. For example, we make the judgement "*It is sunny*" based on our visual evidence or we make the judgement "*A implies B*" after an evidence of a derivation. In natural deduction, a judgement is something that is knowable or it is an object of knowledge. Moreover, the fundamental judgement used within natural deduction is "*A true*".

Before giving the details of natural deduction, it will be good to give some general notation:

- $\vdash$ known as the turnstile, separates assumptions on the left from propositions on the right.
- t denotes an arbitrary term where a term can be built up from *variables* x, y, *etc. function symbols* f, g, *etc.* and *parameters* a, b, *etc.*:

$$\textit{Terms } t := \quad x \mid a \mid f(t_1, ..., t_n)$$

- A and B denote propositions of a logic where P, Q, *etc.* corresponds to *predicate symbols* and a proposition can be defined as follows:
- Ψ, Γ and Δ are finite sequences of propositions, called contexts. On the left of the $\vdash$, sequence of propositions is considered conjunctively. The context cannot be on the right side of the $\vdash$ in intuitionistic logic; but in classical

$$\begin{array}{rl} \textit{Propositions}\ A := & P(t_1, ..., t_n) \mid A_1 \wedge A_2 \mid A_1 \supset A_2 \mid A_1 \vee A_2 \\ & \mid \neg A \mid \top \mid \bot \mid \forall x.A \mid \exists x.A \end{array}$$

logic, contexts can be placed on the right side. When a context is in the right side, the sequence of propositions is considered disjunctively.

- $A[t]$ denotes a proposition A, in which some occurrences of a term t are of interest.
- $A[s/t]$ denotes the proposition obtained by substituting the term s for all occurrences of t in $A[t]$.
- a variable is said to occur free within a formula if its only occurrences in the formula are not within the scope of quantifiers $\forall$ or $\exists$.

In natural deduction, each logical connective and quantifier is defined by its *introduction rule(s)* which specify how to infer that a conjunction, disjunction, etc. is true. The *elimination rule* tells what other truths can be deduced from the truth of a conjunction, disjunction, etc. Introduction and elimination rules must ensure that the rules are meaningful and the overall system can capture mathematical reasoning. We need to look at some properties to guarantee that our rules are meaningful.

If we introduce a connective and then immediately eliminate it, we should be able to find an other derivation of the conclusion without using the connective. This property is called as *local soundness* and if this property fails, the elimination rules are said to be too strong: they allow us to conclude more than we should be able to know.

We can eliminate a connective in a way that we can reconstruct it by an introduction rule. This property is called as *local completeness* and if this property fails, the elimination rules are too weak: they do not allow us to conclude everything we should be able to know.

One of the important principles of natural deduction is that each connective should be defined only in terms of inference rules without referring to other logical connectives or quantifiers. This property is called as *orthogonality* of the connectives. It means that we can understand a logical system as a whole by understanding each connective separately. It also allows us to extend the logical system directly without changing other inference rules.

Let us give an example inference rules of conjunction for propositional logic to clarify the concepts in natural deduction:

$$\frac{A\ \mathit{true} \qquad B\ \mathit{true}}{A \wedge B\ \mathit{true}}\ \wedge\mathtt{I}$$

This is an example of introduction of conjunction rule. It asserts that if we have the judgement "*A true*" and "*B true*", we can derive and introduce the judgement "$A \wedge B$ *true*" in the conclusion.

The elimination rule for conjunction will be written such that:

$$\frac{A \wedge B\ \mathit{true}}{A\ \mathit{true}}\ \wedge\mathtt{E}_L \qquad\qquad \frac{A \wedge B\ \mathit{true}}{B\ \mathit{true}}\ \wedge\mathtt{E}_R$$

In the above example, we say that we can derive the judgements *A true* and *B true* separately with the elimination of the judgement $A \wedge B$ *true*.

The judgement "*A true*" alone is not very suitable to introduce all logical connectives and quantifiers. We thus need two additional judgements. One of them is hypothetical judgement to introduce implication connective and the other one is the parametric judgement to apply for quantifiers.

The hypothetical judgement has a form that encodes "J_2 under hypothesis J_1". We consider this judgement evident if we make the judgement J_2 once provided with evidence for J_1.

The parametric judgement has the form "*J for any a*". We make this judgement if we make the judgement $[O/a]J$ for arbitrary objects O.

Natural deduction enables us to formalize a specific logic via inference rules of each connective. In addition to this, valid deductions are described through inference rules and proof tree can be constructed with these deductions. Let us look at the example derivation of $(A \wedge B) \supset (B \wedge A)$ in natural deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B}\,u}{B}\,\wedge\mathtt{E}_R \qquad \dfrac{\dfrac{}{A \wedge B}\,u}{A}\,\wedge\mathtt{E}_L}{B \wedge A}\,\wedge\mathtt{I}}{(A \wedge B) \supset (B \wedge A)}\,\supset\mathtt{I}^u$$

The first step of this derivation starts at the bottom with $(A \wedge B) \supset (B \wedge A)$:

$$\dfrac{}{(A \wedge B) \supset (B \wedge A)}$$

In the second step, we need the assumption $A \wedge B$ to satisfy the conclusion $(A \wedge B) \supset (B \wedge A)$ with implication introduction:

$$\dfrac{\begin{array}{c}\dfrac{}{A \wedge B}\,u \\ \vdots \\ B \wedge A\end{array}}{(A \wedge B) \supset (B \wedge A)}\,\supset\mathtt{I}^u$$

The implication introduction generates a gap between two premises. Filling the gap is crucial part to find the proper derivation for a rule. In other words, the purpose for finding derivation is to meet axioms and the second premise of implication introduction in some way. Thus, the third step tries to connect axiom $A \wedge B$ with conclusion $B \wedge A$. Consequently, the third step uses the conjunction introduction to introduce $B \wedge A$:

$$\dfrac{\dfrac{\begin{array}{cc} \dfrac{}{A \wedge B}\,u & \dfrac{}{A \wedge B}\,u \\ \vdots & \vdots \\ B & A \end{array}}{B \wedge A}\,\wedge\mathtt{I}}{(A \wedge B) \supset (B \wedge A)}\,\supset\mathtt{I}^u$$

We have to eliminate axiom $A \wedge B$ twice in order to find A and B separately. Thus, the last step fills the gap and shows the proper derivation for $(A \wedge B) \supset (B \wedge A)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B}\,u}{B}\,\wedge\mathtt{E}_R \quad \dfrac{\dfrac{}{A \wedge B}\,u}{A}\,\wedge\mathtt{E}_L}{B \wedge A}\,\wedge\mathtt{I}}{(A \wedge B) \supset (B \wedge A)}\,\supset\mathtt{I}^u$$

2.2.2 Sequent Calculus

In natural deduction, the flow of information is bi-directional. In other words, flow of information in elimination rules is downwards and flow of information in introduction rules is upwards. As we see in the example in previous section, we have always two possible way to meet propositions. Due to the dual way of information flow (bottom-up and top-down), it is difficult to automate proof search in natural deduction. We need a deterministic mechanism to find derivations for a given proposition. The deterministic way of flow of information should be either downwards or upwards.

In order to address this problem, Gentzen proposed in 1935 his sequent calculus. However, his primary idea for using sequent calculus was to prove the consistency of his natural deduction system. Kleene, in his seminal book *Introduction to Metamathematics* [27] gave the first formulation of sequent calculus with its modern style in order to employ sequent calculus to define logics and their proofs.

Sequent calculus offers a system to flip elimination rules in natural deduction upside-down. Along with this modification, proof search steps in sequent calculus proceed only bottom-up. This modification divides inference rules into *right* and *left* rules, which correspond to introduction and elimination rules of natural deduction, respectively. We denote the notation of sequent calculus as a *sequent* by

$$A_1, ..., A_n \Rightarrow A,$$

where propositions placed at the left side of arrow, A_1 to A_n, are called assumptions and the judgement placed at the right side of arrow, A, is called the goal. The right rules apply to the goal, while the left rules apply to assumptions. Here is the right rule for conjunction in sequent calculus,

$$\frac{\Gamma \Rightarrow A \qquad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B} \wedge R$$

The above right rule in sequent calculus is almost the same as the introduction rule in natural deduction. When we read this rule from bottom up, we say that the right rule of conjunction divides the goal $(A \wedge B)$ into two parts, A and B.

The left rule is different from the elimination rule of conjunction in natural deduction:

$$\frac{\Gamma, (A, B) \Rightarrow C}{\Gamma, (A \wedge B) \Rightarrow C} \wedge L$$

The left rule is viewed from bottom-up direction and the above rule says that A and B are inserted to context of assumptions when $A \wedge B$ is on the left side of the arrow.

In natural deduction, we have given an example derivation of $(A \wedge B) \supset (B \wedge A)$. We will see the derivation of this example in sequent calculus. Before that, it is necessary to look at the right implication rule:

$$\dfrac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \supset B} \supset R$$

It is time to describe how we find the proof search of $(A \wedge B) \supset (B \wedge A)$. In the first step, the rule is written at the right hand side.

$$\dfrac{}{. \Rightarrow (A \wedge B) \supset (B \wedge A)}$$

First, we apply the right rule of implication:

$$\dfrac{A \wedge B \Rightarrow B \wedge A}{. \Rightarrow (A \wedge B) \supset (B \wedge A)} \supset R$$

Now, it is time to apply left rule of conjunction:

$$\dfrac{\dfrac{A, B \Rightarrow B \wedge A}{A \wedge B \Rightarrow B \wedge A} \wedge L}{. \Rightarrow (A \wedge B) \supset (B \wedge A)} \supset R$$

The last step is applying right rule of conjunction:

$$\dfrac{\dfrac{\dfrac{A, B \Rightarrow B \qquad A, B \Rightarrow A}{A, B \Rightarrow B \wedge A} \wedge R}{A \wedge B \Rightarrow B \wedge A} \wedge L}{. \Rightarrow (A \wedge B) \supset (B \wedge A)} \supset R$$

The top two sequents, $A, B \Rightarrow B$ and $A, B \Rightarrow A$, are initial rules where we can achieve goal B and A from assumptions (A, B).

2.2.3 Proof Terms

Howard, suggested in his book [22] that there is a strong correspondence between intuitionistic derivations and λ-calculus and referred to this correspondence as Curry-Howard Isomorphism. This is an isomorphism between formulas and types and also between derivations and simply-typed λ-terms. This isomorphism is generally called *propositions-as-types* and *proofs-as-programs.* However, we rather employ the latter since we would like to find programs in a robotic domain.

In order to illustrate the relationship between proofs and programs we need a new judgment,

$$M : A$$

where M is *proof term* for proposition A. Moreover, this interpretation can be read as "*M is a program of type A*". Since we have new judgement, we need to annotate all the inference rules of natural deduction with proof terms. Previously, we have an example derivation of $(A \wedge B) \supset (B \wedge A)$ in natural deduction. We will show how proof terms can be assigned to each derivation in proof search steps. Prior to this, we give inference rules of conjunction and implication including proof terms.

$$\frac{M : A\ true \qquad N : B\ true}{\langle M, N \rangle : A \wedge B\ true} \wedge I$$

In this introduction rule, $A \wedge B\ true$ has a combinational proof as a pair, one comes from $A\,true$ and one from $B\,true$. In functional programming, this rule (function) takes two distinct arguments and combines these arguments in a pair and returns this pair.

We have two elimination rules that take a pair as an argument, one of them selects and returns the first element of pair while the other one returns the second element.

$$\frac{M : A \wedge B\ true}{\mathtt{fst}\, M : A\ true} \wedge E_L \qquad \frac{M : A \wedge B\ true}{\mathtt{snd}\, M : B\ true} \wedge E_R$$

The type of conjunction $A \wedge B$ corresponds to the product type $A \times B$.

In programming languages, we can define a function f of a variable x by writing for example $f(x) = x^2 + 1$. This example can be transformed to $f = \lambda x.x^2 + 1$, that is form a functional object by λ-abstraction. The proof of $A \supset B\ true$ as a function which transforms a proof of $A\ true$ into a proof of $B\ true$. Therefore, λ-abstraction is a great candidate to annotate the implication rule.

$$\frac{\begin{array}{c}\overline{u : A}\ u \\ \vdots \\ M : B\end{array}}{\lambda u : A.M : A \supset B} \supset I^u$$

Using these changes on inference rules, we can annotate the proof of $(A \wedge B) \supset (B \wedge A)$ with proof terms. Thus, we obtain a function which takes a pair $\langle M, N \rangle$ and returns the reverse pair $\langle N, M \rangle$.

$$\frac{\dfrac{\dfrac{\overline{u : A \wedge B}\ u}{\mathtt{snd}\, u : B} \wedge E_R \quad \dfrac{\overline{u : A \wedge B}\ u}{\mathtt{fst}\, u : A} \wedge E_L}{\langle \mathtt{snd}\, u, \mathtt{fst}\, u \rangle : B \wedge A} \wedge I}{(\lambda u.\, \langle \mathtt{snd}\, u, \mathtt{fst}\, u \rangle) : (A \wedge B) \supset (B \wedge A)} \supset I^u$$

The above example illustrates that the proof that is constructed by proof terms directly corresponds to a program. In addition to this, the proposition in this example should correspond to a type by Curry-Howard Isomorphism. Before this, it is necessary to give the summary of propositions and types which correspond to each other as shown below:

In the above illustration, the base type b and the proposition A are left unspecified. From this encoding, the type of the example proposition $(A \wedge B) \supset (B \wedge A)$

$$\begin{array}{rcccccc} \text{Types } \tau ::= & b \mid & \tau_1 \times \tau_2 \mid & \tau_1 \rightarrow \tau_2 \mid & \tau_1 + \tau_2 \mid & \mathbf{1} \mid & \mathbf{0} \\ \text{Propositions } A ::= & p \mid & A_1 \wedge A_2 \mid & A_1 \supset A_2 \mid & A_1 \vee A_2 \mid & \top \mid & \bot \end{array}$$

is $(\tau_1 \times \tau_2) \rightarrow (\tau_2 \times \tau_1)$ when we take τ_1 as the type of proposition A and τ_2 as the type of proposition B.

2.2.4 Proof Normalization

The Proof Normalization confirms the existence of normal forms in natural deduction by using Normalization Theorem. This theorem asserts that every formula in natural deduction has normal form. In addition to this, The Church-Rosser property states that this normal form is unique. The proof of this property is not given in this thesis, however the result of this theory is highly important. As we have seen in the previous section, the proofs directly corresponds to a program. Therefore, if one can find at most one normal form of a proof, one can also find one program for the application domain. Furthermore, this program is directly corresponds to a plan in terms of robotics applications. Uniqueness of this normal proof makes possible to find only a single plan under same conditions.

2.2.4.1 Cut Elimination in Sequent Calculus

In order to establish soundness and completeness with respect to natural deductions, a rule called *cut* is added to the sequent calculus. The cut rule is written as,

$$\frac{\Gamma \Rightarrow A \qquad \Gamma, A \Rightarrow C}{\Gamma \Rightarrow C}\ \mathit{cut}$$

When the cut rule is viewed in the bottom-up direction during proof search, it introduces a new and arbitrary proposition A. Clearly, this introduces a great amount of non-determinism into the search. The cut elimination theorem tells us that we never need to use this rule. In other words, we should prove that cut rule is *admissible.* For the first-order intuitionistic logic in Pfenning's notes [34], the admissibility of cut is proved using inductive techniques. All the rules have the property that the premises contain only instances of propositions in the conclusion, or parts. This latter property is often called the *subformula property.*

2.3 Automated Theorem Proving

In previous sections, we have seen the isomorphism between intuitionistic proofs and programs, and between propositions and types. Moreover, we have described how we can find proofs by using bottom-up search. An automated theorem prover is a tool that searches the proof of a given proposition and eventually terminates and either gives a proof or fails [35].

Depending on the problem, proof search in logic can have a variety of applications. In the domain of planning problems searching for a proof means searching for a plan. In the domain of functional programming, searching for a proof means searching for a program satisfying a given specification. For example, assume that a robot wants to traverse in an environment that has some obstacles. If we want the robot to move across this environment autonomously, degrading the planning problem to proof searching problem and placing an automated theorem prover into the robot will be enough. This robot can find a path using the automated theorem prover in order to construct a plan without consulting a centralized computer.

Depending on the underlying logic, the problem of deciding whether a proof can be found varies from trivial to impossible. For propositional logic, the problem is decidable but NP-complete. For a first order predicate calculus, valid statements can be proved, however invalid statements cannot always be recognized making it undecidable.

We should specify some rules to provide determinism for automated theorem prover. The important question here is how the theorem prover will select which rule to apply among set of propositions. Before giving the answer to this question, the new concept *invertible* should be defined. An invertible rule means that the premises of this rule are derivable whenever the conclusion is derivable. The usual direction states that the conclusion is evident whenever the premises are. For example, the following rule is invertible,

$$\frac{\Gamma \Rightarrow A \qquad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B}\ \wedge R$$

On the other hand, the following rule is non-invertible,

$$\frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A \vee B}\ \vee R_1$$

The following sections describe some techniques to reduce non-determinism in proof search.

2.3.1 Inversion

Theorem prover applies invertible rules whenever possible because it never loses completeness. The order of these rule applications does not matter. After some steps, we arrive at a sequent where all applicable rules are non-invertible. At this point, the theorem prover applies one of the non-invertible rules.

2.3.2 Unification

Unification is a technique for eliminating existential non-determinism. When proving a proposition of the form $\exists x : A$ by its right rule in the sequent calculus, we must supply a term t and then prove $[t = x]A$. The domain of quantification may include infinitely many terms. At that point, we could not try all possible terms t. Rather, we postpone the choice of t and instead substitute x to a new variable X called a meta-variable. When we reach initial sequents, we check if there is a substitution for the meta-variable such that the hypothesis matches the conclusion. If so, we apply this instantiation globally to the partial derivation and continue to search for proofs of other subgoals. This instantiation for meta-variables is called unification.

Herbrand gives the first description of a unification algorithm. However, the comprehensive work on unification is introduced into automative deduction by Alan Robinson [42]. Generally variants of Robinson's algorithm are still used in the subject of theorem proving.

Chapter 3

Example Domain

3.1 Traditional Blocks World

3.1.1 Description

The blocks world is one of the most commonly used example domains for planning problems within artificial intelligence research [19]. It consists of a set of blocks on the table and a robot capable of picking up pieces and placing them on either the table or another block. The goal is to stack blocks vertically to achieve a desired stacking order. One of the main advantages of the Blocks World for us as an example domain is that it provides a simple setting in which reasoning with changing state information can be studied. Clearly, this is an important element in robotic planning problems since robots are expected to perform useful physical work, thereby effecting the state of the environment and themselves. Blocks World has simple discrete actions and models nontrivial constraints on ordering of actions (i.e. which block has to be picked up and placed first).

In general, planning in the Blocks World and other similar domains involves an initial configuration as well as a goal state to be achieved at the end of the plan. Therefore, we need to describe the state, the moves and the goal to be achieved in a logical form appropriate for automatic construction of plans.

The first element to be introduced in such a description is the definition of logical propositions describing the state of the environment at a given time. For example, propositions listed in Table 3.1 can be used to capture the current state in Blocks World.

$\mathsf{on}(x, y)$	block x is on block y
$\mathsf{tb}(x)$	block x is on the table
$\mathsf{clear}(x)$	the top of block x is clear
empty	robot arm is empty
$\mathsf{hold}(x)$	robot arm holds block x

Table 3.1: Propositions for the Blocks World Domain

Using these propositions, one can describe an example initial state such that block a is on block b, block b and block c are on the table and the robot arm is empty as follows:

$$\Delta_0 = (\mathsf{on}(a, b),\ \mathsf{tb}(b),\ \mathsf{clear}(a),\ \mathsf{tb}(c),\ \mathsf{clear}(c),\ \mathsf{empty}).$$

In addition to this initial state, one can also describe the goal state using the same set of propositions. Having block b on block c may be the goal state for the above example,

$$\Delta_g = \mathsf{on}(b, c).$$

Moreover, intermediate states can also be described using the same set of propositions. In order to achieve the example goal state with a given initial state, one of the example intermediate states can be

$$\Delta_i = (\mathsf{tb}(b),\ \mathsf{tb}(c),\ \mathsf{clear}(c),\ \mathsf{holds}(a)).$$

3.1.2 Describing Actions and the Frame Problem

Along with the set of propositions, we need to declare some actions in order to affect changes on these states. We consider four possible legal actions for our application domain:

1. Picking up a block which is on the table.
2. Picking up a block which is on the other block.
3. Putting a block on the table.
4. Putting a block on another block.

In order to model these actions, logical implication is a possible candidate. In general, logical implication is a logical relation that holds between a set S of formulae and a formula A when every model of S is also a model of A and is donated as

$$S \Rightarrow A.$$

We then say that A *is an logical consequence of* S. In the above implication, S is called the antecedent, while A is called the succedent. In the definition of implication, we say that it relates the antecedent and succedent part. Because of the nature of implication, this relation is strong.

In the beginning of this section, we mentioned that we have four actions to describe the moves for our application domain. Since we intend to use a logical formalism to solve planning problems, the logical implication would be appropriate for describing actions. For example, we can use the following four implications in Table 3.2 to model the four actions above.

In practice, it seems that these implications are capable of giving consistent ending states for the given initial states. However, there are several problems associated with these implications. Incorrect encoding is the main issue about this

$$\begin{array}{l}
\forall\, x.\ \forall\, y.\ (\mathsf{empty} \land \mathsf{tb}(x) \land \mathsf{clear}(x)) \Rightarrow (\mathsf{holds}(x)) \\
\forall\, x.\ \forall\, y.\ (\mathsf{empty} \land \mathsf{clear}(x) \land \mathsf{on}(x,y)) \Rightarrow (\mathsf{holds}(x) \land \mathsf{clear}(y)) \\
\forall\, x.\ \forall\, y.\ (\mathsf{holds}(x)) \Rightarrow (\mathsf{empty} \land \mathsf{tb}(x) \land \mathsf{clear}(x)) \\
\forall\, x.\ \forall\, y.\ (\mathsf{holds}(x) \land \mathsf{clear}(y)) \Rightarrow (\mathsf{empty} \land \mathsf{clear}(x) \land \mathsf{on}(x,y)).
\end{array}$$

Table 3.2: Implications for the Blocks World Domain

logical formalism. Because of the nature of implication, propositions in the antecedent part will preserve after applying implication rule. For instance, assume that we have empty, $\mathsf{tb}(a)$ and $\mathsf{clear}(a)$ propositions initially. After executing the first implication in Table 3.2, the new proposition $\mathsf{holds}(a)$ is added to the proposition set without deleting any other propositions. Therefore, two contradictory propositions $\mathsf{holds}(a)$ and empty are added into the proposition set. This is because the ordinary predicate calculus has no notion of state.

The frequently applied solution for this problem in the literature is to add a time parameter to propositions which are then changed as time goes on. Since all propositions in Table 3.1 depend on time, they all need to be changed accordingly as shown in Table 3.3.

$\mathsf{on}(x,y,t)$	block x is on block y at time t
$\mathsf{tb}(x,t)$	block x is on the table at time t
$\mathsf{clear}(x,t)$	the top of block x is clear at time t
$\mathsf{empty}(t)$	robot arm is empty at time t
$\mathsf{hold}(x,t)$	robot arm holds block x at time t

Table 3.3: Propositions for the Blocks World Domain with Time Extension

In addition, our actions also need to be modified to accommodate for the newly introduced time variables as illustrated in Table 3.4.

As a result of these modifications, contradictory propositions will not be drawn anymore. However, while fixing the problem, a new problem emerges. Not specifying the unchanged propositions in these logical implications leads this problem. For instance, the first implication rule says that when robot arm is empty, block x is on the table and there is no other block on block x at time t, the robot arm

$$\begin{array}{ll}
\forall\, x.\ \forall\, y.\ \forall\, t. & (\mathsf{empty}(t) \wedge \mathsf{tb}(x,t) \wedge \mathsf{clear}(x,t)) \Rightarrow (\mathsf{holds}(x,t+1)) \\
\forall\, x.\ \forall\, y.\ \forall\, t. & (\mathsf{empty}(t) \wedge \mathsf{clear}(x,t) \wedge \mathsf{on}(x,y,t)) \Rightarrow \\
 & (\mathsf{holds}(x,t+1) \wedge \mathsf{clear}(y,t+1)) \\
\forall\, x.\ \forall\, y.\ \forall\, t. & (\mathsf{holds}(x,t)) \Rightarrow (\mathsf{empty}(t+1) \wedge \mathsf{tb}(x,t+1) \wedge \mathsf{clear}(x,t+1)) \\
\forall\, x.\ \forall\, y.\ \forall\, t. & (\mathsf{holds}(x,t) \wedge \mathsf{clear}(y,t)) \Rightarrow \\
 & (\mathsf{empty}(t+1) \wedge \mathsf{clear}(x,t+1) \wedge \mathsf{on}(x,y,t+1))
\end{array}$$

Table 3.4: Implications for the Blocks World Domain with Time Extension

can pick the block x and hold it at time $t+1$. If there were another block y on the table at time t, block y should have also been on the table at time $t+1$. However, the first implication rule loses this information. This problem occurs because specifying only which conditions or states are changed by actions do not allow us to conclude that all other conditions or states are not changed. This is called the frame problem in logic and there are numerous studies on solving it in the literature. The common solution is to introduce new predicates or terms to keep unchanged states after executing actions. This is done with different approaches such as in fluent occlusion [44], predicate completion [10] or situation calculus [40]. Because of the current nature of the first-order predicate calculus, these solutions always need to introduce new predicates or terms which are only used for fixing the frame problem. Therefore, one should spend time not only encoding the application domain in logic but also declaring new predicates or terms to avoid the frame problem. Moreover, there will be many unused propositions that remain after executing several actions over and over.

An alternative and easier way to solve such a problem is changing the nature of the first-order predicate calculus by introducing the notion of state in the logical formalism itself. Hence, changing the way in which assumptions are used in the rules would be sufficient for this purpose. If we have rules that use every assumption exactly once, the notion of state will be introduced and the frame problem will be solved automatically without specifying new predicates or terms. Furthermore, it is easily possible using the linear logic which will be described in detail in Chapter 4.

3.2 The Balanced Blocks World 3D

In the traditional Blocks World, the problem is easy to solve using linear logic which includes the notion of state. However, in real life one also has to deal with the problem of achieving balance of blocks while putting a block on another block. We have a new version of the Blocks World named as the *Balanced Blocks World 2D*, in which the desired order of the blocks is also considered in conjunction with the dynamic balance of blocks [45]. In such a problem, while putting down the block A to the block B, the center of mass of block A should lie on the supporting surface of block B (Figure 3.1). However, only placing the center of mass of the latest placed block on the supporting surface of the block tower would not be enough to satisfy the balance of the entire block tower. The combination of the center of mass of the latest placed block and the top block in a tower should also be on the supporting surface of the second top block in the tower. For the entire block tower, there has to be a check for each block in the tower so that the combination of the centers of mass of the blocks above it have to be on its supporting surface. This modification is introduced into the Traditional Blocks World so that the planning mechanism deals not only with the order of the actions to satisfy the desired goal but also dynamical features of the environment.

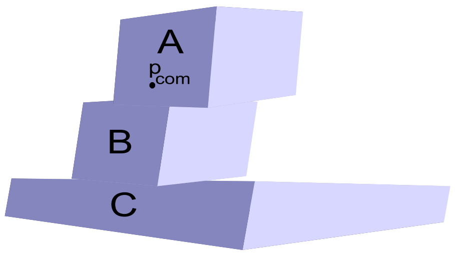

Figure 3.1: 3D Illustration of Properly Placing Block A onto Block B

3.2.1 Definition and Domain Properties

In this section, a new blocks world domain called *Balanced Blocks World 3D* will be described in detail. This domain has the same characteristics with the Balanced Blocks World 2D in that it combines dynamical constraints with state transitions. However, unlike Balanced Blocks World 2D, the constraint domain is composed of set constraints. Since the only contact part of a newly placed block with the top block in a tower is its base, the points of base areas of blocks will form our sets. For the sake of simplicity, the base of every block is assumed to be the same as its top surface. Figure 3.2 illustrates definitions related to these base areas for every block while Table 3.5 defines concepts and abbreviations in this figure.

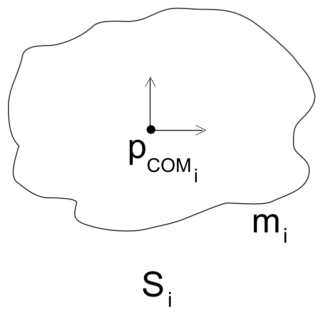

Figure 3.2: Concepts in Blocks World 3D

p_{COMi}	center of mass point of Object i at Frame i
m_i	mass of Object i
S_i	set of all points in Object i at Frame i

Table 3.5: Abbreviation and Meanings in Blocks World 3D

Each object has its own set and a reference frame called F_i such that its center of mass is located at the origin. In order to relate the locations of objects to each other, it is necessary that their positions and orientations are represented in the World Frame W. Therefore, we need to rotate and translate all the points of an object from its own frame to the World Frame W. Table 3.6 definitions are needed for that purpose.

We now need to relate these definitions to each other. Equation 3.1 associates the coordinates of the center of mass point of an object with the World Frame W by rotating and translating the center of mass point of an object from its current frame to the World Frame W.

p_i	an arbitrary point of Object i expressed in Frame i
${}^0p_i$	an arbitrary point of Object i expressed in the World Frame W
${}^0S_i$	set of all points of Object i expressed in the World Frame W
${}^0_iR$	rotation matrix from Frame i to the World Frame W
${}^0_it$	translation from Frame i to the World Frame W

Table 3.6: Definitions in Blocks World 3D

$$ {}^0p_{COM\,i} = \{{}^0_iR \; p_{COM\,i} + {}^0_it\} \tag{3.1} $$

Equation 3.2 can be used to transform all points of an object from the current frame to the World Frame W.

$$ {}^0S_i = \{{}^0_iR \; p_i + {}^0_it \quad | \quad p_i \in S_i\} \tag{3.2} $$

3.2.2 Constraints

In order to ensure that the tower remains balanced, we should not only check that the new block itself remains in balance, but also make sure that the placement does not disturb existing blocks on the tower. To this end, we recursively compute the center of mass of connected groups of blocks from top to bottom and ensure that all combined centers of mass lie on support surfaces. Thus, the following constraints have to be satisfied to ensure that the towers balanced. We assume that we have k number of objects.

The first constraint is that the center of mass point of the block that is placed on the top block has to be on the support surface of the top block. In set theory, this constraint will be expressed in a way that the center of mass point of the newly placed block has to be the member of the set of all points of the top block in the tower provided that both are on the World Frame W. It is modelled in Equation 3.3.

$$ {}^{0}p_{COMk} \in {}^{0}S_{k-1} \tag{3.3} $$

The other constraint is that all the combined center of mass of the connected groups of blocks from top to bottom has to lie on support surfaces. Assume that we have $k-1$ number of blocks and we would like to place another block on top of them. In such a condition, we should start to observe that the combination of the center of mass point of the top two blocks should be the member of the set of the top third block. This condition check will continue until reaching the bottom block as an support surface. This constraint is given in Equation 3.4.

$$ \sum_{j=0}^{i} \frac{m_{k-j} \quad {}^{0}p_{COMk-j}}{m_{k-j}} p_{COMk} \in {}^{0}S_{k-i-1} \quad where \quad i > 0 \quad and \quad i < k-1 \tag{3.4} $$

3.2.3 Term Language

So far, we have described the definition and properties of our application domain, relations between these definitions and constraints on this domain. However, all these explanations have to be encoded in a way that they can be used in a logical system. Hence, the term language of our domain will be defined using BNF notation. It will subsequently form the main part of the constrained domain for linear logic.

Set:	s_i | `Rotate`(Set, Angle) | `Translate`(Set, Point) | `toSet`(Point)
Point:	`Point_Def`(Coordinate, Coordinate) | `Scale`(Point, Point, Mass, Mass)
Mass:	m_i
Coordinate:	x | 0
Angle:	y
Constraint:	Set $\supset$ Set

Table 3.7: BNF notation of Blocks World 3D

In Table 3.7, BNF notations for Blocks World 3D are defined. Here, constraints consist of only `Set`s. Since we would like to find whether the center of mass point is the member of a set, the `toSet` predicate is employed to transform an arbitrary point to the set which will include only one element. In order to rotate and translate the set from its own frame to the World Frame, the `Rotate` and `Translate` predicates are used respectively. The `Scale` predicate calculates the mass balance point between two points. The usage of this encoding will be found in Chapter 4.

Chapter 4

Intuitionistic Linear Logic

4.1 Introduction and Motivation

Linear Logic was discovered by J.Y. Girard in 1987 and published in his famous paper [16]. In the abstract of this paper, he states that "completely new approach to the whole area between constructive logics and computer science is initiated". The fundamental idea under Linear Logic is to control the use of resources.

In Chapter 2, we clarified the difference between Intuitionistic and Classical Logics. One of the important features for Intuitionistic Logic over Classical Logic was the isomorphism between proofs and programs. This has the benefit that each proof in intuitionistic logic directly corresponds to a program. Therefore, a specific domain problem such as robotic planning problem would be possible to solve only searching a proof by using proof search procedures on the application domain that is encoded in logical form. Since we deal with the planning problems, intuitionism will fit for our purpose. The materials in this chapter have been inspired from the Pfenning's notes about Intuitionistic Linear Logic [36].

In Chapter 3, while describing about the frame problem, we noted that we needed to introduce the notion of state. We have already described the notion of judgement and proposition in Chapter 2. However, this two forms of judgement is not sufficient to explain logical reasoning from assumptions. Therefore, we need to introduce new primitive notions such as *linear hypothetical judgement* and *linear hypothetical proof.* The general form of linear hypothetical judgement can be written as

$$A_1 \; true, ..., A_n \; true \Rightarrow C \; true.$$

The meaning of this judgement is that we can prove C from assumptions $A_1, ..., A_n$, using every assumption exactly once. We say that the left side of the entailment part is the consumable resources, where the right side is the goal to be achieved. Since the notion of state concept is introduced using such a judgement, the planning problems which are reduced into the state transition problem could be solved by the logical viewpoint.

The judgement above allows only linear hypothesis. Hence, we have no chance to encode ordinary intuitionistic logic to our system. Furthermore, we mostly need some assumptions that will be used everytime, called non-consumable resources. Therefore, we need to make a distinction between consumable and non-consumable resources in our representation. Generally, the Δ letter is used to range over a collection of linear assumptions while the Γ is used to range over a collection of assumptions that are used arbitrarily many times. These letters correspond to a unique context where the Δ and the Γ letters form the restricted and unrestricted context respectively. For all these definitions, our judgement will be in a form such that:

$$\Gamma; \Delta \Rightarrow A \; true.$$

The combination of linearity and intuitionism will allow us to use both the consumable resources and proofs-as-programs property. This combination is named as Intuitionistic Linear Logic in literature. The proof theory and the connectives of this logic will be mentioned in later sections.

One has to bear in mind that linear assumptions are viewed as resources and conclusions are viewed as the products by spending the given resources.

Therefore, two of three structural rules, *contraction* and *weakening*, of traditional logics are absolutely rejected. The contraction states that a conclusion which follows from two same assumptions can be derived from just one assumption.

$$\frac{\Gamma; A, A \Rightarrow B}{\Gamma; A \Rightarrow B}\ Contraction$$

The weakening expresses that if a conclusion follows from some assumptions then that conclusion can also be derived after increasing the number of these assumptions.

$$\frac{\Gamma; A \Rightarrow B}{\Gamma; A, C \Rightarrow B}\ Weakening$$

The only structural rule that intuitionistic linear logic preserve is the *exchange* which states that the order of the assumptions in derivation does not matter.

$$\frac{\Gamma; A, C \Rightarrow B}{\Gamma; C, A \Rightarrow B}\ Exchange$$

4.2 Linear Connectives with an Example

In this section, we give an introduction to the linear connectives and an example in order to provide the reader with an initial familiarisation for their meanings and usage. He can also find the formal declarations of these connectives in the next section. Before that, a fundamental formula will be mentioned firstly.

$A \Rightarrow A$ is the first rule which is derived from the nature of linear hypothetical judgement itself. It means that the resource A is spent to give a product of A.

Now it is time to give an initial information about the linear connectives. One of the most notable feature about the intuitionistic linear logic is that it has two forms of conjunction ($\otimes$ and $\&$) and two forms of truth ($\top$ and $\mathbf{1}$). On the other hand, there is only one connective for implication ($\multimap$), one form of falsehood ($\mathbf{0}$) and one for disjunction ($\oplus$). There are also various types of linear connectives other than these ones in the literature, however all of the connectives that are mentioned above are utilized for providing intuitionistic purposes while the others satisfy the classical objectives. (*e.g.* multiplicative disjunction connective ($\parr$) and bottom connective($\bot$))

- Linear Implication ($\multimap$):

  $A \multimap B$ means that B can be produced by consuming an A.

- Simultaneous Conjunction ($\otimes$):

  If both A and B are true in the same state then we can write $A \otimes B$. $A \otimes B$ means that our resources can produce both A and B simultaneously.

- Alternative Conjunction ($\&$):

  $A \& B$ means that our resources can make A or B available, but not both simultaneously. This is called *internal choice.* For instance, if we have one dollar and the price of a coffee and a tea is one dollar, then, we have to make a choice to buy a coffee or tea. Our resource, one dollar, can only produce one of them.

- Disjunction ($\oplus$):

  $A \oplus B$ means that our resources can make either A or B available, but you do not know which one is produced. This is called *external choice.* For example, if we have one dollar and we would like to buy a coffee or a tea from vending machine and this machine gives us one of them randomly, then $A \oplus B$ shall be used.

- Unit ($\mathbf{1}$):

  The goal $\mathbf{1}$ can always be produced from the resource of nothing. This is the identity for simultaneous conjunction. ($A \otimes \mathbf{1} \equiv A$)

- Top ($\top$):

The goal $\top$ can always be achieved regardless of which resource we have. It always consumes all of the resources. $\top$ is the identity for alternative conjunction. ($A \,\&\, \top \equiv A$)

- Impossibility (**0**):

  The **0** is the identity for the disjunction. It corresponds to the impossible resource, so that external choice of **0** and A is always A. ($A \oplus \mathbf{0} \equiv A$)

- "Of Course" Modality (!):

  The ! is a way to connect the unrestricted hypothesis with restricted ones. It will be described in next section clearly.

Using these new linear connectives, we can now define an example domain and encode this domain to intensify the meanings of these connectives. The French Restaurant example is one of the famous examples and makes the reader to understand the matter easily.

Menu A: FF 200	FF(200) $\multimap$
Onion Soup or *Clear Broth*	((OS & CB)
Honey Glazed Duck	$\otimes$ HGD
Peas or *Red Cabbage* (according to season)	$\otimes$ (P $\oplus$ RC)
New Potatoes	$\otimes$ NP
Chocolate Mousse (FF 30 extra)	((FF(30) $\multimap$ CM) & **1**)
Coffee	$\otimes$ C
(unlimited refills)	$\otimes$ (!C))

Table 4.1: French Restaurant Menu and Corresponded Linear Logic Encoding

In Table 4.1, note that the alternative conjunction(&) is used when a customer can determine his own meal. However, the disjunction($\oplus$) is used when a meal varies according to season. Also he has to pay extra 30 French Francs to eat chocolate mousse or he can choose nothing (**1**) to eat.

4.3 Proof Theory

Developing linear logic in the form of natural deduction, is highly economical, in that we only need one basic judgement (A true) and two judgement forms (linear and unrestricted hypothetical judgements) to explain the meaning of all connectives we have encountered so far. However, it is not well-suited for proof search, because it involves both forward and backward reasoning.

In Chapter 2, we have mentioned that sequent calculus is the best mechanism to do proof search because of its nature of backward reasoning. In this section, we develop a sequent calculus rather than natural deduction as a calculus of proof search for Intuitionistic Linear Logic.

We now summarize the rules of Intuitionistic Linear Logic. The logic we consider here comprises the following logical operators.

$$
\begin{array}{rll}
\text{Propositions } A := & P & \text{Atoms} \\
& \mid A_1 \multimap A_2 \mid A_1 \otimes A_2 \mid \mathbf{1} & \text{Multiplicatives} \\
& \mid A_1 \,\&\, A_2 \mid \top \mid A_1 \oplus A_2 \mid \mathbf{0} & \text{Additives} \\
& \mid \forall x.A \mid \exists x.A & \text{Quantifiers} \\
& \mid A \supset B \mid !A & \text{Exponentials}
\end{array}
$$

4.3.1 Sequent Calculus

Based on the notion of linear hypothetical judgment, we now introduce the various connectives of Intuitionistic Linear Logic using sequent calculus. Before giving details of linear connectives, we firstly describe two rules. One of them called *init* explicitly states a connection between resources and goals and the other one called *copy* relates the unrestricted context and restricted context. Most of the information in this section is borrowed from Pfenning's notes about Intuitionistic Linear Logic [36].

Init. It is the basic sequent that means with resource A, we can achieve goal A

$$\frac{}{\Gamma; A \Rightarrow A}\ init\ .$$

Copy. It creates a resource from the unrestricted context and copies it into the restricted context

$$\frac{\Gamma, A; \Delta, A \Rightarrow A}{\Gamma, A; \Delta \Rightarrow A}\ copy\ .$$

The remaining rules are divided into *right* and *left* rules. Each linear connective has own left and right rules.

Simultaneous Conjunction. Assume we have some resources and we want to achieve goals A and B simultaneously, written as $A \otimes B$ (A *tensor* B). We need to split our resources into Δ_1 and Δ_2 and show that with resources Δ_1 we can achieve A and with Δ_2 we can achieve B

$$\frac{\Gamma; \Delta_1 \Rightarrow A \qquad \Gamma; \Delta_2 \Rightarrow B}{\Gamma; \Delta_1, \Delta_2 \Rightarrow A \otimes B}\ \otimes R$$

Here, the important part is that the splitting of resources, from bottom to top, is a non-deterministic operation.

The left rule captures that if we achieve goal C using simultaneously occurring resources A and B, then we could achieve same goal C from same resources A and B

$$\frac{\Gamma; \Delta, A, B \Rightarrow C}{\Gamma; \Delta, A \otimes B \Rightarrow C}\ \otimes L\ .$$

Alternative Conjunction. We write $A \mathbin{\&} B$ if we can goals A and B with the current resources, but only alternatively. The right rule for the alternative conjunction asserts that if we achieve goal $A \mathbin{\&} B$ from resources Δ, then we could achieve goal A or alternatively B from same resource Δ

$$\frac{\Gamma; \Delta \Rightarrow A \qquad \Gamma; \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \mathbin{\&} B} \; \& R.$$

The right rule for alternative conjunction appears to duplicate the resources. However, this is an illusion: since we will actually have to make a choice between A and B, we will only need one copy of the resources.

If we achieve goal C from resources $A \mathbin{\&} B$, we could achieve the same goal C, either from A or B but alternatively. Therefore we need two left rules for that purpose:

$$\frac{\Gamma; \Delta, A \Rightarrow C}{\Gamma; \Delta, A \mathbin{\&} B \Rightarrow C} \; \& L_1 \qquad \frac{\Gamma; \Delta, B \Rightarrow C}{\Gamma; \Delta, A \mathbin{\&} B \Rightarrow C} \; \& L_2.$$

Linear Implication. The linear implication incorporates the linear hypothetical judgement within level of propositions. $A \multimap B$ (pronounced A linearly implies B) is used for the goal of achieving B with resource A. Formally, if we have resource Δ to achieve goal $A \multimap B$, then we can achieve goal B from resources Δ and A

$$\frac{\Gamma; \Delta, A \Rightarrow B}{\Gamma; \Delta \Rightarrow A \multimap B} \; \multimap R.$$

The left rule for linear implication needs splitting resources as in the right rule of simultaneous conjunction. Assume that if we have resources Δ and $A \multimap B$ to achieve goal C, then we need to divide our resource Δ into two parts such as Δ_1 and Δ_2 to achieve goal A from resource Δ_1 and to achieve goal C from resources B and Δ_2

$$\frac{\Gamma; \Delta_1 \Rightarrow A \qquad \Gamma; \Delta_2, B \Rightarrow C}{\Gamma; \Delta_1, \Delta_2, A \multimap B \Rightarrow C} \multimap L.$$

Unit. The trivial goal which requires no resources is written as **1**

$$\frac{}{\Gamma; . \Rightarrow \mathbf{1}} \mathbf{1}R.$$

The left rule asserts that if we have resource **1** to achieve goal C, then we could achieve the same goal C without the resource **1**

$$\frac{\Gamma; \Delta \Rightarrow C}{\Gamma; \Delta, \mathbf{1} \Rightarrow C} \mathbf{1}L.$$

Top. The goal which consumes all resources is written as $\top$

$$\frac{}{\Gamma; \Delta \Rightarrow \top} \top R.$$

It is the unit of alternative conjunction and there is no elimination for the $\top$.

Disjunction. The disjunction $A \oplus B$ (called as external choice) is characterized by two introduction rules

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

The left rule of disjunction asserts that if we have resources $A \oplus B$ and Δ to achieve goal C, then we could achieve the same goal C from A and Δ or B and Δ

$$\frac{\Gamma; \Delta, A \Rightarrow C \qquad \Gamma; \Delta, B \Rightarrow C}{\Gamma; \Delta, A \oplus B \Rightarrow C} \oplus L.$$

Note that resources Δ appear in both branches, since only one of those two derivations will actually be used to achieve C.

Impossibility. The impossibility $\mathbf{0}$ has no right rule. In the left rule, if we achieve goal C from resource $\mathbf{0}$, then we could conclude nothing.

$$\frac{}{\Gamma; \Delta, \mathbf{0} \Rightarrow C}\ \mathbf{0}L.$$

Universal Quantification. We say $\forall x.A$ is true with given resource Δ if $[a/x]A$ is true with same resource Δ for an arbitrary a. The notation of $[a/x]A$ means that the occurrences of x in A will be replaced by a. The label a must be "new", that is, it may not occur in Δ or A. In other words, the label a is a *parameter* which will be replaced by another term during backward proof search. Here is the right rule for universal quantification:

$$\frac{\Gamma; \Delta \Rightarrow [a/x]A}{\Gamma; \Delta \Rightarrow \forall x.A}\ \forall R^a.$$

The left rule replaces the parameter x with a term t:

$$\frac{\Gamma; \Delta, [t/x]A \Rightarrow C}{\Gamma; \Delta, \forall x.A \Rightarrow C}\ \forall L.$$

Existential Quantification. The left and right rule of the existential quantifiers are such as:

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

Unrestricted Implication. The proof of an unrestricted implication $A \supset B$ allows an unrestricted assumption A to achieve goal B

$$\frac{(\Gamma, A); \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \supset B} \supset R.$$

In the left rule, if we use resource $A \supset B$ and Δ to achieve goal C, then we could achieve goal A without using linear assumptions and achieve C using resource B

$$\frac{\Gamma; . \Rightarrow A \qquad \Gamma; \Delta, B \Rightarrow C}{\Gamma; \Delta, A \supset B \Rightarrow C} \supset L.$$

"Of Course" Modality. This is an alternative way to connect unrestricted and linear hypothesis. Here are the right and left rules:

$$\frac{\Gamma; . \Rightarrow A}{\Gamma; . \Rightarrow !A} !R \qquad \frac{(\Gamma, A); \Delta \Rightarrow C}{\Gamma; (\Delta, !A) \Rightarrow C} !L.$$

We have defined the formal sequent calculus of Intuitionistic Linear Logic so far, the summary of it can be found in Appendix.

4.3.2 The Linear λ-Calculus

In Chapter 2, we have mentioned about the proof terms that carries enough information to reconstruct its deduction when they are used to establish the truth of a proposition. From intuitionistic point of view such proof terms describe constructions. That means a proof can keep its evidence throughout a proof search. Moreover, these proof terms can be seen as a plan and program in the domain of planning problems. We have also mentioned about the Curry-Howard isomorphism in which there is bijective correspondence between proofs in intuitionistic logic and simply typed λ-terms. This correspondence make us possible to find a plan just only constructing proofs in intuitionistic logic. Since we are working on different type of intuitionistic logic, we now need to define new linear λ-terms and assign these terms into the sequent calculus.

4.3.3 Proof Term Assignment for Sequent Derivations

In this section we address the question of how to assign proof terms to sequent calculus derivations. There are two possibilities to do this assignment. One is that we can either develop a new proof term calculus specifically for sequent derivations, or the other is that we can directly assign natural deduction proof terms. The former is more appropriate for our purposes, since we have described our connectives as sequents.

Firstly, we need a new judgement $\Gamma; \Delta \Rightarrow I : A$ which means I is a proof term for A under unrestricted hypotheses Γ and resources Δ. Proof terms to be assigned to each inference rule can be determined by a close examination of the soundness proof for the sequent calculus. Now, we can write down the corresponding proof terms to each inference rule.

Initial Sequents.

$$\frac{}{\Gamma; u : A \Rightarrow u : A}\ \text{init}$$

Note that, for proof term assignments we abandon the previous convention of omitting labels for hypotheses, since proof terms need to refer to them. Using labelized notation will remove the ambiguity that which one of hypotheses is used.

Linear Implication. Linear implication corresponds to a linear function types with corresponding linear abstraction and application. The proof of an implication $A \multimap B$ will be represented by a function which maps proofs of A to proofs of B. The right rule explicitly forms such a function by λ-abstraction and the left rule applies the function to an argument.

$$\frac{\Gamma; \Delta, u : A \multimap M : B}{\Gamma; \Delta \Rightarrow \lambda u : A.M : A \multimap B}\ \multimap R$$

$$\frac{\Gamma; \Delta_1 \Rightarrow M : A \qquad \Gamma; \Delta_2, w : B \Rightarrow I : C}{\Gamma; \Delta_1, \Delta_2, u : A \multimap B \Rightarrow [uM/w]I : C} \multimap L$$

Simultaneous Conjunction. In the rules for the simultaneous conjunction, the proof term for the left rule is a **let** form which deconstructs a pair, naming the components. The linearity of the two new hypotheses means that the variables must both be used in M.

$$\frac{\Gamma; \Delta_1 \Rightarrow M : A \qquad \Gamma; \Delta_2 \Rightarrow N : B}{\Gamma; \Delta_1, \Delta_2 \Rightarrow M \otimes N : A \otimes B} \otimes R$$

$$\frac{\Gamma; \Delta, u : A, w : B \Rightarrow N : C}{\Gamma; \Delta, v : A \otimes B \Rightarrow \text{let } u \otimes w = v \text{ in } N : C} \otimes L$$

One. The multiplicative unit type allows us to consume linear hypotheses without introducing new linear ones.

$$\frac{}{\Gamma; . \Rightarrow * : \mathbf{1}} \mathbf{1}R \qquad \frac{\Gamma; \Delta \Rightarrow C}{\Gamma; \Delta, \mathbf{1} \Rightarrow C} \mathbf{1}L$$

Alternative Conjunction. The proof term for a alternative conjunction is the pair of proofs of the premises.

$$\frac{\Gamma; \Delta \Rightarrow M : A \qquad \Gamma; \Delta \Rightarrow N : B}{\Gamma; \Delta \Rightarrow \langle M, N \rangle : A \,\&\, B} \& R$$

$$\frac{\Gamma; \Delta, w : A \Rightarrow I : C}{\Gamma; \Delta, u : A \,\&\, B \Rightarrow [\text{fst } u/w]I : C} \& L_1 \qquad \frac{\Gamma; \Delta, w : B \Rightarrow I : C}{\Gamma; \Delta, u : A \,\&\, B \Rightarrow [\text{snd } u/w]I : C} \& L_2$$

Top. The additive unit corresponds to a unit type with no operations on it.

$$\frac{}{\Gamma;\Delta \Rightarrow \langle\rangle : \top}\ \top R$$

Disjunction. It uses injection and case as constructor and destructor forms, respectively.

$$\frac{\Gamma;\Delta \Rightarrow M : A}{\Gamma;\Delta \Rightarrow \mathrm{inl}^B M : A \oplus B}\ \oplus R_1 \qquad \frac{\Gamma;\Delta \Rightarrow M : B}{\Gamma;\Delta \Rightarrow \mathrm{inr}^A M : A \oplus B}\ \oplus R_2$$

$$\frac{\Gamma;\Delta, v : A \Rightarrow I : C \qquad \Gamma;\Delta, w : B \Rightarrow J : C}{\Gamma;\Delta, u : A \oplus B \Rightarrow (\mathrm{case}\ u\ \mathrm{of\ inl}\ v \Rightarrow I \mid \mathrm{inr}\ w \Rightarrow J) : C}\ \oplus L$$

Impossibility.

$$\frac{}{\Gamma;\Delta, u : \mathbf{0} \Rightarrow \mathrm{abort}\ u : C}\ \mathbf{0}L$$

To treat the quantifiers we extend our proof term calculus to handle the quantifier rules. We overload the notation by reusing λ-abstraction and pairing. There is no ambiguity, because the proof term for universal quantification binds a term variable x (rather than a proof variable u), and the first component of the pair for existential quantification is a first-order term, rather than a proof term as for conjunction.

Universal Quantification. The proof term for a universal quantifier $\lambda x.A$ is a function from a term t to a proof of $[t/x]A$.

$$\frac{\Gamma;\Delta \Rightarrow [a/x]I : [a/x]A}{\Gamma;\Delta \Rightarrow \lambda x.I : \forall x.A}\ \forall R^a$$

$$\frac{\Gamma;\Delta, w : [t/x]A \Rightarrow I : C}{\Gamma;\Delta, u : \forall x.A \Rightarrow [ut/w]I : C}\ \forall L$$

Existential Quantification. The proof term for an existential $\lambda x.A$ is a pair consisting of a witness term t and the proof of $[t/x]A$.

$$\frac{\Gamma; \Delta \Rightarrow I : [t/x]A}{\Gamma; \Delta \Rightarrow \langle t, I \rangle : \exists x.A}\ \exists R \qquad \frac{\Gamma; \Delta, w : [a/x]A \Rightarrow [a/x]I : C}{\Gamma; \Delta, u : \exists x.A \Rightarrow (\text{let } \langle x, w \rangle = u \text{ in } I) : C}\ \exists L^a$$

Unrestricted Implication. Unrestricted implication corresponds to the usual function type from the simply-typed λ-calculus.

$$\frac{(\Gamma, v : A); \Delta \Rightarrow M : B}{\Gamma; \Delta \Rightarrow \lambda v : A.M : A \supset B}\ \supset R$$

$$\frac{\Gamma; . \Rightarrow M : A \qquad \Gamma; \Delta, w : B \Rightarrow I : C}{\Gamma; \Delta, u : A \supset B \Rightarrow [uM/w]I : C}\ \supset L$$

"Of Course" Modality. The rules for the of course operator allow us to name term of type $!A$.

$$\frac{\Gamma; . \Rightarrow M : A}{\Gamma; . \Rightarrow !M :!A}\ !R \qquad \frac{(\Gamma, v : A); \Delta \Rightarrow I : C}{\Gamma; (\Delta, u :!A) \Rightarrow (\text{ let } !v = u \text{ in } I) : C}\ !L$$

Below is a summary of the linear λ-calculus:

4.4 Problems with ILL

Intuitionistic Linear Logic provides an effective way of capturing state changes and creating programs for the planning problems that we deal with. However, traditional linear logic lacks continuous reasoning. Almost all robotic planning problems include dynamical constraints in conjunction with state transitions. Since we would like to represent and reason about these problems in our logical

$M :=$	u	*Linear Variables*
	$\mid \lambda u : A.M \mid M_1\ M_2$	$A \multimap B$
	$\mid M_1 \otimes M_2 \mid \text{let } u_1 \otimes u_2 = M \text{ in } N$	$A \otimes B$
	$\mid * \mid \text{let } * = M \text{ in } N$	$\mathbf{1}$
	$\mid \langle M_1, M_2 \rangle \mid \text{fst } M_1 \mid \text{snd } M_2$	$A \,\&\, B$
	$\mid \langle \rangle$	$\top$
	$\mid \text{inl}_B M \mid \text{inr}_A M \mid (\text{case } u \text{ of inl } v \Rightarrow I \mid \text{inr } w \Rightarrow J)$	$A \oplus B$
	$\mid \text{abort } M$	$\mathbf{0}$
	$\mid v$	*Unrestricted Variables*
	$\mid \lambda u : A.M \mid M_1\ M_2$	$A \supset B$
	$\mid !M \mid \text{let } v = M \text{ in } N$	!A

formalism, current nature of Intuitionistic Linear Logic is not enough to achieve our goals. Therefore, we need to find a way to combine dynamical constraints with linear logic. Ideally, it would be possible to add domain specific *constraint solvers* into our logical formalism.

4.5 The Constraint Extensions to ILL: CILL

The constraint extension to linear logic will bring us increased expressive power to represent a domain which comprises both dynamical constraints and state changes. Although there are other ways to encode such a domain within pure logical theory, the complexity in practice leads to significant amount of limitations when applied to specific domain. By using constraint solvers specific to a particular domain, we can reduce the complexity associated with the encoding. Thus, the constraint solver itself can handle all the consistency problems in it and we only focus on just representation of constraints in our logic.

In this section, we describe a new representational framework based on Intuitionistic Linear Logic extended with two new logical connectives to incorporate continuous constraints directly into the language.

4.5.1 Additional Connectives for CILL

At the beginning of this chapter, we mentioned that we have a new form of judgements to differentiate the notion of consumable resources and unrestricted hypotheses. Now, we need a new context to collect and solve constraints called *constraint context.* The new judgement defined for that purpose is

$$\Psi \mid \Gamma; \Delta \Rightarrow A \; true.$$

The meaning of this judgement is just extended from our previous judgement that we could achieve goal A only if constraints in Ψ are satisfiable and we have Δ consumable resources and Γ unrestricted hypotheses. Along with the modification on our judgement, the given sequent calculus for Intuitionistic Linear Logic has to be revised. Since newly added constraint context has no effect on the meanings of existing inference rules and connectives, only adding the new context at the beginning of each judgement would be enough. However, we need new connectives in order to relate constraints with our consumable resources and unrestricted hypotheses. Thus, we now describe two additional connectives to incorporate constraints into our formalism.

Constraint Implication. If we have a constraint implication such as, $D \supset_c A$ then we need to show that the goal A can be achieved under the constraint D. The following left and right rules provide a formal definition for this connective:

$$\frac{(\Psi, D) \mid \Gamma; \Delta \Rightarrow A}{\Psi \mid \Gamma; \Delta \Rightarrow D \supset_c A} \supset_c R \qquad \frac{\Psi \models D \qquad \Psi \mid \Gamma; \Delta, A \Rightarrow C}{\Psi \mid \Gamma; \Delta, D \supset_c A \Rightarrow C} \supset_c L.$$

The right rule for this connective asserts that in order to achieve goal $D \supset_c A$, we need to verify that inserting the new constraint D into constraint context should preserve consistency of within context, and that we can also achieve goal A under the same consumable resources and unrestricted hypotheses.

The left rule for this connective is very similar to typical left rules for implication, however now the constraint D has to be handled by an other proof procedure specific to the constraint domain. We denote this proof procedure as $\Psi \models D$, which means the model of Ψ has to also be super set of the model of D. For example, assume that we use integer constraints for our domain and Ψ includes $x > 1$ and $x < 5$. Thus the constraint context knows that x can take the values $\{2, 3, 4\}$ corresponding to the model of Ψ. In addition to this, assume that D includes $x > 2$. In such a condition, the model of D is $\{3, 4, ...\}$. Since the model of D is not a subset of Ψ, the proof procedure concludes that $\Psi \models D$ is not satisfied. All of the procedures described in this example are carried out by a constraint solver which is not an integral part of the logical formalism. In other words, the constraint proof procedure is a black box that takes constraints and returns their consistency or whether the model of current constraints is satisfied or not. All of the procedures for constraints will be described in Chapter 5 under the Constraint Logic Programming section in detail.

Constraint Conjunction. If we would like to show that an example constraint conjunction, $D \wedge_c A$ is achievable, then we need to show that both the goal A can be achieved with given resources and constraint D is satisfiable under given constraints. The following right rule is the formal definition of it:

$$\frac{\Psi \models D \qquad \Psi \mid \Gamma; \Delta \Rightarrow A}{\Psi \mid \Gamma; \Delta \Rightarrow D \wedge_c A} \wedge_c R .$$

Observe that $\Psi \models D$ is handled in the same way as the previous section. The left rule asserts that the constraint D and consumable resource A is ejected from the structure of $\Psi \mid \Gamma; \Delta, D \wedge_c A \Rightarrow C$ and each component is inserted into its own context. Moreover, the constraints context has to preserve its consistency and goal C has to be achieved under the extended resource context and constraint context.

$$\frac{(\Psi, D) \mid \Gamma; (\Delta, A) \Rightarrow C}{\Psi \mid \Gamma; \Delta, D \wedge_c A \Rightarrow C} \wedge_c L$$

There are also some auxiliary definitions to provide soundness and completeness of the CILL system.

Constraint Contradiction. If we have a inconsitent constraint domain, then we conclude that we can achieve any arbitrary goal:

$$\frac{\Psi \models \bot}{\Psi \mid \Gamma; \Delta \Rightarrow C} \ \bot$$

Constraint Split. A constraint can be divided into two or finite parts. For example, if we cannot conclude anything when the constraint is between 1 and 4, then we can try to conclude using both 2 and 3. It seems that constraint splitting will be very useful for set constraints:

$$\frac{\Psi \models \Psi_1 \vee \Psi_2 \quad \Psi_1 \mid \Gamma; \Delta \Rightarrow C \quad \Psi_2 \mid \Gamma; \Delta \Rightarrow C}{\Psi \mid \Gamma; \Delta \Rightarrow C} \ \vee$$

$$\frac{\Psi \models \exists x.\Psi_1(x) \quad \Psi_1(x) \mid \Gamma; \Delta \Rightarrow C}{\Psi \mid \Gamma; \Delta \Rightarrow C} \ \exists$$

4.5.2 An Example: Encoding Blocks World 3D within CILL

Before using this language in Linear Logic, we have to define predicates in order to represent the dynamic state. Each block in Blocks World 3D is named and their geometry is specified through three functions from the set of block names. These functions encode the set of all points of the block and the center of mass point at its own frame as well as the mass of the block. The predicates are shown in Table 4.2. At the first part of Table 4.2, the predicates to define the dynamic state of the system is shown and at the second part of this table, the invariant states about the world is shown.

dynamic state of the system	
tableempty	There are no blocks on the table
$\mathsf{ontable}(a, \theta, t)$	Block a is on the table with given translation t and rotation θ
$\mathsf{available}(a)$	Block a is available for placement
$\mathsf{on}(a, b, \theta, t)$	Block a is on top of Block b with given translation t and rotation θ
$\mathsf{check}(S_b, \Theta_b, t_b, P_a, P_o, m_a, m_o)$	Checks whether given P_a, P_o with combination mass m_a, m_o is on the supporting surface of S_b
$\mathsf{clear}(a)$	The top of Block a is clear
invariant facts about the world	
$\mathsf{support}(a)$	It gives the set of all points of the Block a at its own frame
$\mathsf{comp}(a)$	It gives the center of mass point of the Block a at its own frame
$\mathsf{mass}(a)$	Mass of Block a

Table 4.2: Resource predicates for the Blocks World 3D

Table 4.3 shows encoding of Blocks World 3D actions using CILL properties. As you see, the predicates, Point_Def, Scale, toSet, Rotate and Translate, defined in Chapter 3 are used to integrate the term language and resources predicates of Blocks World 3D.

4.5.3 Example: Placing a Block on Block Tower using CILL

Placing a block on block tower is a good practice to understand the representation of actions and rules in CILL. Our trivial goal is to achieve the balance of block tower when placing block a on block b that is on the table where the rotation of block a must not pass the angle 45 degrees:

$$G := \exists \Theta_a . \exists t_a . \ (\Theta_a <= 45) \wedge_c \mathsf{on}(a, b, \Theta_a, t_a) \otimes \top$$

Here, $\top$ is used for consuming all unused resources at the end of the proof search. Our initial state, where there are two blocks and empty table yield the linear context:

$$\Delta_0 = (\mathsf{tableempty}, \mathsf{available}(a), \mathsf{available}(b))$$

Our unrestricted hypotheses (Γ) include actions that are available in the form of linear implications, summarized in Table 4.3. These implications are used many times without consuming themselves. At this point, the planning problem reduces to finding a proof for the following CILL sequent:

$$\Psi \,|\, \Gamma;\, \Delta_0 \;\Rightarrow\; G$$

The actions $\mathsf{putonblock1}$ and $\mathsf{putonblock2}$ is used to place a block on top of an existing one on the tower. The only difference between them is that the former places a block on the block that is on the table, while the latter puts a block on the block that is on the another block. In order to perform balance check, when a block is placed on another block using above actions, we ensure that the center of mass of connected groups of blocks from top to bottom should lie on support surfaces. This can be possible by calling $\mathsf{check}(\mathsf{support}(b), \Theta_b, t_b, p_a, p_z, m_a, m_z)$ predicate recursively to figure out whether the center of mass point of each block is a subset of supporting surface set until the bottom block is reached. Here, b, Θ_b, t_b is used to form the supporting surface set with respect to the World Frame. p_a is a center of mass point of the block to be checked. p_z is previous center of mass points of blocks above from the block to be checked. m_a is mass of the block to be checked. m_z is previous total mass of blocks above from the block to be checked. The check predicate is introduced for the topmost block with the application of the $\mathsf{putonblock1}$ or $\mathsf{putonblock2}$ according to the situation whether the block is placed on the block that is on the table or not. It will be iterated through the hypothesis $\mathsf{checkiterblock1}$ until the query reaches one above from the bottom block with the hypothesis $\mathsf{checkiterblock2}$ and eventually bottom block with the hypothesis $\mathsf{checkitertable}$.

For our example, the proof search procedure forms the derivation from bottom that putontable hypothesis is firstly used for the block b with rotation Θ_b and t_b. Therefore, we have consumed the predicates tableempty and available(b) and we product the resources clear(b) and ontable(b):

$$\Delta_1 = (\mathsf{available}(a), \mathsf{clear}(b), \mathsf{ontable}(b, \Theta_b, t_b))$$

putonblock1 hypothesis is used for the block a with rotation Θ_a and translation t_a and its orientation will be checked with checkitertable to ensure that the given orientation does not lead the block tower to collapse. We have consumed the resources, available(a) and clear(b), and have produced the resources $\mathsf{on}(a, b, \Theta_a, t_a)$ and $\mathsf{clear}(a)$:

$$\Delta_2 = (\mathsf{on}(a, b, \Theta_a, t_a), \mathsf{clear}(a), \mathsf{ontable}(b, \Theta_b, t_b))$$

The following is the constraint to be satisfied:

$$C_1 : \mathsf{toSet}(\mathtt{Scale}(\mathsf{comp}(a), (0,0), \mathsf{mass}(a), 0)) \supset \mathtt{Translate}(\mathtt{Rotate}(\mathsf{support}(b), \Theta_b), t_b)$$

means center of mass point of block a at World Frame should be the subset of set of all points of block b at World Frame.

However, we are not done yet, since we need to use the right rule of the $\wedge_c$ connective whether our constraint domain C_1 entails the constraint $\Theta_a <= 45$:

$$C_1 \models (\Theta_a <= 45)$$

Eventually, it returns the required limit of orientation for each block where Θ_a must be smaller than 45 degrees and center of mass point of block a at World Frame should be the subset of set of all points of block b at World Frame. This example shows us that we can solve planning problems including both discrete and continuous properties using CILL encodings.

$\mathsf{putontable}$:	$\forall a.\ \exists \Theta_a.\ \exists t_a.\ \mathsf{tableempty} \otimes \mathsf{available}(a) \multimap \mathsf{ontable}(a, \Theta_a, t_a) \otimes \mathsf{clear}(a)$
$\mathsf{pullofftable}$:	$\forall a.\ \forall \Theta_a.\ \forall t_a.\ \mathsf{ontable}(a, \Theta_a, t_a) \otimes \mathsf{clear}(a) \multimap \mathsf{available}(a) \otimes \mathsf{tableempty}$
$\mathsf{pulloffblock}$:	$\forall a.\ \forall b.\ \forall \Theta_a.\ \forall t_a.\ \mathsf{on}(a, b, \Theta_a, t_a) \otimes \mathsf{clear}(a) \multimap \mathsf{available}(a) \otimes \mathsf{clear}(b)$
$\mathsf{putonblock1}$:	$\forall a.\ \forall b.\ \forall \Theta_b.\ \forall t_b.\ \exists \Theta_a.\ \exists t_a.\ \mathsf{clear}(b) \otimes \mathsf{available}(a) \otimes \mathsf{ontable}(b, \Theta_b, t_b) \multimap \mathsf{on}(a, b, \Theta_a, t_a) \otimes \mathsf{ontable}(b, \Theta_b, t_b) \otimes \mathsf{check}(\mathsf{support}(b),\ \Theta_b,\ t_b,\ \mathsf{comp}(a) + t_a,\ \mathtt{Point\_Def}(0,\ 0),\ \mathsf{mass}(a),\ \mathsf{mass}(0)) \otimes \mathsf{testing}(a)$
$\mathsf{putonblock2}$:	$\forall a.\ \forall b.\ \forall c.\ \forall \Theta_b.\ \forall t_b.\ \exists \Theta_a.\ \exists t_a.\ \mathsf{clear}(b) \otimes \mathsf{available}(a) \otimes \mathsf{on}(b, c, \Theta_b, t_b) \multimap \mathsf{on}(a, b, \Theta_a, t_a) \otimes \mathsf{on}(b, c, \Theta_b, t_b) \otimes \mathsf{check}(\mathsf{support}(b),\ \Theta_b,\ t_b,\ \mathsf{comp}(a) + t_a,\ \mathtt{Point\_Def}(0,\ 0),\ \mathsf{mass}(a),\ \mathsf{mass}(0)) \otimes \mathsf{testing}(a)$
$\mathsf{checkitertable}$:	$\forall a.\ \forall b.\ \forall c.\ \forall \Theta_a.\ \forall t_a.\ \forall \Theta_b.\ \forall t_b.\ \forall p_a.\ \forall p_z.\ \forall m_a.\ \forall m_z.\ \mathsf{ontable}(b, \Theta_b, t_b) \otimes \mathsf{testing}(c) \otimes \mathsf{check}(\mathsf{support}(b), \Theta_b, t_b, p_a, p_z, m_a, m_z) \otimes \mathsf{on}(a, b, \Theta_a, t_a) \multimap \mathsf{toSet}(\mathtt{Scale}(p_a, p_z, m_a, m_z)) \supset \mathtt{Translate}(\mathtt{Rotate}(\mathsf{support}(b),\ \Theta_b),\ t_b) \supset_c \mathsf{on}(a, b, \Theta_a, t_a) \otimes \mathsf{ontable}(b, \Theta_b, t_b) \otimes \mathsf{clear}(c)$
$\mathsf{checkiterblock1}$:	$\forall a.\ \forall b.\ \forall c.\ \forall d.\ \forall \Theta_a.\ \forall t_a.\ \forall \Theta_b.\ \forall t_b.\ \forall \Theta_c.\ \forall t_c.\ \forall p_a.\ \forall p_z.\ \forall m_a.\ \forall m_z.\ \mathsf{on}(c, d, \Theta_c, t_c) \otimes \mathsf{on}(b, c, \Theta_b, t_b) \otimes \mathsf{check}(\mathsf{support}(b), \Theta_b, t_b, p_a, p_z, m_a, m_z) \otimes \mathsf{on}(a, b, \Theta_a, t_a) \multimap \mathsf{toSet}(\mathtt{Scale}(p_a, p_z, m_a, m_z)) \supset \mathtt{Translate}(\mathtt{Rotate}(\mathsf{support}(b),\ \Theta_b),\ t_b) \supset_c \mathsf{on}(a, b, \Theta_a, t_a) \otimes \mathsf{on}(b, c, \Theta_b, t_b) \otimes \mathsf{on}(c, d, \Theta_c, t_c)\ \mathsf{check}(\mathsf{support}(c), \Theta_c, t_c, \mathsf{comp}(b) + t_b, \mathtt{Scale}(p_a, p_z, m_a, m_z), \mathsf{mass}(b), m_a + m_z)$
$\mathsf{checkiterblock2}$:	$\forall a.\ \forall b.\ \forall c.\ \forall \Theta_a.\ \forall t_a.\ \forall \Theta_b.\ \forall t_b.\ \forall \Theta_c.\ \forall t_c.\ \forall p_a.\ \forall p_z.\ \forall m_a.\ \forall m_z.\ \mathsf{ontable}(c, \Theta_c, t_c) \otimes \mathsf{on}(b, c, \Theta_b, t_b) \otimes \mathsf{check}(\mathsf{support}(b), \Theta_b, t_b, p_a, p_z, m_a, m_z) \otimes \mathsf{on}(a, b, \Theta_a, t_a) \multimap \mathsf{toSet}(\mathtt{Scale}(p_a, p_z, m_a, m_z)) \supset \mathtt{Translate}(\mathtt{Rotate}(\mathsf{support}(b),\ \Theta_b),\ t_b) \supset_c \mathsf{on}(a, b, \Theta_a, t_a) \otimes \mathsf{on}(b, c, \Theta_b, t_b) \otimes \mathsf{ontable}(c, \Theta_c, t_c) \otimes \mathsf{check}(\mathsf{support}(c), \Theta_c, t_c, \mathsf{comp}(b) + t_b, \mathtt{Scale}(p_a, p_z, m_a, m_z), \mathsf{mass}(b), m_a + m_z)$

Table 4.3: CILL Representations of BW3D Actions and Supporting Rules for Checking Balance

Chapter 5

Interlude: Logical Formalisms

5.1 A Logical Framework: Twelf

5.1.1 Logical Frameworks

Logical Framework is a meta-language in which one can define and reason about formal properties of different logics and programming languages. Logical Frameworks have been designed for high-level specification of languages in logic and computer science. LF is the abbreviation of LF Logical Framework which is introduced by Harper, Honsell and Plotkin [21].

Formalizing a language or deductive system within a logical framework considerably simplifies reasoning about certain properties of such languages. We refer to a language that we formalize in LF as an *object language*. The techniques of formalization consist of three common stages. The first stage is the representation of the *abstract syntax* for the object language. For example, if we are interested in to represent boolean types, we first need to specify the languages of expressions and types of boolean system. The second stage is the representation of the language *semantics*. There are two types of it. One of them is static semantics which can be constructed with the notion of value and type system. The other is dynamic semantics which can be constructed with operational semantics. The third stage is the representation of *meta-theory* of the object language where the proof of type preservation is the best example to represent it.

5.1.2 Twelf System

Twelf depends on the LF type theory and the judgements-as-types methodology for specification, a constraint logic programming interpreter for implementation [32] and the meta-logic M_2 for reasoning about object languages encoded in LF [46]. It is the extension and reimplementation of the Elf system [33]. Furthermore, it is written in Standard ML and can run under SML of New Jersey on Unix and Windows platforms.

The Twelf System is a tool for experimentation in the theory of programming languages and logics. It supports diverse tasks which are *specification* of object languages and their semantics, implementation of *algorithms* exploiting object-language expressions and deductions, and formal development of the *meta-theory* of an object language.

Twelf uses the representation methodology and underlying type theory of the LF logical framework. Expressions are represented as LF objects employing the technique of *higher-order abstract syntax* abbreviated as HOAS. This technique has an important advantage that variables bound in the object language will be bound by λ in the meta-language. In other words, variables of an object language are mapped to variables in the meta-language. As a result of this property, we do not need to program a new operation for renaming of bound variables or capture-avoiding substitution which are directly supported by the framework.

LF employs the *judgements-as-types* representation technique for semantic specification. This means that a derivation is coded as an object whose type represents the judgement it establishes. Therefore, just type-checking the representation of a derivation in the logical framework provides us to check the correctness of it.

Twelf supports the implementation of algorithms to exploit expressions or derivations by a constraint logic programming interpretation of LF signatures. The operational semantics is based on backtracking and goal-directed search for an object of a given type.

Twelf provides higher-level judgements and the meta-logic M_2 to express the meta-theory of deductive systems.

5.1.3 Main Features of Twelf as a Logical Framework

The choice of using Twelf system as a Logical Framework has numerous advantages. The main advantage is that it is written and tested by great community who has been implemented several case studies, written too many documents and tutorials about it. Because of these properties, it has been chosen several people who are working on Logical Framework and hence it is seen as pioneer in the field of Logical Frameworks.

Since Twelf System is based on LF system, it carries the features of LF. Furthermore, it leads to several features to increase the user's capability on using it. The main features of the Twelf logical framework can be summarized as follows:

- Twelf is a tool to mechanize the metatheory.
- It represents the syntax of the formal system.
- It represents the judgements of the logic.
- It relates among the elements of the syntax.
- It states theorems about the judgements.
- It proves these theorems.

Moreover, Twelf includes a type checker for the LF, an operation interpretation of LF as proof search procedure, a theorem checker about represented logics and a theorem prover. However, the theorem prover is not satisfactory that it does include proof finding.

5.1.4 An Example for Twelf

At the beginning of this chapter, we have described the features of Twelf as a tool that assists people designing deductive systems. If someone has an idea to design a programming language, first he should formalize the syntax of the language and give the meaning of syntax by defining some judgements for it in the LF logical framework. He also uses Twelf to check proofs of theorems. Moreover, he uses Twelf to run language definition itself to try out some examples. In this section, we would like to show how a very simple deductive system (natural numbers with addition) will be introduced according to these formalization techniques. This example offers a trivial tutorial about defining deductive system using Twelf Logical Framework.

Informally, the syntax of the natural numbers is defined by the following grammar:

$$n ::= \mathsf{zero} \mid \mathsf{succ}(n)$$

That is, zero is a natural number, and if n is a natural number, then $\mathsf{succ}(n)$ is as well.

We represent an object language of natural numbers in LF by writing an LF *signature*. A signature declares type and term constants. For example, this LF signature defines a type representing the natural numbers:

```
nat : type.
z   : nat.
s   : nat -> nat.
```

From the above signature, the LF type `nat` classifies the LF representations of natural numbers. The LF constant `z` corresponds to zero and the LF constant `s` corresponds to succ. The signature declares that `z` has type `nat`, which makes sense because zero is a natural number.

The signature declares that `s` has function type `nat -> nat`. An LF term of function type can be applied to another LF term of the appropriate type to form a new term. For instance, the constant `s` can be applied to the constant `z` to form the term `s z` representing the number $\mathsf{succ}(\mathsf{zero})$. Then `s` can be applied to this term to form `s (s z)`, and so on. An informal natural number $\mathsf{succ}(n)$ is represented by the LF term `s N` where n is represented by `N`.

So far, we have seen how we can represent a syntax of deductive system, now we start to represent the judgements. We first state the judgement that a number is even. At the beginning, we need to make informal definition about even numbers:

$$\frac{}{\mathsf{even}(\mathsf{zero})} \qquad \frac{\mathsf{even}(n)}{\mathsf{even}(\mathsf{succ}(\mathsf{succ}(n)))}$$

where zero is an even number and the judgement $\mathsf{even}(n)$ holds when n is even.

Previously, we have represented the syntax of an object language as the individuals of an LF type. Mainly, we also represent the judgement of an object language with an LF type, where the individuals of type correspond exactly to the derivations of judgement. However, the current simply-typed LF is not adequate to represent judgements. Therefore, we need to generalize our simply-typed LF to *dependently-typed* LF. A dependently-typed language is convenient because object language judgements are parameterized by their subjects. For instance, $\mathsf{even}(n)$ is parameterized by the number n being judged to be even. For that reason, in order to represent judgements as LF types, we should consider LF types that are parameterized by the subjects of object language judgements. In other words, the subjects of judgements are the syntax of the object language and LF types of judgement representation are parameterized by these subjects. Since the judgement subjects are represented as LF terms, in order to represent judgements as LF types, it is obvious that families of LF types are parameterized by LF terms.

For example, the following signature represents the judgement even(n):

```
even    : nat -> type.
even-z  : even z.
even-s  : {N:nat} even N -> even (s (s N)).
```

The first declaration states that `even` is a family of types indexed by a `nat` which is defined previously as an LF type of our syntax.

The first term constant, `even-z`, has type `even z`. This constant represents the derivation that consists of the first inference rule above, which concludes even(zero).

The second term constant `even-s`, corresponds to the second inference rule above. It states that for any n, it constructs a derivation of even(succ(succ(n))) from a derivation of even(n). As we have mentioned before, in order to encode this inference rule, the constant `even-s` is given a dependent function type.

The syntax `x:A1 A2` represents a dependent function type, which is a generalization of the ordinary function type `A1 -> A2` that allows the argument term `x` to appear in `A2`. We write the ordinary function type `->` as a synonym when the argument is not free in the result type, for example `A1 -> A2` means `_:A1 A2`. Just as with the ordinary function type, LF terms of dependent function type can be applied to a term of the argument type to form a new term.

The dependent function type is used to bind the variable `N` in the type `(even N -> even (s (s N)))`, expressing that the inference rule is schematic in n. When a term of dependent function type is applied to an argument, the argument is substituted into the result type. For example, the term `even-s z` has type `even z -> even (s (s z))` where it is the specialization of the inference rule to $n =$

zero. Thus, the term

```
even-s z even-z
```

represents the derivation

$$\dfrac{\dfrac{}{\mathsf{even}(\mathsf{zero})}}{\mathsf{even}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))}$$

The other judgement addition can be represented as relation of two natural numbers to their sum:

$$\dfrac{}{\mathsf{plus}(\mathsf{zero}, n_1, n_1)} \qquad \dfrac{\mathsf{plus}(n_1, n_2, n_3)}{\mathsf{plus}(\mathsf{succ}(n_1), n_2, \mathsf{succ}(n_3))}$$

This judgement defines addition by induction on the first argument and it is represented by the following LF signature:

```
plus   : nat -> nat -> nat -> type.
plus-z : {N1:nat} plus z N1 N1.
plus-s : {N1:nat} {N2:nat} {N3:nat}
         plus N1 N2 N3 -> plus (s N1) N2 (s N3).
```

The type family `plus` is indexed by three terms of type `nat` because the informal judgement has three parameters. The constants correspond to the two inference rules. For example,

```
plus-s (s z) (s z) (s (s z))
       (plus-s z (s z) (s z)
               (plus-z (s z)))
```

which has type `plus (s (s z)) (s z) (s (s (s z)))`, represents a derivation that $2 + 1 = 3$.

We sometimes would like to reason about our theories, for example `plus` relation is commutative. Especially whenever we have a derivation of `plus N1 N2 N3`, there is a different derivation of `plus N2 N1 N3`. We therefore ask ourselves the same question: Is the relation *total relation*? Totality relations are one class of statements about LF type families. A totality assertion for a type family is specified by allocating some index positions as inputs and others positions as outputs. We call this specification the mode of the totality assertion. Given a mode specification, the totality assertion asserts that for all inputs, there exist outputs that stand in the relation. Using `plus` as an example, if we allocate the first two positions as inputs and the third position as an output, this specifies the following totality assertion:

For all `N1:nat` and `N2:nat`, there exist `N3:nat` and `D:plus N1 N2 N3`.

For all that, Twelf can check totality of relations. Mechanizing such assertions brings us that Twelf can verify totality assertions about LF type families on its own. Twelf can do this by the following declarations:

- `%mode` => Which arguments are inputs which are outputs.
- `%worlds` => The theorems live in which context.
- `%total` => Which is the induction argument.

Using these declarations, we can show an example totality assertion about `plus` type family such that:

```
plus_ident : {N : nat} plus N z N -> type.
%mode plus_ident +N -D.
```

```
-: plus_ident z plus-z.
-: plus_ident (s N) (plus-s D) <- plus_ident N (plus-s D).

%worlds () (plus_ident _ _).
%total N (plus_ident N _).
```

5.2 Logic Programming through Prolog

There are two types programming languages in computer science. One of them is named as *imperative programming languages* where programs are made up of commands to be executed. A commonly used examples for this type of programming languages are Pascal, Java and C. However, the other one is *declarative programming languages* where programs are made up of definitions and statements about the problem to be solved. Prolog and Lisp are broadly used imperative languages.

In logic programming, a program consists of a collection of statements expressed as formulas in symbolic logic. There are inference rules that allow a new true formula from old true formulas. When a computer executes a logic program, it uses inference rules to derive new formulas from the ones given in the program until it finds one that expresses the solution of the problem. In other words, logic programming systems solve goals by systematically searching for a way to derive the answer from the program.

5.2.1 Constraint Logic Programming

Constraint Logic Programming is a constraint programming in which the logical programming is extended to include concepts from constraint satisfaction that is the process of finding solutions from set of constraints [24, 28, 23]. The following can be an example of CLP:

$$A(X,Y) \text{ :- } (X+Y>0)\ B(X)\ C(Y)$$

which means the satisfaction of $A(X, Y)$ is dependent to the satisfaction of the constraint $(X + Y > 0)$, and the satisfaction of both $B(X)$ and $C(Y)$.

A goal in constraint logic programming may include constraints in addition to the literals. A proof for a goal is composed of the clauses that embody the satisfiable constraints and the literals that can be proved by the other clauses. The interpreter does the execution and the interpreter tries to prove the goal by starting from the goal and recursively scanning the clauses. Since the satisfiability of a constraint cannot always be determined when the constraint is encountered, constraints encountered during the scan are stored in the constraint store. If the set is unsatisfiable after addition of some constraints, the interpreter will backtrack to try to find out the satisfiable constraints from other clauses.

The semantics of a constraint logic programs can be defined in terms of the interpreter which maintains a pair $< G, S >$ during execution. The pair consists of two letters where G infers the current goal and S is called the constraint store. The current goal contains the literals that the interpreter is trying to prove and may contain the constraints to satisfy that it is trying to satisfy. The constraint store contains the constraints that the interpreter is sure that all the constraints are satisfiable with each other.

Initially the current goal is the main goal and the constraint store is empty. The interpreter removes the first element from the current goal and starts to analyse this element. This analysis is continued until there is a failure or a successful termination. During the analysis the new literals may be added to the current goal or the constraints may be added to the constraint store. The addition of the constraints to the constraint store may introduce the unsatisfaction of the constraints in constraint store. When this happens, the interpreter should backtrack to the position where the constraints are satisfied. The successful termination is generated when the current goal is empty and all the constraints in constraint store are satisfiable.

Every time a constraint is added to the constraint store, the unsatisfaction of the constraints is checked. The process of checking the unsatisfaction of the constraint would be inefficient. Rather than pursuing such method, the incomplete satisfiability checker can be used. It rewrites the constraints in the constraint store to their equivalent simpler form. This is incomplete because it rarely proves the satisfiability of unsatisfiable constraints in constraint store.

The semantics of constrained logic programming is defined in terms of derivations. The following notation shows the transition, which means one-step execution: $< G, S >$ -> $< G', S' >$. Such a transition can be possible in three cases. A constraint element c in G, can be moved from to goal to the constraint store. $G' = G/\{c\}$, and $S' = S \cup \{c\}$.

5.2.2 SWI-Prolog: A Prolog Enviroment

SWI-Prolog is a comprehensive software Prolog environment which offers the feature to debug and trace of a Prolog program using its graphics toolkit XPCE. XCPE allows to stop the execution of a program at a demanded point and shows all called predicates with a tree structure. This is not only the benefit of SWI-Prolog. In order to realize our purpose about constraint solvers. It provides libraries for Constraint Handling Rules (CHR), and Constraint Logic Programming (CLP(Q, R)). CLP(Q, R) solves linear equations over rational or real valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects implied equations, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations, and provides for linear optimization. Some other important features of SWI-Prolog are:

- It provides a flexible and fast interface to the C and C++ programming languages. The interface allows for calling both-ways, and embedding of the SWI-Prolog kernel in C/C++ projects.
- It is portable to many platforms, including almost all Unix/Linux platforms, all Windows distributions and MacOS X with both 32-bits and 64-bits support.
- It provides multithreading support.

- It has unit testing support through PlUnit and literate programming support through PlDoc.
- It handles UNICODE character set.

5.2.2.1 ileanSeP: An Intuitionistic Sequent Theorem Prover

ileanSeP is implemented by Jens Otten and it is a Prolog program that implements a very compact theorem prover for first-order intuitionistic logic which preserves the properties of soundness and completeness[1]. It is based on an intuitionistic sequent calculus using free term-variables and skolemization. Inference rules are defined in a modular way which makes modifications to the calculus easily. This property makes us possible to modify current sequent prover with our desired constrained extension. First of all, we give the implementation details of ileanSeP to make clear some concepts like skolemization. The process of removing all the existential quantifiers from a formula is known as Skolemization. The Herbrand theorem states that a formula F of first-order logic can be transformed into a quantifier-free formula F_H such that F is provable if and only if some finite disjunction of instances of F_H can be proved in classical propositional logic. The use of Herbrand functions in proof search is mostly called Skolemization. However, Herbrand theorem cannot be applied directly to most logics, including intuitionistic, linear, and modal logics, which typically lack the transformations needed to convert F to F_H. Hence, Shankar [47] presented an optimized form of dynamic Skolemization for the intuitionistic sequent calculus.

The ileanSeP prover is invoked with `prove`(F) where F is a first-order formula. The logical connectives are expressed by "`~`" (negation), "`,`" (conjunction), "`;`" (disjunction), "`=>`" (implication), " `<=>`" (equivalence); quantifiers are expressed by "`all X:`" (universal) and "`ex X:`" (existential).

The description of the implementation is divided into the parts path checking and unification. In order to search proofs of a given formula, the two predicates `prove` and `fml` are employed. The predicate `prove` gets one argument which

[1]Jens Otten's website for ileanSeP: http://www.leancop.de/ileansep/

takes a formula and initially starts the proof checking process. The predicate `fml` is used to specify the particular characteristics of each logical connective or quantifier:

```
fml(F,Pol,inv/noninv,L1,R1,L2,R2,S,FreeV,V,Cpy,Cpy1).
```

In `fml` predicate, `F` is the non-atomic first order formula. `Pol` is the polarization of the formula which takes `1` for left rules and takes `0` for right rules. `inv/noninv` is used for specifying the formula to be invertible or noninvertible. `L1` and `R1` are the left and right rules of first premise and `L2` and `R2` are the left and right rules of second premise after decomposing formula `F`. `S` is the unique position in the formula tree and `FreeV` is the lists of free quantifier-variables of the current branch in proof tree. `V` is the new free variable revealed after applying current formula decomposition. `Cpy` contains a term which has to be copied later on and bound to the parameter `Cpy1`.

The followings are the left and right rules of the $\wedge$ connective of Gentzen's sequent calculus of intuitionistic logic:

$$\frac{\Gamma \Rightarrow A \qquad \Gamma \Rightarrow B}{\Gamma \Rightarrow A \wedge B}\ \wedge R \qquad\qquad \frac{\Gamma, A, B \Rightarrow C}{\Gamma, (A \wedge B) \Rightarrow C}\ \wedge L$$

The corresponding encoding for the above connective within ileanSeP is as follows:

```
fml((A,B), 0, inv, [],    [A], [], [B], _, _, [], [], [] ).
fml((A,B), 1, inv, [A,B], [],  [], [],  _, _, [], [], [] ).
```

A free variable is introduced by quantifiers. Therefore, explaining one of the quantifiers is beneficial to understand the usage of free variables within ileanSeP. The followings are the existential quantifier of Gentzen's sequent calculus of intuitionistic logic:

$$\frac{\Gamma \Rightarrow [t/x]A}{\Gamma \Rightarrow \exists x.A}\ \exists R \qquad\qquad \frac{\Gamma, [a/x]A \Rightarrow C}{\Gamma, \exists x.A \Rightarrow C}\ \exists L^a$$

For the right rule of existential quantifier a new term t is introduced and variable x in A are replaced by this new term. However, we need to replace x with a new variable a in A for the left rule. As a result, this new variable must be a variable which is not defined before and this variable must not be unified with other variables or terms except itself during unification process. Therefore, we replace the quantified variable by the Skolem-term `S^FV` in the current formula `F` where `S` is the position of `F` (in the formula tree), `FV` is its free variable list. Thus, the corresponding encoding is as follows:

```
fml(ex X:A, 0, nin, [],  [C], [], [], _, _, [Y],  X:A,    Y:C).
fml(ex X:A, 1, inv, [C], [],  [], [], S, FV, [], (X,A), (S^FV,C)).
```

The difference between implication rule and other rules except universal quantifier in Gentzen's intuitionistic sequent calculus is that the left implication rule resumes in the left side of the first premise:

$$\frac{\Gamma, A \Rightarrow B}{\Gamma \Rightarrow A \supset B}\ \supset R \qquad\qquad \frac{\Gamma, (A \supset B) \Rightarrow A \qquad \Gamma, B \Rightarrow C}{\Gamma, (A \supset B) \Rightarrow C}\ \supset L$$

Viewing from bottom-up, the left rule does not shrink compared to other rules. For that reason, Prolog program can continuously call the same rule. Therefore, this has to be considered in such a way:

```
fml((A=>B), 1, nin, [(A=>B)], [C], [D], [], _, _, [!], A:B, C:D).
fml((A=>B), 0, inv, [A],      [B], [],  [], _, _, [],  [],  [] ).
```

An anonymous "`!`" added to `FreeV` whose size will later be compared to `VarLim`. When comparison equals, then the program backtracks to select another matched rule in some point. This modification prevents from calling same rule again and again.

There are totally 15 clauses that define the corresponding characteristics of the intuitionistic connectives and quantifiers.

The predicate actually performing the proof search is:

```
prove(Left,Right,S,FreeV,VarLim).
```

It succeeds if unification finds correspondence between `Left` and `Right` formulas after decomposing them. The parameter `VarLim` is a positive integer used to initiate backtracking in order to obtain completeness within Prolog's depth search. The other parameters have been explained before.

Given a formula `F`, the program starts to search a proof using predicate `prove` and assigns formula `F` to argument `Right`, `1` to argument `S`, `1` to argument `VarLim` and gives empty set to the other two arguments. It initially starts to inspect to match whether the left rule and right rule corresponds to any invertible rule respectively. It later tries non-invertible rules to match. If there is no match, it means atomic rules are in both left and right side and they are conveyed to unification process. If there is a match, then the program checks whether the depth-bound `VarLim` is reached. If depth-bound is reached, then the program backtracks; otherwise the program decomposes the formula with a rule of matched connective.

An example proof tree for $\forall y.P(y) \supset \exists x.P(x)$ is as shown:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P(T_1), \forall y.P(y) \Rightarrow P(T_2)}\ \texttt{init}}{P(T_1), \forall y.P(y) \Rightarrow \exists x.P(x)}\ \exists R}{\forall y.P(y) \Rightarrow \exists x.P(x)}\ \forall L}{. \Rightarrow \forall y.P(y) \supset \exists x.P(x)}\ \supset R$$

In order to search proof within ileanSeP, one has to invoke the program in the following way:

```
prove( all Y:(p(Y)) =>  ex X:(p(X)) ).
```

The program takes the formula and assigns it to right formula set while keeping empty set for left formula set:

```
LEFT: [],   RIGHT: [all Y:(p(Y)) =>  ex X:(p(X))]
```

The program starts to build a proof tree of given example. First of all, given formula matches the right implication rule, therefore left formulas and right formulas are decomposed and added to left and right formula set respectively:

```
LEFT: [all _G590:p(_G590)],   RIGHT: [ex _G597:p(_G597)]
```

After that, the program applies the noninvertible left rule of universal quantifier to the left formula set. The result is added to the left formula set and the right formula set remains unchanged:

```
LEFT: [p(_G748), all _G590:p(_G590)],   RIGHT: [ex _G597:p(_G597)]
```

It is time to apply the noninvertible right rule of existential quantifier to the right formula set, the resultant left and right formula set is such that:

```
LEFT: [p(_G748), all _G590:p(_G590)],   RIGHT: [p(_G779)]
```

Since, both right and left formula sets have atomic expressions, the unification process starts. This process ensures that all of the expression in right formula set are conformed with the left ones where `unify_with_occurs_check` is used for that purpose. It is the specialized Prolog builtin to realize unification process:

```
unify_with_occurs_check([p(_G748)], [p(_G779)]).
```

Eventually, the program verifies that given formula is true and builds its proof tree with given order.

Chapter 6

Implementations of CILL

6.1 Twelf

6.1.1 Constructing CIL in Twelf

In Chapter 5, we have defined the properties and features of Twelf and given a simple example using it. Twelf provides the formalization of defining and reasoning about programming languages and logics. Since we are considering to implement Constrained Intuitionistic Linear Logic(CILL), Twelf can be an important candidate. However, for simplicity we first implement Constrained Intuitionistic Logic(CIL) which is extended from the Frank Pfenning's language of formulas for intuitionistic predicate calculus with constrained domain. We later discuss why we start with CIL rather than CILL.

In LF, we encode the syntax of our sequent calculus as follows:

```
i : type.  % individuals
o : type.  % formulas
c : type.  % constraints
d : type.  % constraint domain
```

```
EC : d.    % empty constraint-context

and    : o -> o -> o.
imp    : o -> o -> o.
or     : o -> o -> o.
not    : o -> o.
true   : o.
false  : o.
forall : (i -> o) -> o.
exists : (i -> o) -> o.

addom  : c -> d -> d.
cimp   : c -> o -> o.
cand   : c -> o -> o.

>>     : integer -> integer -> c.
```

Here, the logic only supports $\top$, $\bot$, $\wedge$, $\vee$, $\supset$ and existential quantifiers. In addition to this, our constrained extension $\supset_c$ and $\wedge_c$ are supported within this language. We have two auxiliary terms to help representing the constrained syntax. One of them is `EC` that represents the empty constraint context and it is needed at the beginning of our proof search when our context is initially empty. The other one is `addom`, that is for inserting a single constraint to our constrained context. From our declaration, we only construct such example constraints: `x >> 3`, `x >> y` or `3 >> 4` where `x` and `y` are individuals of our constraints that are introduced by existential quantifiers.

Now, we define our judgements of our sequent calculus in LF methodology.

```
hyp   : o -> type.  % Hypotheses (left)
conc  : o -> type.  % Conclusion (right)
chyp  : d -> type.  % Constraint hypothesis
cconc : c -> type.  % Constraint conclusion
```

The sequent is represented as the judgement `conc`, which is hypothetical in hypotheses `hyp`.

$$\frac{}{\Psi \mid (\Gamma, A) \Rightarrow A}\ \textsf{axiom}$$

A rule `axiom` allows us to use a hypothesis to satisfy a conclusion.

```
axiom : (hyp A -> conc A).
```

Moreover, we have two other judgements `chyp` and `cconc` to represent our constraint domain. If we have a condition such as $D \models C$ then the correspondent sequent will be `chyp(D) -> cconc(C)`.

Now, let us look at the sequent calculus of ordinary intuitionistic logic:

$$\frac{\Psi \mid \Gamma \Rightarrow A \qquad \Psi \mid \Gamma \Rightarrow B}{\Psi \mid \Gamma \Rightarrow A \wedge B}\ \wedge R$$

$$\frac{\Psi \mid (\Gamma, A) \Rightarrow C}{\Psi \mid (\Gamma, (A \wedge B)) \Rightarrow C}\ \wedge L_1 \qquad \frac{\Psi \mid (\Gamma, B) \Rightarrow C}{\Psi \mid (\Gamma, (A \wedge B)) \Rightarrow C}\ \wedge L_2$$

$$\frac{\Psi \mid (\Gamma, A) \Rightarrow B}{\Psi \mid \Gamma \Rightarrow A \supset B}\ \supset R \qquad \frac{\Psi \mid \Gamma \Rightarrow A \qquad \Psi \mid (\Gamma, B) \Rightarrow C}{\Psi \mid (\Gamma, (A \supset B)) \Rightarrow C}\ \supset L$$

$$\frac{\Psi \mid \Gamma \Rightarrow A}{\Psi \mid \Gamma \Rightarrow A \vee B}\ \vee R_1 \qquad \frac{\Psi \mid \Gamma \Rightarrow B}{\Psi \mid \Gamma \Rightarrow A \vee B}\ \vee R_2$$

$$\frac{\Psi \mid (\Gamma, A) \Rightarrow C \qquad \Psi \mid (\Gamma, B) \Rightarrow C}{\Psi \mid (\Gamma, (A \vee B)) \Rightarrow C}\ \vee L$$

$$\frac{}{\Psi \mid \Gamma \Rightarrow \top}\ \top R \qquad \frac{}{\Psi \mid (\Gamma, \bot) \Rightarrow C}\ \bot L$$

$$\frac{\Psi \mid \Gamma \Rightarrow [a/x]A}{\Psi \mid \Gamma \Rightarrow \forall x.A}\ \forall R^a \qquad \frac{\Psi \mid (\Gamma, [t/x]A) \Rightarrow C}{\Psi \mid (\Gamma, \forall x.A) \Rightarrow C}\ \forall L$$

$$\frac{\Psi \mid \Gamma \Rightarrow [t/x]A}{\Psi \mid \Gamma \Rightarrow \exists x.A}\ \exists R \qquad \frac{\Psi \mid (\Gamma, [a/x]A) \Rightarrow C}{\Psi \mid \Gamma, \exists x.A \Rightarrow C}\ \exists L^a$$

The encoded inference rules in Twelf will be such that:

```
andr  : conc A ->
        conc B ->
        conc (A and B).

andl1 : (hyp A -> conc C) ->
        (hyp (A and B) -> conc C).

andl2 : (hyp B -> conc C) ->
        (hyp (A and B) -> conc C).

impr  : (hyp A -> conc B) ->
        conc (A imp B).

impl  : conc A ->
        (hyp B -> conc C) ->
        (hyp (A imp B) -> conc C).

orr1  : conc A ->
        conc (A or B).

orr2  : conc B ->
```

```
        conc (A or B).

orl    : (hyp A -> conc C) ->
        (hyp B -> conc C) ->
        (hyp (A or B) -> conc C).

truer : conc (true).

falsel : (hyp (false) -> conc C).

forallr : ({a:i} conc (A a)) ->
          conc (forall A).

foralll : {T:i} (hyp (A T) -> conc C) ->
          (hyp (forall A) -> conc C).

existsr : {T:i} conc (A T) ->
          conc (exists A).

existsl : ({a:i} hyp (A a) -> conc C) ->
          (hyp (exists A) -> conc C).
```

We need to show inference rules of constraint extension such that:

$$\frac{(\Psi, R) \mid \Gamma \Rightarrow A}{\Psi \mid \Gamma \Rightarrow R \supset_c A} \supset_c R \qquad \frac{\Psi \models R \qquad \Psi \mid \Gamma, A \Rightarrow C}{\Psi \mid (\Gamma, R \supset_c A) \Rightarrow C} \supset_c L$$

$$\frac{\Psi \models R \qquad \Psi \mid \Gamma \Rightarrow A}{\Psi \mid \Gamma \Rightarrow R \wedge_c A} \wedge_c R \qquad \frac{(\Psi, R) \mid (\Gamma, A) \Rightarrow C}{\Psi \mid (\Gamma, R \wedge_c A) \Rightarrow C} \wedge_c L$$

For all that, the following part describes the encoded constraint sequents within intuitionistic logic.

```
cimpl   : ( chyp D -> cconc R ) ->
          ( chyp D -> hyp A -> conc C) ->
          ( chyp D -> hyp (R cimp A) -> conc C ).

cimpr   : ( chyp (addom R D) -> conc A ) ->
          ( chyp D -> conc (R cimp A) ).

candr   : ( chyp D -> cconc R ) ->
          ( chyp D -> conc A ) ->
          ( chyp D -> conc (R cand A) ).

candl   : ( chyp (addom R D) -> hyp A -> conc C ) ->
          ( chyp D -> hyp (R cand A) -> conc C ).
```

So far, we have encoded our sequent calculus of constrained intuitionistic logic into Twelf. But we need a rule for constraint context to terminate the proof search just like the rule `axiom`. Since Twelf lacks constrained logic programming, we could not find out the consistency of the constraint domain. Therefore, we say that whenever Twelf system encounters a constraint hypothesis R, it will satisfy a trivial conclusion S. This means that all given constraints are satisfiable although in reality they may not. A rule `triv` allows us to use a constraint hypothesis to satisfy a constraint conclusion.

```
triv    : chyp R -> cconc S.
```

In Chapter 5, we have said that Twelf introduces a systematic methodology for representing deductive systems. This methodology is often called the judgements as types principle, because a judgement in a deductive system is represented as an LF type family classifying derivations of the judgement. It offers a great opportunity that derivations in a deductive system can be checked just by type checking their LF representations. Therefore, we need to give how we can do type-checking within Twelf. For simplicity, we below just write down the grammar that describes only the type checking part of Twelf.

```
sig  ::=                    % Empty signature
       | decl sig           % Constant declaration

decl ::= _ = term.          % for type-checking
decl ::= _ : term = term.   % for type-checking

term ::= type               % type
       | id                 % variable x or constant a or c
       | term -> term       % A -> B
       | term <- term       % A <- B, same as B -> A
       | {id : term} term   % Pi x:A. K  or  Pi x:A. B
       | [id : term] term   % lambda x:A. B  or  lambda x:A. M
       | term term          % A M  or  M N
       | term : term        % explicit type ascription
       | _                  % hole, to be filled by term
                              reconstruction
       | {id} term          % same as {id:_} term
       | [id] term          % same as [id:_] term
```

The grammar above defines the non-terminals `sig`, `decl`, `term` and uses the terminal `id` which stands for identifers. Here, we do not give the whole grammar of Twelf, since we only intend to focus on the details of type checking. The important part is that Twelf does not directly support proof searching. It only checks the proofs by classifying derivations of judgements. Hence, one has to write the judgements with its derivations in a signature to ask to Twelf whether derivations are deducable or not. The following Twelf signature is an example type-checking signature that uses our encoded constraint intuitionistic logic representation without constraints.

```
_ = (B’ or A’) : o.

_ =
  ([A’:o] [B’:o] [h1:hyp (A’ or B’)]
     orl ( [h2:hyp A’] orr2 (axiom h2) )
         ( [h3:hyp B’] orr1 (axiom h3) )
         h1 )
:
{A’:o} {B’:o} {h1:hyp (A’ or B’)}
conc (B’ or A’).
```

The above signature shows that the hypothesis `A’ or B’` concludes to `B’ or A’` while having hypotheses `A’` and `B’`.

The following is another type-checking signature that uses constraints of our representation.

```
_ = R cimp ( R cand (A imp A) ) : o.

_ =
( [A:o] [R:c] [d1:chyp EC]
    cimpr ( [d2:chyp (addom R EC)]
            ( candr ( [d3:chyp (addom R EC)] triv d3 )
                    ( [d4:chyp (addom R EC)]
                      ( impr ([h1:hyp A] axiom h1) ) )
                    d2 ) )
  d1 )
:
{A:o} {R:c} {d1:chyp EC}
conc ( R cimp ( R cand (A imp A) ) ).
```

We have mentioned that Twelf lacks constraint logic programming. This property prevents us from implementing our desired constrained intuitionistic linear logic in Twelf. Therefore, we stop going on with Twelf and leaving it just implementing CIL. Miraculously, Prolog provides both constraint logic programming and backward proof search. We describe in next section that Prolog will offer great properties to implement CILL.

6.2 Prolog

In previous chapter, we have mentioned that ileanSeP introduces a theorem prover for first-order intiutionistic logic. Moreover, SWI-Prolog offers a Prolog environment including constraint handling rules. Therefore, we can initially implement constrained intuitionistic logic by extending ileanSeP with constraints.

6.2.1 cileanSeP: Extension of ileanSeP for First-Order Constrained Intuitionistic Logic

As we have mentioned before, SWI-Prolog offers a great property to handle constraints named as CLP(Q, R). {Constraint} predicate is used for constraint solver to add given Constraint to the constraint store. For the sake of simplicity, we below give only the part used for the implementation:

```
Constraint ::=  C
                | C, C            conjunction

         C ::=  Expr = Expr       equation
                | Expr <  Expr    strict inequation
                | Expr >  Expr    strict inequation
                | Expr =<  Expr   nonstrict inequation
                | Expr >= Expr    nonstrict inequation

      Expr ::=  variable          Prolog variable
                | number          floating point or integer
```

The other predicate in CLP(Q, R) for constraint handling is `entailed`(Constraint). It succeeds if and only if the linear Constraint is entailed by the current constraint store. This predicate does not change the state of the constraint store.

After giving constraint handling rules and predicates in SWI-Prolog, it is time to find out how we can use these predicates in order to extend ileanSeP with constraint domain. Since constraint intuitionistic logic introduces a new context, we need to add two arguments to our formula definition(`fml`). One of the arguments is the constraint to be inserted to the constraint context, and the other one is the constraint to be solved:

```
fml(..., ConstToBeAdded, ConstToBeSolved).
```

Another modification is done on `prove` predicate in order to pass the constraint context throughout the proof tree. As you recall, splitted formulas are always called by `prove` predicate until only atomic formulas left. Therefore, `prove` predicate is extended with one argument to keep track of constraint context:

```
prove(..., ConstContext).
```

If a given formula matches with a pattern that a constraint is insterted to the constraint context, then predicate { } is used to add this constraint into the constraint store. Likewise, if a given formula matches with a pattern that a constraint is solved, then predicate `entailed` is used for solving this constraint using current constraint store. Thus, two predicates, `solve_const` and `check_const` are defined. The former takes two arguments which are the constraint context and the constraint to be solved respectively. The latter also takes two arguments which are the constraint context and the constraint to be inserted respectively. This predicate checks consistency of constraint store with a given constraint. It is used for deciding to add the constraint to the constraint store or backtrack to the point that consistency is satisfied.

```
solve_const([], [Right]) :- entailed(Right).
solve_const([Head|Const], [Right]) :- {Head},
                                      solve_const(Const, [Right]).

check_const([], [Right]) :- {Right}.
check_const([Head|Const], [Right]) :- {Head},
                                      check_const(Const, [Right]).
```

Recall that we have two additional connectives for constraints, one is constraint implication:

$$\frac{(\Psi, D) \mid \Gamma \Rightarrow A}{\Psi \mid \Gamma \Rightarrow D \supset_c A} \supset_c R \qquad \frac{\Psi \models D \qquad \Psi \mid (\Gamma, A) \Rightarrow C}{\Psi \mid \Gamma, D \supset_c A \Rightarrow C} \supset_c L.$$

Encoding constraint implication in cileanSeP will be such as:

```
fml((D cimp A), 1,inv,[A], [], [], [], _, _,[], [], [],  [], [D]).
fml((D cimp A), 0,inv,[], [A], [], [], _, _,[], [], [], [D], []).
```

As you see, while the left rule introduces the constraint to be solved, the right rule introduces the constraint to be added to constraint context.

The constraint conjunction was defined before:

$$\frac{\Psi \models D \qquad \Psi \mid \Gamma \Rightarrow A}{\Psi \mid \Gamma \Rightarrow D \wedge_c A} \wedge_c R \qquad \frac{(\Psi, D) \mid (\Gamma, A) \Rightarrow C}{\Psi \mid \Gamma, D \wedge_c A \Rightarrow C} \wedge_c L$$

Following enconding shows the enconding within cileanSeP of constraint conjunction:

```
fml((D cand A), 1,inv,[A], [], [], [], _, _,[], [], [], [D], []).
fml((D cand A), 0,inv,[], [A], [], [], _, _,[], [], [],  [], [D]).
```

The important question to ask ourselves is where we will handle constraints in our program. Well, the answer of this question is quite simple, we will handle these constraints if any constraint is introduced after pattern matching of a given formula. In other words, if a given formula matches with the connectives of constraint implication or constraint conjunction, then constraint solver intervenes the program to handle constraints with a given Prolog syntax:

```
( AddConst=[] -> NewContext=ConstContext;
( check_const(ConstContext, AddConst),
  append(ConstContext, AddConst, NewContext) ) ),
( SolveConst=[] -> true;
( solve_const(NewContext, SolveConst) ) )
```

The code above states that when constraint to be added is introduced; first, it will be checked whether constraint context loses its consistency, then it will be inserted to the constraint store. However, when constraint to be solved is introduced, then it will be solved by current constraint context.

Now, it is a good practice to give a simple example about constraints in cileanSeP. When program is invoked by

```
   prove( (p => p) ).
```

then the program concludes that given formula is satisfiable. However, the following example is not satisfiable because of the inconsistency of constraint context.

```
   prove( (3 > 4) cimp (p => p) ).
```

6.2.1.1 Quantification problem about constraints

In first-order logic, quantification always involves all elements of the universe. When one uses a type of constrained logic, it will therefore be a part of the universe. In such a universe, constrained logic provides restricted quantifiers constraining variables by a formula. A formula F preceeded by such a restricted quantifier can be read as "F holds for all or for some elements (depends on type of the quantifier) satisfying the constraint R". Since the universe of unconstrained hypotheses is extended with the universe of constraints, the unification process and skolemization technique have to be modified in the extension of intuitionistic logic with constraints.

Before describing the modification, an example clarifies the problem. The problem occurs in the left rule of existential quantifier and the right rule of universal quantifier where the parametric variable is introduced. The following is the formula to be solved:

$\exists y.((y = 0 - 1) \wedge_c \mathsf{p}(y) \Rightarrow \mathsf{p}(\text{-}1)$

In this example, the corresponding proof tree looks like that:

$$\dfrac{\dfrac{(a = 0 - 1) \mid \mathsf{p}(a) \Rightarrow \mathsf{p}(-1)}{(a = 0 - 1) \wedge_c \mathsf{p}(a) \Rightarrow \mathsf{p}(-1)}\ \wedge_c L}{\exists y.((y = 0 - 1) \wedge_c \mathsf{p}(y) \Rightarrow \mathsf{p}(-1)}\ \exists L^a$$

Since parametric variables are not unified at the unification level, the judgement $\mathsf{p}(a) \Rightarrow \mathsf{p}(-1)$ could never be satisfied in our current implementation. However, on the constraint side of the proof tree, we have a constraint that the parametric variable (a) has to be -1 $(a = 0 - 1)$. Therefore, it concludes that the judgement $\mathsf{p}(a) \Rightarrow \mathsf{p}(-1)$ is satisfied. For that reason, we need to modify our current unification process that when a parametric variable is encountered, we first examine the constraint over this variable, then unification process is started. Satisfaction of given formula directly bounds with the restriction on this variable. For instance, the $\mathsf{p}(a) \Rightarrow \mathsf{p}(-1)$ judgement will not be satisfiable if constraint solver finds out that parametric variable a is restricted as $a < 1$.

6.2.2 Simple Example in cileanSeP

A robot starting from initial point moves only one step backwards, we find out whether this robot can come one step back from the initial point and how.

For that example, we have an at predicate which takes one argument corresponding the position of robot. Initially, the robot is at 0, which encoded as $\mathsf{at}(0)$. We also need an action that the robot only moves one step backwards. This action can be encoded as follows:

$\forall x.\ (\mathsf{at}(x) \supset \exists y.((y = x - 1) \wedge_c \mathsf{at}(y)\ )$

Moreover, at(-1) is the goal state of the robot. Consequently, the following is the formula of our problem

$$\forall x.\ (\mathsf{at}(x) \supset \exists y.((y = x - 1)\wedge_c\ \mathsf{at}(y)\ ),\ \mathsf{at}(0) \Rightarrow \mathsf{at}(\text{-}1)$$

and the following syntax is used to invoke the program:

```
prove( ( all X: (at(X) => ex Y: ( (Y = X - 1) cand at(Y) ) ),
         at(0) ) => at(-1.0) ).
```

Chapter 7

Conclusions and Future Work

We have mentioned that robotic planning problems mostly include both discrete and continuous properties of the application domain. Finding plans according to the robotic behaviours and environmental structures is a vital and common research area in the context of robotics and artificial intelligence. Several researchers also deal with the automation of finding plans in either uncertain or certain environments. In this thesis, one of the main goals is to build and implement systems that are capable of finding autonomous plans according to given specifications. Therefore, we have chosen logic programming in order to find plans. In logic programming, a program consists of a collection of statements expressed as formulas in symbolic logic in which inference rules allow to deduce a new true formula from old ones. The nature of logic programming leads to find desired information using its inference rules. In other words, logic programming reduces to find proofs, which later correspond to programs, only using actual logical techniques and formalisms without additional information. Several researches select logical systems to overcome the problem of finding plans because of its consistency, soundness and completeness. However, some of these approaches cope with only specific robotic planning problems, some others just deal with the problem of discrete state changing. For that reason, there has been a need for a uniform framework.

In this thesis, we have selected Constrained Intuitionistic Linear Logic (CILL) which is initially defined by Uluç Saranlı and Frank Pfenning in their paper [45] to represent and reason about planning problems. CILL combines both constraint solvers and intuitionistic linear logic. For that reason, it makes possible to define constraints using logical expressions. In other words, constraints has become a part of the language. In all kinds of intuitionistic logic, proofs are directly correspond to programs because a given formula is true only when it is verified. Furthermore, linear logic solves the frame problem without any additions to the language because it acts hypotheses as resources where they are consumed when used.

In this thesis, we have introduced a new robotics application domain named Blocks World 3D which is based on classical Blocks World. This domain has some differences from the original one: one should not only place a block onto another block but also keep track of balancing of blocks tower. Therefore, planning in this domain consists of placing blocks in desired order providing blocks balance in terms of set constraints where each set corresponds to base areas of each block. Two equations are used to satisfy constraints. Together with these equations and predicates defined for describing this domain, we have encoded CILL representation for Blocks World 3D. In theory, this encoding ensures that CILL searches a plan using reasoning techniques and returns a plan in terms of logical derivations if it exists.

Before implementing CILL within any logical framework or Prolog, we have decided that it is a good practice to implement first-order intuitionistic logic with constraint extension using logical programming tools and languages. For that purpose, Twelf Logical Framework is a good candidate where it supports diverse tasks which are specification of object languages and their semantics, implementation of algorithms exploiting object-language expressions and deductions, and formal development of the meta-theory of an object language. Especially, defining specification of object language, which is our first-order intuitionistic logic with constraint extension (CIL), is vital to achieve our goal. However, lack of constraint properties of Twelf Logical Framework prevents from doing further implementations for CILL. Hence, we have implemented CIL using Prolog Language in SWI-Prolog environment. Together with backtracking and declarative property of Prolog Language, and constraint handling rules and CLP(R) support of SWI-Prolog makes us possible to automate proof searching for given application domain. Moreover, some examples of both implementations of CIL has been given.

Not directly employing constraints in Twelf Logical Framework which is mentioned in previous paragraph is the main drawback of implementation of CILL within Twelf. Conversely, Prolog is convenient to encode CILL with its strong properties of handling constraints and backtracking. However, this time linearity extremely decreases the performance of automating proof search. This is because resource splitting of some connectives in linear logic leads to problem of non-determinism and linear logic introduces new context where hypotheses are behaved as resources. For that reason, the way of using hypotheses in linear and unrestricted context makes impractical to encode CILL. However, in the literature there are some techniques for resource splitting and context handling. Therefore, for future work, it is possible to encode CILL within Prolog Language using these techniques. Furthermore, another future work will be automatically extracting linear proof terms of a given formula while searching proof by using Prolog language.

Appendix A

Intuitionistic Linear Logic

A.1 Sequent Calculus

Hypothesis.

$$\frac{}{\Gamma; A \Rightarrow A}\ \mathit{init} \qquad \frac{(\Gamma, A); (\Delta, A) \Rightarrow A}{(\Gamma, A); \Delta \Rightarrow A}\ \mathit{copy}$$

Multiplicative Connectives.

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

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

$$\frac{}{\Gamma; . \Rightarrow \mathbf{1}}\ \mathbf{1}R \qquad \frac{\Gamma; \Delta \Rightarrow C}{\Gamma; \Delta, \mathbf{1} \Rightarrow C}\ \mathbf{1}L$$

Additive Connectives.

$$\frac{\Gamma; \Delta \Rightarrow A \qquad \Gamma; \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \,\&\, B}\ \& R \qquad \frac{\Gamma; \Delta, A \Rightarrow C}{\Gamma; \Delta, A \,\&\, B \Rightarrow C}\ \& L_1$$

$$\frac{\Gamma; \Delta, B \Rightarrow C}{\Gamma; \Delta, A \,\&\, B \Rightarrow C}\ \& L_2$$

$$\frac{}{\Gamma; \Delta \Rightarrow \top}\ \top R \qquad \textit{No } \top \textit{ left rule}$$

$$\frac{\Gamma; \Delta \Rightarrow A}{\Gamma; \Delta \Rightarrow A \oplus B}\ \oplus R_1$$

$$\frac{\Gamma; \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \oplus B}\ \oplus R_2 \qquad \frac{\Gamma; \Delta, A \Rightarrow C \qquad \Gamma; \Delta, B \Rightarrow C}{\Gamma; \Delta, A \oplus B \Rightarrow C}\ \oplus L$$

$$\textit{No } \mathbf{0} \textit{ right rule} \qquad \frac{}{\Gamma; \Delta, \mathbf{0} \Rightarrow C}\ \mathbf{0}L$$

Quantifiers.

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

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

Exponentials.

$$\frac{(\Gamma, A); \Delta \Rightarrow B}{\Gamma; \Delta \Rightarrow A \supset B}\ \supset R \qquad \frac{\Gamma; . \Rightarrow A \qquad \Gamma; \Delta, B \Rightarrow C}{\Gamma; \Delta, A \supset B \Rightarrow C}\ \supset L$$

$$\frac{\Gamma; . \Rightarrow A}{\Gamma; . \Rightarrow !A}\ !R \qquad \frac{(\Gamma, A); \Delta \Rightarrow C}{\Gamma; (\Delta, !A) \Rightarrow C}\ !L$$

Bibliography

[1] J. Avigad. Classical and constructive logic, 2000.

[2] F. Bacchus, F. Kabanza, and U. D. Sherbrooke. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:2000, 2000.

[3] J. Blythe. Decision-theoretic planning. AI Magazine, Summer, 1999.

[4] B. Bonet. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[5] J. Bos. Predicate logic unplugged. In *In Proceedings of the 10th Amsterdam Colloquium*, pages 133–143, 1996.

[6] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[8] R. R. Burridge. *Sequential composition of dynamically dexterous robot behaviors*. PhD thesis, University of Michigan, Ann Arbor, MI, USA, 1996. Chair-Daniel Koditschek.

[9] G. J. Chaitin. Computational complexity and gödel's incompleteness theorem. *SIGACT News*, pages 11–12, 1971.

[10] K. L. Clark. Negation as failure. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 311–325. Kaufmann, Los Altos, CA, 1987.

[11] M. B. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132:151–182, 2001.

[12] R. Epstein. *The Semantic Foundations of Logic: Propositional Logics.* Oxford University Press, 1995.

[13] G. E. Fainekos, H. Kress-Gazit, and G. J. Pappas. Temporal logic motion planning for mobile robots. In *ICRA*, pages 2020–2025. IEEE, 2005.

[14] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 88–97. Kaufmann, San Mateo, CA, 1990.

[15] Y. Gil. Plan representation and reasoning with description logics.

[16] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[17] J.-Y. Girard. Linear logic: Its syntax and semantics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic (Proc. of the Workshop on Linear Logic, Cornell University, June 1993).* Cambridge University Press, 1995.

[18] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types.* Cambridge University Press, New York, NY, USA, 1989.

[19] N. Gupta and D. S. Nau. On the complexity of blocks-world planning. *Artif. Intell.*, 56(2-3):223–254, 1992.

[20] T. L. H. K. Buning. *Propositional Logic: Deduction and Algorithms.* Cambridge University Press, 1999.

[21] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[22] W. A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article.

[23] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[24] J. Jaffar and S. Michaylov. Methodology and implementation of a clp system. In J.-L. Lassez, editor, *Logic Programming: Proc. of the Fourth International Conference (Volume 1)*, pages 196–218. MIT Press, Cambridge, MA, 1987.

[25] M. D. Johnston. SPIKE: AI scheduling for nasa's hubble space telescope. In *Proceedings of the sixth conference on Artificial intelligence applications*, pages 184–190, Piscataway, NJ, USA, 1990. IEEE Press.

[26] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201. AAAI Press, 1996.

[27] S. C. Kleene. *Introduction to Metamathematics*. Wolters-Noordhoff, Groningen, 1971.

[28] S. Michaylov. *Design and implementation of practical constraint logic programming systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[29] R. C. Moore. *Logic and representation*. Center for the Study of Language and Information, Stanford, CA, USA, 1995.

[30] D. Nardi and R. J. Brachman. *An Introduction to Description Logics*. Cambridge University Press, 2002.

[31] A. Peruzzi. The theory of descriptions revisited. *Notre Dame Journal of Formal Logic*, 30(1):91–104, 1989.

[32] F. Pfenning. Logic programming in the LF logical framework. *Logical frameworks*, pages 149–181, 1991.

[33] F. Pfenning. Elf: A meta-language for deductive systems (system description). In *CADE-12: Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, London, UK, 1994. Springer-Verlag.

[34] F. Pfenning. *Constructive Logic Notes.* Carnegie Mellon University, 2000.

[35] F. Pfenning. *Computation and Deduction Notes.* Carnegie Mellon University, 2001.

[36] F. Pfenning. *Linear Logic Notes.* Carnegie Mellon University, 2002.

[37] M. Pistore and P. Traverso. Planning as model checking for extended goals in non-deterministic domains. In *In Proc. IJCAI'01*, pages 479–484. AAAI Press, 2001.

[38] D. Prawitz. *Natural Deduction.* Almqvist & Wiksell, 1965.

[39] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *Journal of Artificial Intelligence Research*, 4:287–339, 1996.

[40] R. Reiter. The frame problem in situation the 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*, pages 359–380, 1991.

[41] E. Rimon. *Exact robot navigation using artificial potential functions.* PhD thesis, Yale University, New Haven, CT, USA, 1990. Advisor-Daniel E. Koditschek.

[42] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[43] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, second edition, 2003.

[44] E. Sandewall. An approach to the frame problem and its implementation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 195–204. Edinburgh University Press, 1972.

[45] U. Saranli and F. Pfenning. Using constrained intuitionistic linear logic for hybrid robotic planning problems. In *Proceedings of the International Conference on Robotics and Automation (ICRA'07)*, Rome Italy, Apr. 2007. IEEE Computer Society Press.

[46] C. Schurmann and F. Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Conference on Automated Deduction*, pages 286–300, 1998.

[47] N. Shankar. Proof search in the intuitionistic sequent calculus. In D. Kapur, editor, *Proceedings 11th Intl. Conf. On Automated Deduction, CADE'92, Saratoga Springs, CA, USA, 15–18 June 1992*, volume 607, pages 522–536. Springer-Verlag, Berlin, 1992.

[48] P. Smith. The phoenix mission to mars. In *Proceedings of IEEE Aerospace Conference*, Tuscon Arizona USA, Mar. 2004. IEEE Computer Society Press.

[49] R. M. Smullyan. *First-Order Logic*. Dover Publications, Inc., 1995.

[50] J. van Heijenoort, editor. *From Frege to Gödel: a Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, 1967. Reprinted 1971, 1976.

[51] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The claraty architecture for robotic autonomy. *In Proceedings of the 2001 IEEE Aerospace Conference*, March 2001.