Abstract

Fundamental issues in representing NP-complete problems

Jatin Shah

2006

In this dissertation, we study the issues involved in representing NP-complete problems in a formal logical framework. We focus on two main issues: representing NP-complete problem instances and their reductions succinctly so that the corresponding correctness proofs can also be represented, and determining statically if a reduction between two NP-complete problems is a polynomial-time reduction. We use logical framework LF and its advanced variants linear LF and concurrent LF to store NP-complete problems and their reductions. We identify advantages and disadvantages of these systems for our purpose.

Finally, we develop fairly comprehensive static criteria for distinguishing polynomial time algorithms from non-polynomial time algorithms. These criteria can be represented within the system as a proof of the fact that a reduction is polynomial-time algorithm. The criteria are general enough to be applicable for a large class of functional and logic programming languages.

Fundamental issues in representing NP-complete problems

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Jatin Shah

Dissertation Director: Carsten Schürmann

June 16, 2006

Copyright © 2006 by Jatin Shah

All rights reserved.

Contents

List of Figures

Acknowledgements

I would like to thank my advisor Carsten Schürmann for his invaluable help and advice at every stage of the research that led to this dissertation. I am especially indebted to Dana Angluin for her mentorship and guidance during my initial years as a graduate student. I am also thankful to her for proofreading my dissertation and giving me detailed comments.

I'd also like to express my sincerest gratitude to Helmut Schwichtenberg, Charles Stewart and Drew McDermott for their comments and assistance during the various stages of research and writing of this dissertation.

Chapter 1

Introduction

The primary motivation that led to this dissertation was the desire to build a library to store NP-complete problems, reductions between those problems and the proofs of correctness. On one hand, this work falls naturally within the research area of representation of mathematics that started with de Bruijn's AUTOMATH project [20]. Yet, it is unique in that it requires formalization of concepts from theoretical computer science such as algorithms, properties about algorithms and their run-time complexity. Thus, the initial goal of the research was to identify systems that were developed for representation of mathematics and adapt them for the purpose of representing NP-complete problems and their reductions.

In this thesis, we shall identify fundamental issues involved in developing a formal system for this purpose. We will present most of our results in the logical framework LF [31] or its advanced variants linear LF (LLF) and concurrent LF (CLF). While this choice was partly influenced by our prior familiarity with the system, LF provides a rich environment for representing a variety of logics. Moreover, it also provides a logic programming language for representing algorithms. Thus, we believe logical framework LF with its *proofs-as-programs* paradigm to be a good starting point for the research.

We shall begin by attempting to represent the most common NP-complete problems like 3-SAT in logical framework LF. It will become quite obvious that plain LF is not adequate to express all but the simplest NP-complete problems. In particular, we shall focus on representing NP-complete problems and their associated reductions formally and develop a decidable criteria for analyzing their run-time complexity.

We would like to point out that many of our ideas are general enough to be exported to variety of theorem proving environments that use concepts from functional and logic programming and support features such as *higher-order* pattern matching. In fact, many of our ideas have potential applications in areas beyond representation of mathematics. We will discuss these ideas more fully in the conclusion.

In this chapter, we shall begin by giving a brief history and description of the main developments in the representation of mathematics. We shall also provide our rationale for choosing logical framework LF for presenting our main observations and results. Section 1.2 provides a flavor of our approach and summarizes our main results.

1.1 Brief history of representation of mathematics

The AUTOMATH project [20] initiated by de Bruijn was a pioneering attempt at formalizing mathematical arguments in a language suitable for machine checking. During the late sixties and seventies, the AUTOMATH project led to the creation of an entire class of languages whose main goal was to provide a formal framework for representing various logics and logical theories.

Many ideas from the AUTOMATH language family have found their way into modern systems. The NuPRL system [16] is one such system that is based on AU-

TOMATH, as is Milner's LCF system [29] for interactive proof development. NuPRL is a full-scale interactive proof development environment that provides support not only for interactive proof construction, but also notational extension, abbreviations, library management, and automated proof search. The AUTOMATH language family has strongly influenced the design of the logical framework LF [31]. The logical framework LF captures uniformities of a wide class of logics allowing the user to choose a suitable logic for proof development. Logics are represented in LF via the *judgments-as-types* principle [31] whereby each judgment is identified with the type of its proofs. This principle can be regarded as a meta-theoretic analogue of the well-known proposition-as-types principle [18, 19, 37]. Later, logic programming based proof search was incorporated within LF [59] to improve its proof search capabilities.

These formal reasoning systems and their related variants have been successfully applied in several domains such as compiler verification [3, 14, 30, 43, 60], proof carrying code [2, 56] and verification of cryptographic protocols [23, 46, 47, 48, 49, 50]. However, we are not aware of any substantial attempt to formalize problems and algorithms that are encountered in theoretical computer science within these systems.

1.2 Formal representations of NP-complete problems – A sample problem

NP problems are a class of decision problems for which all known solutions require exponential time in the worst case. However, for these problems, it can be checked in polynomial time whether a given claimed solution, called a *witness*, is indeed an actual solution to the problem. Thus, these problems can be solved efficiently by a non-deterministic guess-and-verify algorithm. A problem is said to be NP-complete if it is in NP and every other problem in NP *reduces* to the problem. Intuitively, a problem A is said to be *reducible* to a problem B if there is a way to encode instances x of a problem A as instances $\sigma(x)$ of problem B. The encoding function σ is called a reduction. If σ is a polynomial time function, then any efficient algorithm for B will yield an efficient algorithm for A by composing it with σ.

Thousands of interesting problems in engineering and sciences are known to be NP-complete. In fact, when confronted with a new computational problem, the first step is often to determine if it is indeed an NP-complete problem by *reducing* it to a known NP-complete problem. We would like to note here that most NP-complete problems usually have a very obvious and a fairly simple polynomial time reduction.

The theory of NP-completeness and polynomial time reductions was initiated in early 1970s. The two principal papers that first demonstrated the importance of these concepts were by Cook [17], who showed by Boolean satisfiability was NP-complete, and Karp [39, 40] who showed that many interesting combinatorial problems were interreducible and hence NP-complete. Garey and Johnson's text [26] provides a good introduction to the theory of NP-completeness and contains an extensive list of NP-complete problems. In this dissertation, we shall primarily focus on the 21 NP-completeness problems identified by Karp [39] in his seminal paper. We have reproduced this list in the Appendix A.

A large number of NP-complete problems and their instances are usually described in terms of graphs, formulas, partitions, matchings, and colorings. Since we are focusing on decision problems, i.e. a problem whose solution is either *yes* or *no*. Thus, these instances are further classified as *Yes* instances or *No* instances.

For example, Definitions 1.2.1 and 1.2.2 describe two problems concerning the satisfiability of boolean formulas in conjunctive normal form. In this case, $(u_1 \vee u_2) \wedge (\bar{u}_2 \vee u_3 \vee u_4) \wedge (\bar{u}_1 \wedge u_3) \wedge (\bar{u}_4)\}$, an instance of SAT, has a satisfying truth assignment $\{u_1 \rightarrow \mathsf{true}, u_2 \rightarrow \mathsf{false}, u_3 \rightarrow \mathsf{true}, u_4 \rightarrow \mathsf{false}\}$ and hence is a *Yes* instance. On the other hand, the boolean formula $(u_1 \vee u_2) \wedge (\bar{u}_1 \vee u_3) \wedge (\bar{u}_2 \vee \bar{u}_3) \wedge (\bar{u}_1)$ has no satisfying truth assignment and is a *No* instance.

Definition 1.2.1 (SAT)**.** Given a set $U = \{u_1, u_2, \ldots, u_n\}$ of Boolean variables and a conjunctive normal form formula $f = c_1 \wedge c_2 \wedge \ldots c_m$ on Boolean variables such that $c_i = l_{i1} \vee l_{i2} \vee \ldots \vee l_{ik_i}$, $\forall i = 1, \ldots, m$ and $k_i \in \mathbb{Z}$, and $l_{i1}, l_{i2}, \ldots, l_{ik_i} \in U \cup \overline{U}$ where $\overline{U} = \{\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n\}$.

QUESTION: Is there a truth assignment to the Boolean variables such that every clause in f is satisfied?

Definition 1.2.2 (3-SAT)**.** Given a set $U = \{u_1, u_2, \ldots, u_n\}$ of Boolean variables and a conjunctive normal form formula $f = c_1 \wedge c_2 \wedge \ldots c_m$ on the Boolean variables in U such that $c_i = l_{i1} \vee l_{i2} \vee l_{i3}, \forall i = 1, \ldots, m$ and $l_{i1}, l_{i2}, l_{i3} \in U \cup \overline{U}$ where $\overline{U} = \{\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n\}$.

QUESTION: Is there a truth assignment to the Boolean variables such that every clause in f is satisfied?

We would like to represent reductions between NP-complete problems, their associated correctness proofs, i.e. a proof that the reduction maps every *Yes* instance of the first NP-complete problem to a *Yes* instance of the second NP-complete problem and vice-versa, and finally be able to prove that the reduction represents a polynomial time computable function.

In this dissertation, we shall begin by attempting to implement these ideas in the logical framework LF. LF, with its dependently-typed term algebra, gives us ample freedom to choose an appropriate representation for the problems and their corresponding instances. After identifying advantages and difficulties in using LF, we will show how advanced logical frameworks like linear logical framework LLF [11, 12] and concurrent logical framework CLF [13, 71] resolve many of these difficulties.

Finally, we will develop a syntactic analyzer to determine if a reduction described in these logical frameworks is a polynomial time computable function.

1.2.1 Representing problem instances

Instances of an NP-complete problem are represented as terms in the logical framework LF with a common type. We require that these representations are adequate in the sense that every problem instance corresponds to an object in LF with the given type and vice-versa.

LF with its dependently typed λ-calculus, $\lambda^{\to\Pi}$, is expressive enough to be able to express a wide class of NP-complete problems and their instances. The canonical forms are β-normal and η-long.

For example, the instances of SAT and 3-SAT are just propositional formulas with connectives $\wedge$, $\vee$ and $\neg$ which can be represented in $\lambda^{\to\Pi}$ quite easily. Boolean variables are schematic and denoted by $u_1, \ldots, u_n$. New variables are introduced using the binder new as shown below. In the formula $\mathsf{new}\ u.F$, the variable u bound by new is free in the formula F.

$$\begin{array}{llll} \textit{Boolean variables} & & & u, u_n \\ \textit{Boolean formulas} & F, F_n & ::= & \mathsf{pos}\ u \mid \mathsf{neg}\ u \mid \mathsf{new}\ u.F \mid F_m \wedge F_n \mid F_m \vee F_n \end{array}$$

The corresponding LF representation is given in Figure 1.1. In this case, we have chosen two base types, v and o corresponding to variables and formulas respectively. For binary operators $\wedge$ and $\vee$, we define functions $\mathsf{and} : \mathsf{o} \to \mathsf{o} \to \mathsf{o}$ and $\mathsf{or} : \mathsf{o} \to \mathsf{o} \to \mathsf{o}$ respectively. The variable binding of the binder new is modeled using the binder λ of λ-calculus.

The fact that an instance of 3-SAT is a *Yes* instance is written as $\eta \vdash F\ \mathsf{SAT}$

$$
\begin{array}{rcll}
\ulcorner x \urcorner & = & x & \\
\\
\ulcorner \mathsf{pos}\ x \urcorner & = & \mathsf{pos}\ x & \mathsf{pos} : \mathsf{o} \to \mathsf{o} \\
\ulcorner \mathsf{neg}\ x \urcorner & = & \mathsf{neg}\ x & \mathsf{neg} : \mathsf{o} \to \mathsf{o} \\
\ulcorner \mathsf{new}\ u.F \urcorner & = & \mathsf{new}\ (\lambda u. \ulcorner F \urcorner) & \mathsf{new} : (\mathsf{v} \to \mathsf{o}) \to \mathsf{o} \\
\ulcorner F_m \wedge F_n \urcorner & = & \ulcorner F_m \urcorner\ \mathsf{and}\ \ulcorner F_n \urcorner & \mathsf{and} : \mathsf{o} \to \mathsf{o} \to \mathsf{o} \\
\ulcorner F_m \vee F_n \urcorner & = & \ulcorner F_m \urcorner\ \mathsf{or}\ \ulcorner F_n \urcorner & \mathsf{or} : \mathsf{o} \to \mathsf{o} \to \mathsf{o}
\end{array}
$$

Figure 1.1: Representation of Boolean formulas in LF

$$
\frac{}{\eta, u \to \mathsf{true} \vdash (\mathsf{pos}\ u)\ \mathsf{SAT}}\ satp \qquad \frac{}{\eta, u \to \mathsf{false} \vdash (\mathsf{neg}\ u)\ \mathsf{SAT}}\ satn
$$

$$
\frac{\eta \vdash F_1\ \mathsf{SAT} \quad \eta \vdash F_2\ \mathsf{SAT}}{\eta \vdash (F_1 \wedge F_2)\ \mathsf{SAT}}\ sat\wedge
$$

$$
\frac{\eta \vdash F_1\ \mathsf{SAT}}{\eta \vdash (F_1 \vee F_2)\ \mathsf{SAT}}\ sat\vee 1 \qquad \frac{\eta \vdash F_2\ \mathsf{SAT}}{\eta \vdash (F_1 \vee F_2)\ \mathsf{SAT}}\ sat\vee 2
$$

$$
\frac{\eta, u \to \mathsf{true} \vdash F\ \mathsf{SAT}}{\eta \vdash \mathsf{new}\ u.F\ \mathsf{SAT}}\ satt \qquad \frac{\eta, u \to \mathsf{false} \vdash F\ \mathsf{SAT}}{\eta \vdash \mathsf{new}\ u.F\ \mathsf{SAT}}\ satf
$$

Figure 1.2: Inference rules for “Yes” instances of SAT and 3-SAT.

where environments η contain assignments for free variables in F. Environments satisfy standard properties of intuitionistic contexts such as *weakening*, *strengthening* and *permutation* and behave as a *witness* for the NP-complete problem instance being considered.

$$
\text{Environments:} \quad \eta \quad ::= \quad \cdot \mid \eta, u \to \mathsf{true} \mid \eta, u \to \mathsf{false}
$$

The set of inference rules corresponding to SAT and 3-SAT are given in Figure 1.2. It is also worth noting that the intractability of finding *witnesses* for satisfiability of SAT and 3-SAT have mirror images in the world of inference systems as well – finding a proof of the judgment $\cdot \vdash F$ SAT may involve checking the exponentially many assignments to boolean variables.

$$
\begin{array}{rcl}
(l_1) & \Rightarrow & (l_1 \vee v_{i1} \vee v_{i2}) \wedge (l_1 \vee \bar{v}_{i1} \vee v_{i2}) \wedge \\
 & & (l_1 \vee v_{i1} \vee \bar{v}_{i2}) \wedge (l_1 \vee \bar{v}_{i1} \vee \bar{v}_{i2}) \\
(l_1 \vee l_2) & \Rightarrow & (l_1 \vee l_2 \vee v_{i1}) \wedge (l_1 \vee l_2 \vee \bar{v}_{i1}) \\
(l_1 \vee l_2 \vee l_3) & \Rightarrow & (l_1 \vee l_2 \vee l_3) \\
(l_1 \vee l_2 \ldots \vee l_{ik_i}) & \Rightarrow & (l_1 \vee l_2 \vee v_{i1}) \wedge \\
 & & (\bar{v}_{i1} \vee l_3 \vee v_{i2}) \wedge \\
 & & (\bar{v}_{i2} \vee l_4 \vee v_{i3}) \wedge \\
 & & \vdots \\
 & & (\bar{v}_{ik_i-4} \vee l_{k_i-2} \vee v_{ik_i-3}) \wedge \\
 & & (\bar{v}_{ik_i-3} \vee l_{k_i-1} \vee l_{k_i}) \wedge
\end{array}
$$

Figure 1.3: Transformation of an instance of SAT to 3-SAT

We shall present LF representation of these inference rules in Chapter 2 and identify some particular difficulties that are encountered in this representation. In Chapter 3, we shall present another approach based on using concurrent logical framework (CLF) [13, 71] that resolves some of these difficulties. The following section gives a brief overview of this approach for representing NP-complete problem instances and the corresponding reductions.

1.2.2 Representing reduction algorithms

The Elf [58, 59] programming language combines the representation style of LF with logic programming model of λProlog [53, 54]. This language serves as a good choice for representing reductions of NP-complete problems, because it provides us with a good framework for both representing proofs of correctness and analyzing the complexity of the reduction algorithm.

We shall illustrate this approach here briefly by presenting a reduction of SAT to 3SAT in Elf. A more detailed presentation can be found in Chapters 2 and 3.

As given in Definition 1.2.1, $U = \{u_1, u_2 \ldots, u_n\}$ is the set of variables and $C = \{c_1, c_2, \ldots, c_m\}$ is the set of clauses. We shall construct a set C' of three literal clauses on the set $U' \supseteq U$ of variables such that C' is satisfiable if and only if C

is satisfiable. Each clause $c_i \in C$ is replaced with a set of three literal clauses with some new variables as shown in Figure 1.3. Let $c_i = (l_1 \vee l_2 \ldots \vee l_{ik_i})$ where l_i are literals and $v_{i1}, \ldots, v_{ik_i-3}$ are new variables.

We shall present an Elf program corresponding to this reduction in Chapter 2. We shall see that the reduction algorithm is quite intuitive with a program statement corresponding to every case given in Figure 1.3.

At this point, we would like to note an important discrepancy between our conceptual representation of literals within clauses and the corresponding representation in LF. In the algorithm given in Figure 1.3, the order in which the literals are processed does not matter. On the other hand, our representation of boolean formulas forces us to choose an *artificial* ordering on the literals within each clause. As we shall see, this limitation does not affect the LF encoding of this reduction. However, when we need to encode a relation on two or more objects this drawback unnecessarily increases the complexity of the representation. We shall also encounter this problem when we begin encoding reduction algorithms for more complicated NP-complete problems involving mathematical objects like graphs and also when we represent correctness proofs of NP-complete reductions.

We resolve this difficulty by allowing *active pattern matching* as described by Erwig [22]. Briefly, active pattern matching rearranges the data elements in the data-type until pattern match succeeds. Thus, an *active pattern match* will pick the right element to be head depending on the function. Essentially, this allows random access to elements of a list. In some cases, we shall use intuitionistic and linear contexts to store data instead of explicitly using terms with active pattern matching. An extension of the logical framework LF known as the concurrent logical framework CLF [13, 71] incorporates a form of active pattern matching. In Chapter 3, we shall describe a representation approach using active patterns in CLF that resolves much

$$\begin{aligned} T_1(1) &= 1 \\ T_1(2) &= 1 \\ T_1(3) &= 1 \\ T_1(x) &= T_1(\lfloor x/3 \rfloor) + T_1(\lfloor x/4 \rfloor) + x \end{aligned} \qquad \begin{aligned} T_2(1) &= 1 \\ T_2(2) &= 1 \\ T_2(x) &= T_2(x-1) + T_2(x-2) \end{aligned}$$

Figure 1.4: Two recursive functions

of this difficulty for NP-complete problems on complex mathematical objects.

1.2.3 Proving complexity theoretic properties

A significant contribution of this thesis is the development of a sound and sufficient criterion for identifying polynomial-time recursive functions. The criterion is not only expressive enough to identify functions over higher-order data-types but also complete in the sense that every polynomial-time recursive function has at least one implementation (usually over binary strings) that is identified by the criterion. We shall illustrate the applicability of this criterion for checking that reduction between NP-complete problems are polynomial time functions through several examples in Chapters 4 and 5.

The main idea is based on the observation that when sum of the sizes of the arguments passed to the recursive calls does not exceed the size of the input arguments and the additional non-recursive computation is requires polynomial-time, the function is polynomial-time computable. For example, consider the functions given in Figure 1.4. It is easy to show that $T_1(x) = O(x^3)$ as $x \geq \lfloor x/3 \rfloor + \lfloor x/4 \rfloor$. However, $T_2(x) = \Theta(\phi^x)$ where $\phi = \frac{1+\sqrt{5}}{2}$. In this case, $x \not\geq (x-1) + (x-2)$.

We shall develop these ideas fully in Chapter 4. Since we wish to incorporate this analyzer within logical frameworks, we shall use logic programming as the primary model of computation. An important feature of our approach is that it transforms the complexity analysis problem to a simpler problem of solving multi-variable linear and polynomial inequalities with integer coefficients.

In Chapter 5, we present a criterion for identifying polynomial time forward-chaining logic programs. This criterion is similar to our approach for recursive functions in that it is also based on identifying how the sizes of the inputs to the logic program change during the program execution.

The results in these chapters are presented in logic programming languages based on the Horn fragment. Hence, they can be read independently of the earlier chapters.

1.3 Statement of achievement

We began this research project with the goal of building a system for storing NP-complete problems and their reductions. Admittedly, we are still very far away from that goal. However, we have made certain fundamental observations and provided important contributions that would be useful for any future research on this topic.

We have given several detailed examples of representation of NP-complete problems and their reductions in the logical framework LF. We have also identified limitations of using a system like LF for this purpose and identified several features that would be needed for effective representations of these problems. We have also identified the difficulty of representing correctness proofs of these reductions. While LF allows us to represent these proofs, its more recent extensions like LLF and CLF lack proper support for this purpose (Chapters 2 and 3).

The most important contribution of this dissertation is the development of *sufficient* criteria for statically identifying polynomial time algorithms. The criteria assume that the algorithms are represented as a logic programs. We have given separate criteria for both backward-chaining logic programs without backtracking and forward-chaining logic programs (Chapters 4 and 5). We have also given several examples to illustrate that these criteria are practical and can recognize a large class of reductions between NP-complete problems. These results have potential applications in many areas beyond identifying reductions between NP-complete problems. We shall discuss these potential applications in conclusion and future work (Chapter 6).

Thus, we believe that this dissertation is a positive step towards the goal of representing NP-complete problems formally.

1.4 Organization of this thesis

This thesis is organized as follows. In Chapter 2, we describe a preliminary attempt at representing some basic NP-complete problems and their associated reductions within the logical framework LF and linear logical framework LLF. Here we also identify limitations of this approach some of which have been discussed in this chapter. In the following chapter, we use the concurrent logical framework CLF to represent the same problems and the reductions.

Later, in Chapters 4 and 5, we give detailed polynomial-time checkers for backward-chaining and forward-chaining models of computation. The interested reader can directly skip to these chapters without loss of continuity. Finally, we close with conclusions and future work.

Chapter 2

Representing NP-complete problems

A framework that allows representation of NP-complete problems should have three main features:

1. It should allow representation of NP-complete problem instances and proofs that those instances are *Yes* or *No* instances

2. It should also be possible to represent the reduction algorithms between NP-complete problems with their associated correctness proofs, i.e. the algorithm converts *Yes* instance of the first problem to *Yes* instance of the second problem and vice-versa.

3. Finally, it should be possible to represent a proof of the fact that the reduction algorithm is a polynomial-time computable function.

In this chapter, we shall focus on the first two features and leave the third feature for detailed study in Chapters 4 and 5. In particular, we shall illustrate through a few examples how the logical framework LF can be used for this purpose. Later we shall also discuss advantages in using the linear logical framework LLF over plain LF.

2.1 Logical framework: LF

Logical framework LF is structured into three kinds of terms: objects, types and kinds. Usually objects (denoted by M, N or P) are used to represent entities, proofs or inference rules and types (denoted by A, B or C) are used to represent judgments and assertions. Kinds (denoted by K or L) are introduced for technical reasons to classify types.

The abstract syntax of LF is given below. The variables x, y and z range over the variables, c and d range over the object constants, and a and b over the type family level constants.

$$
\begin{array}{llcl}
\textit{Kinds} & K & ::= & \mathsf{type} \mid \Pi x : A.K \\
\textit{Types} & A, B & ::= & \mathsf{a} \mid \Pi x : A.B \mid A \rightarrow B \\
\textit{Objects} & M, N & ::= & \mathsf{c} \mid x \mid \lambda x : A.M \mid MN
\end{array}
$$

λ and Π are binding operators binding the variable x that is free in the object M and type B respectively. We write $A \rightarrow B$ for $\Pi x : A.B$ when x does not occur free in B and assume that $\rightarrow$ is right associative. In LF, *signatures* are used to keep track of types and kinds assigned to constants, and *contexts* are used to keep track of types assigned to variables. Thus, they have the following syntax:

$$
\begin{array}{llcl}
\textit{Signatures} & \Sigma & ::= & \cdot \mid \Sigma, \mathsf{a} : K \mid \Sigma, \mathsf{c} : A \\
\textit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

Contexts are sets, which are syntactically represented as lists. In the next chapter, we shall use this idea to use contexts to store complex mathematical objects. The notion of definitional equality that is used in LF is based on β-conversion and the fact that kinds, types and objects are all strongly normalizing. The canonical forms in LF are β-normal and η-long. Harper, *et al.* [31] describe these concepts in detail.

2.1.1 Logic programming in Elf

The Elf programming language [58] builds upon the syntax of the logical framework LF to provide a logic programming based operational semantics. This is achieved by distinguishing types into *predicates* or atomic formulas, *goals* and *clauses*, thus giving them an operational interpretation.

The type system of LF can be rewritten as shown below where D, G and P correspond to clauses, goals and predicates respectively. LF level objects are denoted by M and are extended with logic variables. X_Γ^A is a logic variable valid in $\Gamma \vdash X : A$. This extension is well-understood [53].

$$
\begin{array}{lrcl}
\textit{Clauses} & D & ::= & P \mid G \rightarrow D \mid \Pi x : A.D \\
\textit{Goals} & G & ::= & P \mid D \rightarrow G \mid \Pi x : A.G \\
\textit{Predicates} & P & ::= & \mathsf{a} \mid P M \\
\textit{Objects} & M, N & ::= & \mathsf{c} \mid x \mid \lambda x : A.M \mid M\ N \mid X_\Gamma^A
\end{array}
$$

The logic-programming-based operational semantics are based on operational semantics for abstract logic programming languages [53]. The proof search rules are given in Figure 2.1. It consists of two judgments, viz. $\Sigma \vDash G, \theta$ and $\Sigma \vdash D \gg P, \theta$. The first judgment corresponds to the task of proving goal G under signature Σ. And the second judgment is essentially the *back-chaining* judgment which identifies the right program clause D for the atomic goal P. In both cases, θ is a list of unification constraints that are produced during the search. The unification algorithm is described in detail in Pfenning [59].

Thus, θ is an output produced after the search is complete. We would like to note here that in the most general case, unification in LF with logic variables is not decidable and θ may contain unsolved constraints. The language of θ is given below:

$$\theta ::= \cdot \mid X \doteq Y, \theta \mid X \mapsto M, \theta$$

In the operational semantics given in Figure 2.1, $\theta(G)$ corresponds to the goal produced after substitutions from θ have been performed on G. Similarly, $\theta'(\theta)$ is the new substitution list produced after applying substitutions from θ' on the substitution list θ. It is defined as given below:

$$\begin{aligned} \cdot(\theta) &= \theta \\ (X \mapsto M, \theta')(\theta) &= \theta'([M/X]\theta, X \mapsto M) \\ (X \doteq Y, \theta')(\theta) &= \theta'(\theta, X \doteq Y) \end{aligned}$$

In the rules SOME and ALL, $[X/x]D$ and $[\mathsf{c}/x]D$ are the types after free occurrences of x in D have been replaced by the logic variable X and c respectively.

The Π quantifier in rules ALL and SOME perform quite different roles. In the first case, the Π quantifier binds any new variables used in embedded implications of the form $D \rightarrow G$ which in effect are dynamic run-time extensions to the main signature Σ. In contrast, the Π quantifier in SOME binds variables of the clause D selected during backward chaining and represent input and output variables of that clause. We shall return to this topic in more detail when we present the criterion for identifying polynomial-time recursive functions in Chapter 4, but the reader can refer to Pfenning [59] for more details.

Goals:

$$\frac{\Sigma, \mathsf{c} : D \vDash D \gg P, \theta}{\Sigma, \mathsf{c} : D \vDash P, \theta} \; \mathsf{CLAUSE}$$

$$\frac{\mathsf{c}\ \mathsf{new} \quad \Sigma, \mathsf{c} : D \vDash G, \theta}{\Sigma \vDash D \to G, \theta} \; \mathsf{IMP} \qquad \frac{\mathsf{c}\ \mathsf{new} \quad \Sigma, \mathsf{c} : A \vDash [\mathsf{c}/x]G, \theta}{\Sigma \vDash \Pi x : A.G, \theta} \; \mathsf{ALL}$$

Clauses:

$$\frac{\theta \in \mathsf{unify}(P, Q)}{\Sigma \vDash Q \gg P, \theta} \; \mathsf{ATOM} \qquad \frac{\Sigma \vDash D \gg P, \theta \quad \Sigma \vDash \theta(G), \theta'}{\Sigma \vDash G \to D \gg P, \theta'(\theta)} \; \mathsf{SUBGOAL}$$

$$\frac{\Sigma \vDash [X/x]D \gg P, \theta}{\Sigma \vDash \Pi x : A.D \gg P, \theta} \; \mathsf{SOME}\ (\Sigma \vdash X : A)$$

Figure 2.1: Logic programming in logical framework Elf

2.1.2 Mode correct logic programs

We are interested in a particular subclass of logic programs, namely those whose arguments positions have a well-defined meaning with respect to input and output behavior of ground terms. Being *ground* means that terms cannot contain free logic variables. Modes have been proposed for expressing such aspects of the operational semantics of logic programs [38, 62, 69]. The simplest and most useful modes declare the *input* and *output* arguments of a predicate. The input arguments to a predicate should be ground when it is called. The output arguments should be free logic variables when the predicate is called and ground on successful return.

Such logic programs are called *well-moded* logic programs. Being well-moded is intuitively necessary to assign any kind of functional behavior to logic programs. For well-moded logic programs, there are no unsolved unifications constraints after the proof search, i.e. in the output θ, all logic variables have only *ground* terms assigned to them.

We assign polarities $p ::= + \mid -$ for input and output respectively, and a *mode* $m_a = \langle p_1, \ldots, p_n \rangle$ for every type family $a : \Pi x_1 : A_1.\Pi x_2 : A_2.\ldots.\Pi x_n : A_n.\mathsf{Type} \in$

Σ. We also use the following abbreviation for the input positions of the predicate $a : m_a^+ = \{i | m_a \overset{def}{=} \langle p_0, p_1, \ldots, p_n \rangle \wedge p_i = +\}$ and define m_a^- similarly. Recently, it has been shown that *modes* of type families in a LF signature can be checked effectively [64].

2.2 Representing problem instances

Instances of NP-complete problems will be represented as LF level objects. New object constants will be introduced as needed for particular problems.

Let us revisit the NP-complete problems SAT and 3-SAT that we introduced in the introduction. We introduce five new object constants, viz. $\mathsf{pos} : \mathsf{o} \rightarrow \mathsf{o}$, $\mathsf{neg} : \mathsf{o} \rightarrow \mathsf{o}$, $\mathsf{and} : \mathsf{o} \rightarrow \mathsf{o} \rightarrow \mathsf{o}$, $\mathsf{or} : \mathsf{o} \rightarrow \mathsf{o} \rightarrow \mathsf{o}$, $\mathsf{new} : (\mathsf{v} \rightarrow \mathsf{o}) \rightarrow \mathsf{o}$. These constants are sufficient to express boolean formulas. Thus, the boolean formula in conjunctive normal form, $(u_1 \vee u_2 \vee \bar{u}_3) \wedge (\bar{u}_2 \vee u_3) \wedge (u_4 \vee u_1)$ is written as

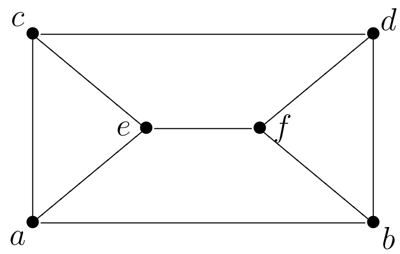
$$
\begin{aligned}
&\mathsf{new}\ u_1.\mathsf{new}\ u_2.\mathsf{new}\ u_3.\mathsf{new}\ u_4. \\
&\qquad (\mathsf{and}\ (\mathsf{or}\ u_1\ (\mathsf{or}\ u_2\ (\mathsf{neg}\ u_3)))\ (\mathsf{and}\ (\mathsf{or}\ (\mathsf{neg}\ u_2)\ u_3)\ (\mathsf{or}\ u_4\ u_1)))
\end{aligned}
$$

in LF.

On the other hand, consider the NP-complete problem, VERTEX COVER, defined below.

Definition 2.2.1 (VERTEX COVER)**.** Given a graph G and a positive integer $K \leq |V|$.

QUESTION: Is there a *vertex cover* of size K or less for G, that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and, for each edge $\{u, v\} \in E$, at least one of u and v belongs to V'?



$\mathsf{newv}\ a.\mathsf{newv}\ b.\mathsf{newv}\ c.\mathsf{newv}\ d.$
$\mathsf{newv}\ e.\mathsf{newv}\ f.$
$\mathsf{newe}\ e_1 : (a, b).\mathsf{newe}\ e_2 : (a, c).$
$\mathsf{newe}\ e_3 : (c, d).\mathsf{newe}\ e_4 : (b, d).$
$\mathsf{newe}\ e_5 : (e, f).$
$\mathsf{newe}\ e_6 : (a, e).\mathsf{newe}\ e_7 : (c, e)$
$\mathsf{newe}\ e_8 : (b, f).\mathsf{newe}\ e_9 : (d, f).\#$

Figure 2.2: A graph and its corresponding graph expression in LF

The primary mathematical object used in defining a problem instance is a graph. We shall represent graphs in a style similar to that of boolean formulas by introducing two new binders newv and newe for vertices and edges respectively. # denotes an empty graph. In effect, the expression corresponding to a graph is simply a list of all the edges in the graph. We shall also assume that all newv binders appear before any newe binders. This assumption greatly simplifies representation of many graph algorithms.

Vertex variables			v, w, x, v_n, w_n, x_n
Edges			e, e_n
Graphs	G, G_n	$::=$	$\# \mid \mathsf{newv}\ v.G \mid \mathsf{newe}\ e : (v_m, v_n).G$

For example, Figure 2.2 shows a graph and the corresponding graph expression in our syntax.

While such a representation suffices for the moment, we will very soon realize limitations that are imposed by fixed ordering of the vertices and edges.

2.2.1 Representing *Yes* instances

We have already seen in the introduction a deductive system for representing satisfiable boolean formulas (Figure 1.2). A simple proof search technique of assigning boolean variables true and false would eventually (after considering at most exponentially many possibilities) identify if a given boolean formula is a *Yes* instance. For example, the proof that the boolean formula $\bar{u}_1 \wedge u_2$ is satisfiable is given below in Figure 2.3.

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot, u_1 \to \mathsf{false}, u_2 \to \mathsf{true} \vdash (\mathsf{neg}\ u_1)\ \mathsf{SAT}}\,satn \quad \dfrac{}{\cdot, u_1 \to \mathsf{false}, u_2 \to \mathsf{true} \vdash (\mathsf{pos}\ u_2)\ \mathsf{SAT}}\,satp}{\cdot, u_1 \to \mathsf{false}, u_2 \to \mathsf{true} \vdash (\mathsf{neg}\ u_1) \wedge (\mathsf{pos}\ u_2)\ \mathsf{SAT}}\,sat\wedge}{\cdot, u_1 \to \mathsf{false} \vdash \mathsf{new}\ u_2.(\mathsf{neg}\ u_1) \wedge (\mathsf{pos}\ u_2)\ \mathsf{SAT}}\,satt}{\cdot \vdash \mathsf{new}\ u_1.\mathsf{new}\ u_2.(\mathsf{neg}\ u_1) \wedge (\mathsf{pos}\ u_2)\ \mathsf{SAT}}\,satf
$$

Figure 2.3: Proof that the Boolean formula $\bar{u}_1 \wedge u_2$ is satisfiable

The corresponding LF representation (Figure 2.4) follows quite naturally from the deductive system. It is based on the *judgments-as-types* methodology. Thus, the representation of the judgment $\eta \vdash F\ \mathsf{SAT}$ would be $\ulcorner \eta \urcorner \to \ulcorner F \urcorner$ where $\ulcorner \eta \urcorner$ and $\ulcorner F \urcorner$ are LF representations of η and F respectively. We have a new object constant corresponding to every inference rule. The type of the object constant depends on the inference rule that is being encoded. Inference rules which have antecedents are represented as arrow types. Assignment of boolean values true and false to boolean variables is modeled using hypothetical judgments. In the encoding, the term object satt has an arrow type and the type $\Pi v : \mathsf{v.hyp}\ v\ \mathsf{true} \to (F\ v)$ introduces the hypothesis that the boolean variable v is assigned the value true. This hypothesis maybe used later by term object satp during proof construction. Thus, the proof given in Figure 2.3 would be encoded as the LF object $\mathsf{satf}\ (\lambda h_1 : \mathsf{hyp}\ u_1\ \mathsf{true.satt}\ (\lambda h_2 : \mathsf{hyp}\ u_2\ \mathsf{false.sat}\wedge\ (\mathsf{satn}\ h_1)\ (\mathsf{satt}\ h_2)))$.

$$
\begin{array}{rcl}
\mathsf{hyp} & : & \mathsf{o} \rightarrow \mathsf{type}. \\
\mathsf{sat} & : & \mathsf{o} \rightarrow \mathsf{type}. \\
 & \vdots & \\
\mathsf{satp} & : & \mathsf{hyp}\ u\ \mathsf{true} \rightarrow \mathsf{sat}\ (\mathsf{pos}\ u). \\
\mathsf{satn} & : & \mathsf{hyp}\ u\ \mathsf{false} \rightarrow \mathsf{sat}\ (\mathsf{neg}\ u). \\
\mathsf{sat}\wedge & : & \mathsf{sat}\ F_1 \rightarrow \mathsf{sat}\ F_2 \rightarrow \mathsf{sat}\ (F_1 \wedge F_2). \\
\mathsf{sat}\vee_1 & : & \mathsf{sat}\ F_1 \rightarrow \mathsf{sat}\ (F_1 \vee F_2). \\
\mathsf{sat}\vee_2 & : & \mathsf{sat}\ F_2 \rightarrow \mathsf{sat}\ (F_1 \vee F_2). \\
\mathsf{satt} & : & (\Pi v : \mathsf{v}.\mathsf{hyp}\ v\ \mathsf{true} \rightarrow \mathsf{sat}\ (F\ v)) \rightarrow \mathsf{sat}\ (\mathsf{new}\ F). \\
\mathsf{satf} & : & (\Pi v : \mathsf{v}.\mathsf{hyp}\ v\ \mathsf{false} \rightarrow \mathsf{sat}\ (F\ v)) \rightarrow \mathsf{sat}\ (\mathsf{new}\ F).
\end{array}
$$

Figure 2.4: Encoding of *Yes* instances of SAT and 3-SAT in Elf

2.3 Representing reduction algorithms in LF

Reductions between NP-complete problems are formulated as inference systems and represented in Elf. Let us illustrate this approach using the reduction from SAT to 3-SAT that was introduced earlier.

A *polynomial time reduction* from SAT to 3-SAT consists in showing that there exists an algorithm which runs in time polynomial in the size of the Boolean formula that converts every instance of SAT to an instance of 3-SAT such that all *Yes* instances of SAT are mapped to *Yes* instances of 3-SAT and vice-versa. Figure 2.5 gives the reduction in Elf. In Chapter 4 (page 102), we show formally that the reduction is a polynomial time algorithm.

The program is written in a declarative programming style and is based on the logic programming based operational semantics of LF. This ensures that the program is closer to the actual mathematical representation and allows reasoning about its properties and the corresponding proofs.

We shall now give some more examples of reductions between NP-complete problems that are represented in LF. The reader can find complete definitions of NP-complete problems mentioned below in Appendix A. The reader may refer Karp [39]

$$
\begin{array}{rcl}
\mathsf{literal} & : & \mathsf{o} \rightarrow \mathsf{type}. \\
\mathsf{lpos} & : & \mathsf{literal}\ (\mathsf{pos}\ B). \\
\mathsf{lneg} & : & \mathsf{literal}\ (\mathsf{neg}\ B). \\
\mathsf{lnew} & : & (\Pi v : v.\mathsf{literal}\ (Fv)) \rightarrow \mathsf{literal}\ (\mathsf{new}\ \lambda u.F). \\
\\
\mathsf{conv} & : & \mathsf{o} \rightarrow \mathsf{o} \rightarrow \mathsf{type}. \\
\mathsf{conv}_1 & : & \mathsf{literal}\ F_1 \rightarrow \mathsf{conv}\ F\ (\mathsf{new}\ \lambda a.\mathsf{new}\ \lambda b.((\mathsf{pos}\ a) \vee (\mathsf{pos}\ b) \vee F) \wedge \\
 & & \qquad ((\mathsf{pos}\ a) \vee (\mathsf{neg}\ b) \vee F) \wedge \\
 & & \qquad ((\mathsf{neg}\ a) \vee (\mathsf{pos}\ b) \vee F) \wedge \\
 & & \qquad ((\mathsf{neg}\ a) \vee (\mathsf{neg}\ b) \vee F)). \\
\mathsf{conv}_2 & : & \mathsf{literal}\ F_1 \rightarrow \mathsf{literal}\ F_2 \rightarrow \mathsf{conv}\ (F_1 \vee F_2)\ (\mathsf{new}\ \lambda a.((\mathsf{pos}\ a) \vee F_1 \vee F_2) \wedge \\
 & & \qquad ((\mathsf{neg}\ a) \vee F_1 \vee F_2)). \\
\mathsf{conv}_3 & : & \mathsf{literal}\ F_1 \rightarrow \mathsf{literal}\ F_2 \rightarrow \mathsf{literal}\ F_3 \\
 & & \rightarrow \mathsf{conv}\ (F_1 \vee F_2 \vee F_3)\ (F_1 \vee F_2 \vee F_3). \\
\mathsf{conv}_\mathsf{n} & : & \mathsf{literal}\ F_1 \rightarrow \mathsf{literal}\ F_2 \rightarrow \mathsf{literal}\ F_3 \rightarrow (\Pi v : \mathsf{v}.\mathsf{conv}\ ((\mathsf{neg}\ v) \vee F_3 \vee F)\ (F'\ v)) \\
 & & \rightarrow \mathsf{conv}\ (F_1 \vee F_2 \vee F_3 \vee F)\ (\mathsf{new}\ \lambda a.((\mathsf{pos}\ a) \vee F_1 \vee F_2) \wedge (F'a)) \\
\mathsf{conv}\wedge & : & \mathsf{conv}\ F_1\ F_1' \rightarrow \mathsf{conv}\ F_2\ F_2' \rightarrow \mathsf{conv}\ (F_1 \wedge F_2)\ (F_1' \wedge F_2').
\end{array}
$$

Figure 2.5: Reduction from SAT to 3-SAT in Elf

for detailed reductions.

Reduction from VERTEX COVER to FEEDBACK ARC SET

The reduction from VERTEX COVER to FEEDBACK ARC SET is given in Figure 2.6. If $G = (V, E)$ is the graph in the instance of VERTEX COVER, the graph $G' = (V', E')$ in the instance of FEEDBACK ARC SET has vertices and edges given by $V' = V \times \{0, 1\}$ and $E' = \{((u, 0), (u, 1)) \mid u \in V\} \cup \{((u, 1), (v, 0)) \mid \{u, v\} \in E\}$.

Reduction from DIRECTED HAMILTON CIRCUIT to UNDIRECTED HAMILTON CIRCUIT

The reduction from DIRECTED HAMILTON CIRCUIT to UNDIRECTED HAMILTON CIRCUIT is shown in Figure 2.7. If $G = (V, E)$ is the directed graph in the instance of DIRECTED HAMILTON CIRCUIT, the vertices and edges in the undi-

$$
\begin{array}{rcl}
\mathsf{vertex} & : & \mathsf{type} \\
\mathsf{edge} & : & \mathsf{vertex} \to \mathsf{vertex} \to \mathsf{type} \\
\mathsf{graph} & : & \mathsf{type} \\
\# & : & \mathsf{graph} \\
\mathsf{newv} & : & (\mathsf{vertex} \to \mathsf{graph}) \to \mathsf{graph} \\
\mathsf{newe} & : & (\mathsf{edge}\ A\ B \to \mathsf{graph}) \to \mathsf{graph} \\
\\
\mathsf{conv} & : & \mathsf{graph} \to \mathsf{graph} \to \mathsf{type} \\
\mathsf{relate} & : & \mathsf{vertex} \to \mathsf{vertex} \to \mathsf{vertex} \to \mathsf{type}. \\
\\
\mathsf{convb} & : & \mathsf{conv}\ \#\ \# \\
\mathsf{conv1} & : & \mathsf{conv}\ (\mathsf{newv}\ \lambda u.G)\ (\mathsf{newv}\ \lambda v.\mathsf{newv}\ \lambda w.\mathsf{newe}\ \lambda e : \mathsf{edge}\ v\ w.G') \\
 & & \leftarrow (\Pi u.\Pi v.\Pi w.\mathsf{relate}\ u\ v\ w \to \mathsf{conv}\ (G\ u); (G'\ v\ w)) \\
\mathsf{conv2} & : & \mathsf{conv}\ (\mathsf{newe}\ \lambda e : \mathsf{edge}\ A\ B.G)\ (\mathsf{newe}\ \lambda e' : \mathsf{edge}\ W_1\ V_2.G') \\
 & & \leftarrow \mathsf{relate}\ A\ V_1\ W_1 \\
 & & \leftarrow \mathsf{relate}\ B\ V_2\ W_2 \\
 & & \leftarrow \mathsf{conv}\ G\ G'
\end{array}
$$

Figure 2.6: Reduction from VERTEX COVER to FEEDBACK ARC SET

rected graph $G' = (V', E')$ is given by $V' = V \times \{0, 1, 2\}$ and

$$E' = \{\{(u,0),(u,1)\},\{(u,1),(u,2)\}\} \mid u \in V\} \cup \{\{(u,2),(v,0)\} \mid (u,v) \in E\}$$

.

2.4 Logical framework: Linear LF (LLF)

We have seen in the previous section how the the logical framework LF can be used to represent instances of NP-complete problems and reductions between them. However, the expressiveness of logical framework LF is quite limited and it is not easy to represent algorithms which may require multiple passes over its inputs. For example, it is quite hard to check if a set of edges given over a vertex set form a clique. In this case, we need to check that for each vertex, whether there is an edge

$$
\begin{array}{rcl}
\mathsf{conv} & : & \mathsf{graph} \to \mathsf{graph} \to \mathsf{type} \\
\mathsf{relate} & : & \mathsf{vertex} \to \mathsf{vertex} \to \mathsf{vertex} \to \mathsf{vertex} \to \mathsf{type}. \\
\\
\mathsf{convb} & : & \mathsf{conv}\ \#\ \# \\
\mathsf{conv1} & : & \mathsf{conv}\ (\mathsf{newv}\ \lambda u.G)\ (\mathsf{newv}\ \lambda v.\mathsf{newv}\ \lambda w.\mathsf{newv}\ \lambda x. \\
 & & \qquad\qquad \mathsf{newe}\ \lambda e_1 : \mathsf{edge}\ v\ w.\mathsf{newe}.\lambda e_2 : \mathsf{edge}\ w\ x.G') \\
 & & \leftarrow (\Pi u.\Pi v.\Pi w.\mathsf{relate}\ u\ v\ w\ x \to \mathsf{conv}\ (G\ u); (G'\ v\ w\ x)) \\
\mathsf{conv2} & : & \mathsf{conv}\ (\mathsf{newe}\ \lambda e : \mathsf{edge}\ A\ B.G)\ (\mathsf{newe}\ \lambda e' : \mathsf{edge}\ X_1\ V_2.G') \\
 & & \leftarrow \mathsf{relate}\ A\ V_1\ W_1\ X_1 \\
 & & \leftarrow \mathsf{relate}\ B\ V_2\ W_2\ X_2 \\
 & & \leftarrow \mathsf{conv}\ G\ G'
\end{array}
$$

Figure 2.7: Reduction from DIRECTED HAMILTON CIRCUIT to UNDIRECTED HAMILTON CIRCUIT

$$
\frac{}{\Gamma, u : A; \cdot \vdash u : A}\ Iax \qquad \frac{}{\Gamma; \Delta, u : A \vdash u : A}\ Lax \qquad \frac{}{\Gamma; \Delta \vdash \langle\rangle : \top}\ Top
$$

$$
\frac{\Gamma; \Delta \vdash M_1 : A \quad \Gamma; \Delta \vdash M_2 : B}{\Gamma; \Delta \vdash \langle M_1, M_2\rangle : A\&B}\ \&I \qquad \frac{\Gamma; \Delta \vdash M : A\&B}{\Gamma; \Delta \vdash \mathsf{FST}M : A}\ \&E_1 \qquad \frac{\Gamma; \Delta \vdash M : A\&B}{\Gamma; \Delta \vdash \mathsf{SND}M : B}\ \&E_1
$$

$$
\frac{\Gamma; \Delta, x : A \vdash M : B}{\Gamma; \Delta \vdash \hat{\lambda} x : A.M : A \multimap B}\ \multimap I \qquad \frac{\Gamma; \Delta \vdash M_1 : B \multimap A \qquad \Gamma; \cdot \vdash M_2 : B}{\Gamma; \Delta \vdash M_1\hat{\ }M_2 : A}\ \multimap E
$$

Figure 2.8: Type checking in Linear LF

between it and every other vertex. Thus, we need to first enumerate over all the vertices and then repeat the edge check process for each of those vertices. In such cases, the formal meta-theory becomes quickly intractable. Linear LF resolves many of these difficulties and is a suitable choice in such problems.

Linear LF [11, 12] is a conservative extension over LF incorporating three connectives from linear logic, namely *multiplicative implication* ($\multimap$), *additive conjunction* (&) and *additive truth* ($\top$). It is a two zone system that explicitly distinguishes between *intuitionistic* assumptions (denoted by Γ), and *linear* assumptions (that play the role of resources and are denoted by Δ). In a derivation, linear assumptions can be used exactly once. The linear fragment of LLF is given in Figure 2.8. Note that the *additive conjunction* (&) allows the use of the same set of linear assumptions for proving both the conjuncts and *multiplicative implication* ($\multimap$) puts a linear assumption into the linear context. The rule *Iax* expresses that an intuitionistic assumption can only be used if no linear assumptions are present as opposed to the *Lax* rule that consumes one single linear assumption. Thus, the type system of LLF extends that of LF by adding type constructors for $\multimap$, & and $\top$ as shown below.

$$\textit{Linear Types}\ A, B \quad ::= \quad A \multimap B \mid A \& B \mid \top$$

LLF supports the *judgments-as-types* methodology for representation and incorporates the aforementioned linear connectives as type constructors. In addition, each rule is endowed with proof objects that correspond to the introduction and elimination forms as shown below.

$$\begin{array}{lll} \multicolumn{3}{l}{\textit{Objects}\ M ::=} \\ & \hat{\lambda} x : A.M \mid M_1 \hat{} M_2 & \text{(Linear functions)} \\ \mid & \langle M_1, M_2 \rangle \mid \mathsf{FST}\ M \mid \mathsf{SND}\ M & \text{(Additive conjunction)} \\ \mid & \langle\rangle & \text{(Additive unit)} \end{array}$$

The intuitionistic and linear contexts are denoted by Γ and Δ respectively and their syntax is given below. We shall occasionally use a single context instead of two separate contexts and denote the linear assumptions as $x \hat{:} A$ and intuitionistic assumptions as $x : A$.

$$\begin{array}{rcl} \textit{Intuitionistic Contexts} & \Gamma & ::= \ \cdot \mid \Gamma, x : A \\ \textit{Linear Contexts} & \Delta & ::= \ \cdot \mid \Delta, x : A \end{array}$$

Thus, a linear LF representation of the judgment $\Gamma; \Delta \vdash J$ is $\ulcorner\Gamma\urcorner \rightarrow \ulcorner\Delta\urcorner \multimap \ulcorner J \urcorner$ where $\ulcorner\Gamma\urcorner$, $\ulcorner\Delta\urcorner$ and $\ulcorner J\urcorner$ are linear LF representations of Γ, Δ and J respectively. And finding a proof is equivalent to finding an object of the corresponding type generated from the language of linear LF objects augmented as above.

2.4.1 Logic programming in LLF

Logic programming semantics of LLF is based on that of LF and the typing rules given in Figure 2.8. In this case, the type system is rewritten as shown below [53]. The LLF objects M are extended with logic variables X_Γ^A valid under $\Gamma \vdash X : A$.

$$
\begin{array}{lcll}
\mathit{Clauses} & D & ::= & P \mid G \rightarrow D \mid G \multimap D \mid \Pi x : A.D \mid D_1 \& D_2 \\
\mathit{Goals} & G & ::= & \top \mid P \mid D \rightarrow G \mid D \multimap G \mid \Pi x : A.G \mid G_1 \& G_2 \\
\mathit{Predicates} & P & ::= & \mathsf{a} \mid PM
\end{array}
$$

The additional proof search rules for the new type constructors introduced in linear LF are given in Figure 2.9. The judgments in this case are $\Sigma, \Delta \vDash G$ and $\Sigma, \Delta \vDash D \gg P$ for goals and clauses respectively. We explicitly mention the linear context Δ which keeps track of the linear assumptions. The rules are quite closely based on the linear logic semantics of the corresponding operators. Thus, $\mathsf{AND_G}$ rule for *additive conjunction* (&) duplicates both the *linear* and *intuitionisitic* parts of the program, and it follows from TOP that the goal $\top$ is always provable even if the program Σ has linear assumptions or clauses. Moreover, when the ATOM rule from Figure 2.1 is applied for LLF, the linear context should be empty.

2.5 Reduction from 3-SAT to CHROMATIC

In this section, we shall describe in complete detail a reduction between two NP-complete problems, namely satisfiability of boolean formulas in conjunctive normal form with 3 literals per clause (3-SAT) and chromatic number of a graph (CHROMATIC). We shall present the reduction as a inference system and also the corresponding logic program in the logical framework linear LF. Finally, we shall give a

Goals:

$$\frac{\Sigma, \Delta \vDash D \gg P, \theta}{\Sigma, \Delta, \mathsf{c}\hat{:}D \vDash P, \theta}\ \textsf{LIN-CLAUSE} \qquad\qquad \frac{}{\Sigma, \Delta \vDash \top, \cdot}\ \textsf{TOP}$$

$$\frac{\mathsf{c}\ \textsf{new} \quad \Sigma, \Delta, \mathsf{c}\hat{:}D \vDash G, \theta}{\Sigma, \Delta \vDash D \multimap G, \theta}\ \textsf{LIN-IMP} \qquad \frac{\Sigma, \Delta \vDash G_1, \theta_1 \quad \Sigma, \Delta \vdash G_2, \theta_2}{\Sigma, \Delta \vDash G_1 \& G_2, \theta_1 \cup \theta_2}\ \textsf{AND}_{\textsf{G}}$$

Clauses:

$$\frac{\theta \in \textsf{unify}(P, Q)}{\Sigma \vDash Q \gg P, \theta}\ \textsf{ATOM}$$

$$\frac{\Sigma, \Delta_1 \vDash D \gg P, \theta \quad \Sigma, \Delta_2 \vDash \theta(G), \theta'}{\Sigma, \Delta_1, \Delta_2 \vDash G \multimap D \gg P, \theta'(\theta)}\ \textsf{LIN-SUBGOAL}$$

$$\frac{\Sigma, \Delta \vDash D_1 \gg P, \theta}{\Sigma, \Delta \vDash D_1 \& D_2 \gg P, \theta}\ \textsf{AND}_{\textsf{L}_1} \qquad \frac{\Sigma, \Delta \vDash D_2 \gg P, \theta}{\Sigma, \Delta \vDash D_1 \& D_2 \gg P, \theta}\ \textsf{AND}_{\textsf{L}_2}$$

Figure 2.9: Logic programming in LLF (additional rules)

correctness proof, i.e. a proof that the reduction maps *Yes* instances of 3-SAT to *Yes* instances of CHROMATIC and vice-versa.

The definition below gives a description of the problem CHROMATIC in a style that is commonly used in theoretical computer science. Figure 2.10 presents an inference system for identifying *Yes* instances of the problem.

Definition 2.5.1 (CHROMATIC)**.** Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, and a positive integer C.

QUESTION: Is G C-colorable, i.e., does there exist a function

$$\chi : V \to \{1, 2, \ldots, C\}$$

such that $\chi(u) \neq \chi(v)$ whenever $\{u, v\} \in E$?

Following [39] and [44] we sketch the reduction first informally before formalizing it further. Suppose, we are given an instance of 3-SAT as described in Definition 1.2.2.

$$\frac{}{\eta \vdash \# \; C \; \textsf{COLORING}} \; cgempty$$

$$\frac{C' \leq C \quad \eta, v \rightarrow C' \vdash G \; C \; \textsf{COLORING}}{\eta \vdash \textsf{newv} \; v.G \; C \; \textsf{COLORING}} \; cgvertex$$

$$\frac{C_1 \leq C \quad C_2 \leq C \quad C_1 \neq C_2 \quad \eta, A \rightarrow C_1, B \rightarrow C_2 \vdash G \; C \; \textsf{COLORING}}{\eta, A \rightarrow C_1, B \rightarrow C_2 \vdash \textsf{newe} \; e : (A, B).G \; C \; \textsf{COLORING}} \; cgedge$$

$$\frac{\eta \vdash G_1 \; C \; \textsf{COLORING} \quad \eta \vdash G_2 \; C \; \textsf{COLORING}}{\eta \vdash (G_1 \cup G_2) \; C \; \textsf{COLORING}} \; cgunion$$

Figure 2.10: Inference rules for *Yes* instances of CHROMATIC

1. For every variable u_i, create vertices v_i, v_i' and x_i. For every clause c_j, create a vertex c_j in the graph.

2. Connect the edges between these vertices as below:

   (a) For every i, add an edge $\{v_i, v_i'\}$.

   (b) For every i and j, add an edge $\{x_i, x_j\}$ when $i \neq j$.

   (c) For every i and j, add an edge $\{v_i, x_j\}$ and $\{v_i', x_j\}$ when $i \neq j$.

   (d) For every i and j, add an edge $\{c_i, v_j\}$ if u_j does not appear in c_i and an edge $\{c_i, v_j'\}$ if $\bar{u}_j$ does not appear in c_i.

It is not hard to see that if the Boolean formula with n variables has a truth assignment then the graph has a $n+1$-coloring and vice versa. Essentially, the construction given above – connecting v_i's and v_i''s to the clique on x_i's – forces creation of n true colors and one false color.

A formalization of this reduction, again in form of a inference system is given in Figure 2.11. The main judgment is of the form $\Gamma; \Delta \vdash K \diamond F \Rightarrow_C C', G$, where Γ is a list of assumptions of the form (u, v, v', x) representing a relationship between a free Boolean variable u in F and its corresponding free graph vertices v, v' and x in

G. Δ is a list of all distinct Boolean variables used in the Boolean formula (see rule *c_new*). Eventually, it will contain all free Boolean variables in F. We also maintain two counters C and C': C is incremented every time we see a new variable and C' corresponds to the total number of variables. All clauses that are contained in the Boolean formula prompt the insertion of edges into the graph corresponding to step (d) of the conversion algorithm. We achieve this by maintaining a "continuation" stack of clauses that were already encountered but not yet processed. The language of continuation stack is given below. Here $*$ is the initial continuation, indicating that we have no more clauses left.

$$\text{Continuations} \quad K \quad ::= \quad * \mid K; f$$

Thus, these inference rules allow us to build a valid deduction for a judgment $\Gamma; \Delta \vdash K \diamond F \Rightarrow_C C', G$ if and only if the conversion algorithm given above converts the Boolean formula represented by combining the clauses in F and K to the graph G; C should always be more than the sum of the free Boolean variables in F and K; and C' is the total number of Boolean variables in F. Since C is updated every time we see a new variable, the invariant given in the lemma below always holds.

Lemma 2.5.1 (Invariant of the Reduction)**.** *Given any continuation K, Boolean formula F, graph G and colors C, C': If $\mathcal{D} :: \Gamma \vdash K \diamond F \Rightarrow_C C', G$ then $C \leq C'$.*

Proof. A straightforward induction. The theorem is denoted by the LF type family lemma2.5.1 and the encoding of the proof is given in Schürmann and Shah [66]. Note that lemma2.5.1 is renamed as `conv_invariant` in Schürmann and Shah [66]. □

The edges in step (a) are added immediately when we encounter a new variable in rule *c_new*, the edges in step (b) are added through the inference rules associated

$$\frac{\Gamma, (u, v, v', x); \Delta, u \vdash K \diamond F \Rightarrow_{C+1} C', G}{\Gamma; \Delta \vdash K \diamond \mathsf{new}\ u.F \Rightarrow_C C', \mathsf{newv}\ v\ v'\ x.\mathsf{newe}\ e : (v, v').G}\ c\_new$$

$$\frac{\Gamma; \Delta \vdash K; F \diamond F' \Rightarrow_C C', G}{\Gamma; \Delta \vdash K \diamond F \wedge F' \Rightarrow_C C', G}\ c\_\wedge$$

$$\frac{\Gamma; \Delta \vdash K; (F_1 \vee F_2 \vee F_3) \Rightarrow G_1 \quad \Gamma; \Delta \vdash G_2\ \mathsf{CLIQUE} \quad \Gamma; \Delta \vdash G_3\ \mathsf{VARS\text{-}TO\text{-}CLIQUE}}{\Gamma; \Delta \vdash K \diamond (F_1 \vee F_2 \vee F_3) \Rightarrow_C C, G_1 \cup G_2 \cup G_3}\ c\_\vee$$

$$\frac{}{\Gamma; \Delta \vdash * \Rightarrow \#}\ c'\_*$$

$$\frac{\Gamma; \Delta \vdash K \Rightarrow G_1 \quad \Gamma; \Delta \vdash F \Rightarrow G_2}{\Gamma; \Delta \vdash K; F \Rightarrow G_1 \cup G_2}\ c'\_;$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2), (u_3, v_3, v_3', x_3); \Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma, (u_1, v_1, v_1', x_1), \ldots, (u_3, v_3, v_3', x_3); \Delta, u_1, u_2, u_3 \vdash (\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_2) \vee (\mathsf{pos}\ u_3) \\ \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1 : (c, v_1')\ e_2 : (c, v_2')\ e_3 : (c, v_3').G\end{array}}\ c''5.1$$

(39 similar rules. See Appendix B)

$$\frac{}{\Gamma; \cdot \vdash C \downarrow \#}\ c'''\_\#$$

$$\frac{\Gamma, (u, v, v', x); \Delta \vdash C \downarrow G}{\Gamma, (u, v, v', x); \Delta, u \vdash C \downarrow \mathsf{newe}\ e : (C, v)\ e' : (C, v').G}\ c'''\_v$$

$$\frac{}{\Gamma; \cdot \vdash \#\ \mathsf{CLIQUE}}\ clique\_\#$$

$$\frac{\Gamma, (u, v, v', x); \Delta \vdash G_1\ \mathsf{CLIQUE} \quad \Gamma, (u, v, v', x); \Delta \vdash \mathsf{CONNECTX}\ x\ G_2}{\Gamma, (u, v, v', x); \Delta, u \vdash (G_1 \cup G_2)\ \mathsf{CLIQUE}}\ clique\_v$$

$$\frac{}{\Gamma; \cdot \vdash \#\ \mathsf{VARS\text{-}TO\text{-}CLIQUE}}\ v2c\_\#$$

$$\frac{\Gamma, (u, v, v', x); \Delta \vdash G_1\ \mathsf{VARS\text{-}TO\text{-}CLIQUE} \quad \begin{array}{l}\Gamma, (u, v, v', x); \Delta \vdash \mathsf{CONNECTX}\ v\ G_2 \\ \Gamma, (u, v, v', x); \Delta \vdash \mathsf{CONNECTX}\ v'\ G_3 \\ \Gamma, (u, v, v', x); \Delta \vdash \mathsf{CONNECTV}\ x\ G_4\end{array}}{\Gamma, (u, v, v', x); \Delta, u \vdash (G_1 \cup G_2 \cup G_3 \cup G_4)\ \mathsf{VARS\text{-}TO\text{-}CLIQUE}}\ v2c\_v$$

$$\frac{}{\Gamma; \cdot \vdash \mathsf{CONNECTV}\ X\ \#}\ connectV\_\#$$

$$\frac{\Gamma, (u, v, v', x'); \Delta \vdash \mathsf{CONNECTV}\ X\ G}{\Gamma, (u, v, v', x'); \Delta, u \vdash \mathsf{CONNECTV}\ X\ \mathsf{newe}\ e : (X, v)\ e' : (X, v').G}\ connectV\_v$$

$$\frac{}{\Gamma; \cdot \vdash \mathsf{CONNECTX}\ X\ \#}\ connectX\_\#$$

$$\frac{\Gamma, (u, v, v', x'); \Delta \vdash \mathsf{CONNECTX}\ X\ G}{\Gamma, (u, v, v', x'); \Delta, u \vdash \mathsf{CONNECTX}\ X\ \mathsf{newe}\ e : (X, x').G}\ connectX\_v$$

Figure 2.11: Linear LF representation of reduction from 3-SAT to CHROMATIC

with judgment $\Gamma; \Delta \vdash G$ CLIQUE and the edges in step (c) are added through the inference rules associated with $\Gamma; \Delta \vdash G$ $\mathsf{VARS\text{-}TO\text{-}CLIQUE}$.

In step (d), we create a vertex corresponding to every clause and add edges connecting the clause to vertices corresponding to literals not in the clause. These edges are added through the inference rules associated with $\Gamma; \Delta \vdash K; F \Rightarrow G$. We are only considering clauses with three literals and hence there are 40 different kinds of clauses: each of the 3 literals in a clause can have a variable appearing as itself or as its complement, giving us 8 choices and each clause can have up to 3 distinct variables, giving us 5 choices[1]. For the sake of conciseness we give only one representative rule *c"5.1* in Figure 2.11, the other 39 rules are given in Appendix B.

The predicate $\mathsf{CONNECTX}$ adds an edge between its first argument and every vertex among the resource in Δ. We note that once we access a vertex in Δ, it is automatically consumed (see for example rules *c"5.1*, *c'_;*, *clique_v*, *v2c_v*, *connectV_v*, and *connectX_v*). Thus, Δ's properties are best described as those of the linear context in the sense of linear logic [27].

Cliques are built recursively, using Δ as a structure over which to iterate. Every vertex in Δ is connected through an edge to every other vertex in that context defining a clique (see rule *clique_v*).

If a Boolean formula F has n variables, 0 free variables and m clauses, then the number of inference rules used in the derivation $\cdot; \cdot \vdash *\diamond F \Rightarrow_{\mathsf{Z}} C, G$ are $m+n+1$ (each new variable corresponds to the inference rule *c_new*, each clause corresponds to the inference rule *conv*$\wedge$ and there is one base case). Further, the deductions for the judgments $\Gamma; \Delta \vdash K \Rightarrow G_1$, $\Gamma; \Delta \vdash G_2$ CLIQUE, and $\Gamma; \Delta \vdash G_3$ $\mathsf{VARS\text{-}TO\text{-}CLIQUE}$ have height $O(n)$. Hence, the total number of inference rules used in the derivation

[1]When variables appear only positively in each of the 3 literals, the 5 choices are: $(\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_1)$, $(\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_2)$, $(\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_2) \vee (\mathsf{pos}\ u_1)$, $(\mathsf{pos}\ u_2) \vee (\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_1)$, $(\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_2) \vee (\mathsf{pos}\ u_3)$

$$
\begin{array}{rcl}
\mathsf{satK} & : & \mathsf{cont} \rightarrow \mathsf{type}. \\
\mathsf{satK}{*} & : & \mathsf{satK}\ * . \\
\mathsf{satK;} & : & \mathsf{sat}\ F \rightarrow \mathsf{satK}\ K \rightarrow \mathsf{satK}\ (K;F)
\end{array}
$$

Figure 2.12: Encoding of continuation satisfiability.

of the reduction is $O(m+n)$. Thus, the proposed reduction algorithm is a polynomial time reduction.

2.5.1 Representation of the reduction in LLF

The Twelf code for the inference rules that encode 3SAT (Figure 1.2) is given in Figure 2.4 and for CHROMATIC (Figure 2.10) is given in Figure 2.13. Continuations are represented as objects of type `cont`. We write `*` for the initial continuation and $K;F$ for a continuation stack with F being the top element.

$$
\begin{array}{rcl}
* & : & \mathsf{cont}. \\
; & : & \mathsf{cont} \rightarrow \mathsf{o} \rightarrow \mathsf{cont}.
\end{array}
$$

The notion of satisfiability generalizes to continuations, and is given in Figure 2.12.

Each of the two problems is hypothetical in nature, 3SAT for example relies on correctly selecting a truth assignment for rules *satt* and *satf* and CHROMATIC on correctly assigning a color to a vertex in rules *cgvertex* and *cgedge*. In LLF, the encoding of these hypotheses gives rise to new two type families hyp and colorvertex, respectively.

The 3-SAT CHROMATIC reduction (Figure 2.11) is given in Figure 2.19 and encoded as a relation over Boolean formulas F, colors C and C', continuation K and graph G. It is implemented by the type family

$$\mathsf{conv} : \mathsf{o} \rightarrow \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{cont} \rightarrow \mathsf{graph} \rightarrow \mathsf{type}.$$

$$
\begin{array}{rcl}
\mathsf{colorvertex} & : & \mathsf{vertex} \longrightarrow \mathsf{nat} \longrightarrow \mathsf{type}. \\
\mathsf{coloring} & : & \mathsf{nat} \longrightarrow \mathsf{graph} \longrightarrow \mathsf{type}. \\
\mathsf{cg\#} & : & \mathsf{coloring}\ N\ \#. \\
\mathsf{cgvertex} & : & \mathsf{coloring}\ C\ (\mathsf{newv}\ \lambda v.(Gv)) \\
 & & \leftarrow (C \leq C') \\
 & & \leftarrow (\Pi.v : \mathsf{vertex}.\mathsf{colorvertex}\ v\ C' \rightarrow \mathsf{coloring}\ C\ (G\ v)). \\
\mathsf{cgedge} & : & \mathsf{coloring}\ C\ (\mathsf{newe}\ \lambda e : \mathsf{edge}\ A\ B.G) \\
 & & \leftarrow \mathsf{colorvertex}\ A\ C_1 \leftarrow \mathsf{colorvertex}\ A\ C_2 \\
 & & \leftarrow C_1 \neq C_2 \leftarrow C_1 \leq C \leftarrow C_2 \leq C \\
 & & \leftarrow \mathsf{coloring}\ C\ G. \\
\mathsf{cgunion} & : & \mathsf{coloring}\ C\ (G1 + G2) \\
 & & \leftarrow \mathsf{coloring}\ C\ G1 \\
 & & \leftarrow \mathsf{coloring}\ C\ G2.
\end{array}
$$

Figure 2.13: Encoding of coloring

$$
\begin{array}{rcl}
\mathsf{connectX} & : & \mathsf{vertex} \longrightarrow \mathsf{graph} \longrightarrow \mathsf{type}. \\
\mathsf{connectX\_\#} & : & \mathsf{connectX}\ X\ \#. \\
\mathsf{connectX\_v} & : & (\mathsf{var}\ U \multimap \mathsf{connectX}\ X\ (\mathsf{newe}\ \lambda e : \mathsf{edge}\ X\ X'.G)) \\
 & & \leftarrow \mathsf{relate}\ U\ \_\ \_\ X' \\
 & & \leftarrow \mathsf{connectX}\ X\ G. \\
 & & \\
\mathsf{clique} & : & \mathsf{graph} \longrightarrow \mathsf{type}. \\
\mathsf{clique\_\#} & : & \mathsf{clique}\ \#. \\
\mathsf{clique\_v} & : & (\mathsf{var}\ U \multimap \mathsf{clique}\ (G + G')) \\
 & & \leftarrow \mathsf{clique}\ G\ \&\ \mathsf{connectX}\ X\ G' \\
 & & \leftarrow \mathsf{relate}\ U\ \_\ \_\ X.
\end{array}
$$

Figure 2.14: Encoding of CLIQUE

The assumptions (u, v, v', x) relating a Boolean variable u with graph vertices v, v' and x are implemented by the type family

$$\mathsf{relate} : \mathsf{u} \rightarrow \mathsf{vertex} \rightarrow \mathsf{vertex} \rightarrow \mathsf{vertex} \rightarrow \mathsf{type}.$$

Recall, that our reduction is based on the construction of cliques, which we encode in LLF in terms of a type families clique shown in Figure 2.14 and vars2clique in Figure 2.15. Note in these two figures, how the harmless looking & in clique_v is respon-

$$
\begin{array}{rcl}
\mathsf{connectV} & : & \mathsf{vertex} \rightarrow \mathsf{graph} \rightarrow \mathsf{type}.\\
\mathsf{connectV\_\#} & : & \mathsf{connectV}\ X\ \#.\\
\mathsf{connectV\_v} & : & \mathsf{connectV}\ X\ (\mathsf{newe}\ \lambda e : \mathsf{edge}\ V\ X.\mathsf{newe}\ \lambda e : \mathsf{edge}\ V'\ X.G)\\
 & & \circ\!\!-\ \mathsf{var}\ U\\
 & & \leftarrow \mathsf{relate}\ U\ V\ V'\ \_\\
 & & \leftarrow \mathsf{connectV}\ X\ G.\\
 & & \\
\mathsf{vars2clique} & : & \mathsf{graph} \rightarrow \mathsf{type}.\\
\mathsf{v2c\_\#} & : & \mathsf{vars2clique}\ \#.\\
\mathsf{v2c\_v} & : & (\mathsf{var}\ U \multimap \mathsf{vars2clique}\ (G1 + G2 + G3 + G4))\\
 & & \leftarrow \mathsf{vars2clique}\ G1\ \&\ \mathsf{connectX}\ V\ G_2\\
 & & \quad \&\ \mathsf{connectX}\ V'\ G_3\ \&\ \mathsf{connectV}\ X\ G_4\\
 & & \leftarrow \mathsf{relate}\ U\ V\ V'\ X.
\end{array}
$$

Figure 2.15: Encoding of VARS-TO-CLIQUE

$$
\begin{array}{rcl}
\mathsf{conv'''} & : & \mathsf{vertex} \rightarrow \mathsf{graph} \rightarrow \mathsf{type}.\\
\mathsf{c'''\_\#} & : & \mathsf{conv'''}\ C\ \#.\\
\mathsf{c'''\_v} & : & (\mathsf{var}\ U \multimap \mathsf{conv'''}\ C\ (\mathsf{newe}\ \lambda e : \mathsf{edge}\ C\ V.\lambda e' : \mathsf{edge}\ C\ V'.G))\\
 & & \leftarrow \mathsf{conv'''}\ C\ G\\
 & & \leftarrow \mathsf{relate}\ U\ V\ V'\ \_.
\end{array}
$$

Figure 2.16: Encoding of the $\Gamma, \Delta \vdash K \Rightarrow G$.

sible for duplicating the context Δ in rule *clique_v* in Figure 2.11. In fact, all assumptions in Δ are treated as resources (to control iteration), and hence exclusive represented within the linear context by assumptions of type family var : vertex $\rightarrow$ type.

The three auxiliary judgments $\Gamma, \Delta \vdash K \Rightarrow G$; $\Gamma, \Delta \vdash F \Rightarrow G$; and $\Gamma, \Delta \vdash C \downarrow G$ are given in Figures 2.16-2.18, respectively, leading up to the encoding of the reduction `conv` that is given in Figure 2.19.

2.6 Evaluation of the approach

This discussion has highlighted several advantages of using LF and LLF, and a few difficulties that we face when we attempt to formalize NP-complete problems and their reductions successfully in these proof assistants. The drawbacks largely

$$
\begin{array}{rcl}
\mathsf{conv''} & : & \mathsf{o} \rightarrow \mathsf{graph} \rightarrow \mathsf{type}. \\
\mathsf{c''51} & : & \mathsf{conv''}\ ((\mathsf{pos}\ U_1) \vee (\mathsf{pos}\ U_2) \vee (\mathsf{pos}\ U_3)) \\
 & & \qquad (\mathsf{newv}\ \lambda c.\mathsf{newe}\ \lambda e_1 : \mathsf{edge}\ c\ V_1'.e_2 : \mathsf{edge}\ c\ V_2'. \\
 & & \qquad\qquad\qquad\qquad e_3 : \mathsf{edge}\ c\ V_3'.(G\ c)) \\
 & & \multimap \mathsf{var}\ U_3 \multimap \mathsf{var}\ U_2 \multimap \mathsf{var}\ U_1 \\
 & & \leftarrow \mathsf{relate}\ U_1\ V_1\ V_1'\ \_ \leftarrow \mathsf{relate}\ U_2\ V_2\ V_2'\ \_ \leftarrow \mathsf{relate}\ U_3\ V_3\ V_3'\ \_ \\
 & & \leftarrow (\Pi c : \mathsf{vertex}.\mathsf{conv'''}\ c\ (Gc)).
\end{array}
$$

Figure 2.17: Encoding of $\Gamma, \Delta \vdash F \Rightarrow G$ (case 5 of 40).

$$
\begin{array}{rcl}
\mathsf{conv'} & : & \mathsf{cont} \rightarrow \mathsf{graph} \rightarrow \mathsf{type}. \\
\mathsf{c'\_*} & : & \mathsf{conv'}\ *\ \# \multimap \top. \\
\mathsf{c'\_;} & : & \mathsf{conv'}\ K\ G_1\ \&\ \mathsf{conv''}\ F\ G_2 \rightarrow \mathsf{conv'}\ (K;F)\ (G_1 + G_2)
\end{array}
$$

Figure 2.18: Encoding of the $\Gamma, \Delta \vdash C \downarrow G$.

fall into three main categories: representation of mathematical entities like graphs, static analysis of complexity of algorithms and representation of correctness of the reduction.

Representation of mathematical entities: The main challenge in representing graphs as objects lies in capturing isomorphisms between graphs in type theory. Neither LF nor LLF provide a notion of definitional equality that is compatible with graph isomorphism. The solution adopted in this paper is to consider two graphs equivalent if and only if the order in which vertices and edges were introduced into the graph coincides.

Further, many standard operations that are usually performed on a graph like addition or deletion of a vertex, iteration over subgraphs, edges and vertices are necessary to express basic reductions. Our solution for incorporating iteration is based on linear constraints [12], where the linear context enforces complete traversal over the set of edges or vertices. Ideally, however, operations of this kind should be directly supported by the underlying logical framework. In many cases, we also need to represent operations like graph complementation

$$
\begin{array}{rcl}
\mathsf{conv} & : & \mathsf{o} \rightarrow \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{cont} \rightarrow \mathsf{graph} \rightarrow \mathsf{type}. \\[1em]
\mathsf{c\_new} & : & \mathsf{conv}\ (\mathsf{new}\ F)\ C\ C'\ K \\
& & \qquad (\mathsf{newv}\ \lambda v.\mathsf{newv}\ \lambda v'.\mathsf{newv} \lambda x.\mathsf{newe}\ \lambda e : \mathsf{edge}\ v\ v'.(G\ v\ v'\ x)) \\
& & \leftarrow (\Pi u : \mathsf{v}.\Pi v : \mathsf{vertex}.\Pi v' : \mathsf{vertex}.\Pi x : \mathsf{vertex}.\mathsf{relate}\ u\ v\ v'\ x \\
& & \qquad \mathsf{var}\ u \multimap \mathsf{conv}\ (F\ u)\ (\mathsf{s}\ C)\ C'\ K\ (G\ v\ v'\ x)) \\
\mathsf{c\_}\wedge & : & \mathsf{conv}\ F'\ C\ C'\ (K; F)\ G \rightarrow \mathsf{conv}\ (F \wedge F')\ C\ C'\ K\ G \\
\mathsf{c\_}\vee & : & \mathsf{conv}\ (F_1 \vee F_2 \vee F_3)\ C\ C\ K\ (G_1 + G_2 + G_3) \\
& & \leftarrow \mathsf{conv}'\ (K; (F_1 \vee F_2 \vee F_3))\ G_1\ \&\ \mathsf{clique}\ G_2\ \&\ \mathsf{vars2clique}\ G_3
\end{array}
$$

Figure 2.19: Encoding of the reduction

or test if an element *does not* belong to a set of elements. Such operations are hard to represent directly in LF and we need to use the linear assumption in a cumbersome manner to implement such operations.

The following chapter presents the concurrent logical framework CLF which has many built-in language primitives that allow encoding of many of these standard graph operations making the corresponding proofs more natural.

Static analysis of complexity: Any valid reduction between two NP complete problems must be a polynomial time reduction. The reduction of a Boolean formula F to a graph G is denoted by the derivation $\mathcal{D} :: \Gamma; \Delta \vdash K \diamond F \Rightarrow_C C', G$. Thus, given an appropriate notion of *size* of $\mathcal{D}$ and F, it is possible to argue that the reduction is polynomial time if the size of $\mathcal{D}$ is a polynomial in the size of F. Furthermore, since we are choosing a framework with higher-order terms, we need to develop notions of polynomial time when higher-order unification is permitted and these terms are reduced to their canonical forms. We shall discuss our solutions to this problem in Chapters 4 and 5.

Representation of correctness of the reduction: A good representation system for NP-complete problems should also be able to represent the theorem and the corresponding proof that a given reduction is *correct*, i.e. it converts *Yes* instances of the first problem to the *Yes* instances of the second problem and vice-versa. We have given a detailed correctness proof of the reduction between 3-SAT and CHROMATIC in Schürmann and Shah [66, 67]. The correctness proof exposes many limitations of current logical frameworks and proposes several directions for future research.

Chapter 3

Representing NP-complete problems II

In the previous chapter, we have showed how logical frameworks LF and linear LF (LLF) can be used for representing NP-complete problems, reductions between those problems and the corresponding proofs of correctness. While the approach gives the user tremendous flexibility in choosing the representation primitives for each NP-complete problem, we illustrated that entities such as graphs have no natural representation within the system. In this chapter, we shall attempt to address this difficulty by using an advanced variant of LF named concurrent logical framework (CLF) [13, 45, 71].

CLF, a superset of LLF, has several advantages over LF and LLF. It has both forward-chaining and backward-chaining models of computation. The backward-chaining model of computation is the traditional Prolog-like proof search generalized by Miller, *et al.* [53] to a general class of abstract logic programming languages (ALPL). In the forward-chaining model of computation [24, 25], there is an initial database of assertions and a set of logic program clauses. The program execution computes the set of all assertions that are derivable from the initial database using the logic program clauses.

The two modes of computation are separated by using monadic types $\{S\}$. In the original presentation of CLF, objects of monadic types were meant to represent concurrent computations and therefore they have forward-chaining operational semantics. As we shall see in the following section, reduction algorithms which can be naturally represented in CLF exploit the inherent concurrency in the algorithm and thus can be represented quite elegantly.

This chapter is structured as follows. First, we shall give some motivation for choosing CLF to represent NP-complete problems based on mathematical entities like graphs. Then we shall describe CLF and its operational semantics. Finally, we will give several examples of CLF representations of reductions between NP-complete problems.

3.1 Challenges in representing graphs

Logical frameworks LF and LLF suffer from two main limitations when representing NP-complete problems and their reductions. We have already touched upon the first one. Many NP-complete problems are defined in terms of mathematical entities like graphs, matchings, and sets. To the best of our knowledge, such entities have no natural representation in these logical frameworks. We have seen that any attempt to represent these entities imposes an ordering which did not exist in their conceptual representation. For example, the representation of graphs shown in Section 2.2 would require traversal over the entire expression to access an edge that may just happen to be deep inside the expression. Thus, simple graph operations like edge or vertex deletion have complex representations in LF and LLF requiring an explicit search over the entire expression corresponding to a graph.

The second limitation concerns the representation of reduction algorithms in these logical frameworks. Many reduction algorithms when described in theoretical computer science discourse can be described most naturally as concurrent computations. For example, consider the step 2(a) of the reduction algorithm from 3-SAT to CHROMATIC described in Section 2.5:

2(a) For every i, add an edge $\{v_i, v_i'\}$

When we represented this step in LLF, we had to serialize the operation as there is no natural way for representing concurrent operations in LLF.

We address this first difficulty by using intuitionistic and linear contexts to represent problem instances. We will use the classification system for contexts called the *world system* [65] to precisely specify the number and kind of intuitionistic and linear assumptions required to define a single problem instance.

The second problem is addressed by simply representing the concurrent steps of the reduction algorithm using the forward-chaining semantics of CLF wherever possible. The parts of the algorithm which are not concurrent can be represented using the standard backward-chaining semantics that we have used so far.

3.1.1 Related work

Several researchers have attempted to develop solutions to address the aforementioned problems that arise when graph algorithms are to be represented in functional and logic programming languages. Martin Erwig has introduced the concept of *active patterns* [21, 22] to express traversal over data-types representing graphs. For example, consider the following data-type definition for *set* in ML syntax:

```
datatype 'a set = Empty | Add 'a * 'a set
```

The function definition for membership test would be:

```
fun member x (Add (y,ys)) = if x=y then true else member x ys
  | member x Empty        = false
```

An *active pattern* `Add'(y,ys)` transforms the term so that `y` is not the head of the original term, but could be any member within the set. Thus, the new function definition is simply:

```
fun member x (Add' (x,ys)) = true
  | member x xs = false
```

Concurrent LF contexts also have a similar property. In addition, CLF also has a new let-expression which incorporates the concept of active patterns much more directly.

Cardelli, et al. [9] have introduced a query language designed for analyzing and manipulating graphs and graphical data. The language is based on their previous work on spatial logic [7, 8] meant for modeling concurrency. The most important connective in their query language is the composition operator | and its corresponding structural form $\phi \mid \psi$ that describes a graph that can be split into two parts: one part satisfying ϕ and other part satisfying ψ. This simple connective is quite powerful and enables them to express many graph properties and operations quite elegantly.

The semantics of *multiplicative product* operator $\otimes$ of CLF is very similar to that of the composition operator | of Cardelli's spatial logic. We shall give several examples later in this chapter where $\otimes$ greatly simplifies expression of graph properties and algorithms.

3.2 Logical framework: Concurrent LF (CLF)

The basic structure of CLF [13, 45, 71] is similar to that of LF: *objects* are classified by *types* and *types* are classified by *kinds*. Like LF, *kinds* classify *type constructors* P. CLF differs from LF and LLF in that it has two categories of types: the *asynchronous types* A and the *synchronous types* S. The asynchronous types include all the type constructors of LF and LLF, as well as a new *monadic type constructor* written as $\{S\}$. The synchronous types which are allowed only within the monadic constructor includes further type constructors of intuitionistic linear logic. Intuitively, concurrent computations are modeled using these synchronous types. The syntax of CLF kinds and types is given below.

$$
\begin{array}{llcl}
\textit{Kinds} & K & ::= & \mathsf{type} \mid \Pi x : A.K \\
\textit{Asynchronous types} & A, B, C & ::= & \underbrace{\Pi x : A.B \mid A \to B \mid P}_{\text{LF types}} \mid \\
 & & & \underbrace{A \multimap B \mid \top \mid A \& B}_{\text{LLF types}} \mid \{S\} \\
\textit{Atomic type constructors} & P & ::= & \mathsf{a} \mid P N \\
\textit{Synchronous types} & S & ::= & S_1 \otimes S_2 \mid 1 \mid \exists x : A.S \mid A
\end{array}
$$

$$
\begin{array}{llcl}
\textit{Normal Objects} & N & ::= & \lambda x.N \mid \underbrace{\hat{\lambda} x.N \mid \langle N_1, N_2\rangle \mid \langle\rangle}_{\text{LLF normal objects}} \mid \{E\} \mid R \\
\textit{Atomic Objects} & R & ::= & \mathsf{c} \mid x \mid RN \mid \underbrace{R\hat{\ }N \mid \pi_1 R \mid \pi_2 R}_{\text{LLF atomic objects}} \\
\textit{Expressions} & E & ::= & \mathsf{let}\ \{p\} = R\ \mathsf{in}\ E \mid M \\
\textit{Monadic Objects} & M & ::= & M_1 \otimes M_2 \mid 1 \mid [N, M] \mid N \\
\textit{Patterns} & p & ::= & p_1 \otimes p_2 \mid 1 \mid [x, p] \mid x
\end{array}
$$

The first two categories of objects, namely *normal objects* and *atomic objects* correspond to the quasi-canonical and quasi-atomic forms of LF object, respectively as described by Harper and Pfenning [32]. The most notable addition in CLF is that of the constructor $\{E\}$ associated with monadic types. These *expressions* and *monadic objects* correspond to concurrent computations described by their respective monadic types. The notion of definitional equality for these expressions is based on *active patterns* that we mentioned earlier. These objects are subject to *permutative conversions* by which the monadic bindings can be reordered. Thus the following two expressions are equivalent in CLF under the condition that no variable bound by p_1 can appear free in R_2, and vice-versa.

$$
(\mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ \mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ E) = (\mathsf{let}\ \{p_2\} = R_2\ \mathsf{in}\ \mathsf{let}\ \{p_1\} = R_1\ \mathsf{in}\ E)
$$

If we think of each let binding as a single computation step, the computation steps appearing in a single expression that are independent in the above sense can be thought of as occurring concurrently.

3.2.1 Logic programming

$$
\begin{array}{lrcl}
\textit{Clauses} & D & ::= & P \mid \{S_D\} \mid G \rightarrow D \mid G \multimap D \mid \Pi x : A.D \mid D_1 \& D_2 \\
\textit{Goals} & G & ::= & \top \mid P \mid \{S_G\} \mid D \rightarrow G \mid D \multimap G \mid \Pi x : A.G \mid G_1 \& G_2 \\
\textit{Predicates} & P & ::= & \mathsf{a} \mid PM
\end{array}
$$

The operational semantics of CLF describe in detail in Lopez, *et al.* [45] combines both forward-chaining and backward-chaining computational paradigms. The objects M are extended with logic variables in a manner similar to that for LF and LLF. As we have described earlier in the chapter, the monadic type corresponds to the forward-chaining component. Thus, the proof search rules for CLF include the backward-chaining search rules of LF and LLF in addition to new rules for forward-chaining monadic types. In this case, we can rewrite the CLF type system as shown above. We distinguish between the synchronous types depending on whether they appear within a goal G or a program clause D. Their syntax is quite similar except that the type A within the type S is to be interpreted as type G for types S_G and D for types S_D. Their formal syntax is given below.

$$
\begin{array}{rcl}
S_G & ::= & S_G \otimes S_G \mid 1 \mid \exists x : A.S_G \mid !G \mid G \\
S_D & ::= & S_D \otimes S_D \mid 1 \mid \exists x : A.S_D \mid !D \mid D
\end{array}
$$

First, we shall give the backward-chaining rules for completeness before describing in detail the forward-chaining part. The backward-chaining rules for CLF are shown in Figure 3.1. These rules combine the proof search rules for LF and LLF.

The proof search rules for the forward-chaining part of CLF are given in Figure 3.2. The rules FWDCLAUSE and LIN-CWDCLAUSE are the forward-chaining steps that select a particular clause whose head is of the form $\{S'_D\}$.

The rule BACKCHAIN terminates the forward-chaining and the proof search moves to backward-chaining mode. The exact criteria for making this non-deterministic

Goals:

$$\frac{}{\Sigma, \Delta \vDash \top, \cdot}\ \mathsf{TOP}$$

$$\frac{\Sigma, \mathsf{c} : D, \Delta \vDash D \gg P, \theta}{\Sigma, \mathsf{c} : D, \Delta \vDash P, \theta}\ \mathsf{CLAUSE} \qquad \frac{\Sigma, \Delta \vDash D \gg P, \theta}{\Sigma, \Delta, \mathsf{c} \hat{:} D \vDash P, \theta}\ \mathsf{LIN\text{-}CLAUSE}$$

$$\frac{\mathsf{c}\ \mathsf{new} \quad \Sigma, \mathsf{c} : D, \Delta \vDash G, \theta}{\Sigma, \Delta \vDash D \to G, \theta}\ \mathsf{IMP} \qquad \frac{\mathsf{c}\ \mathsf{new} \quad \Sigma, \Delta, \mathsf{c} \hat{:} D \vDash G, \theta}{\Sigma, \Delta \vDash D \multimap G, \theta}\ \mathsf{LIN\text{-}IMP}$$

$$\frac{\mathsf{c}\ \mathsf{new} \quad \Sigma, \mathsf{c} : A, \Delta \vDash [\mathsf{c}/x]G, \theta}{\Sigma, \Delta \vDash \Pi x : A.G, \theta}\ \mathsf{ALL} \qquad \frac{\Sigma, \Delta \vDash G_1, \theta_1 \quad \Sigma, \Delta \vdash G_2, \theta_2}{\Sigma, \Delta \vDash G_1 \& G_2, \theta_1 \cup \theta_2}\ \mathsf{AND_G}$$

Clauses:

$$\frac{\theta \in \mathsf{unify}(P, Q)}{\Sigma, \cdot \vDash Q \gg P, \theta}\ \mathsf{ATOM} \qquad \frac{\Sigma, \Delta \vDash [X/x]D \gg P, \theta}{\Sigma, \Delta \vDash \Pi x : A.D \gg P, \theta}\ \mathsf{SOME}$$

$$\frac{\Sigma, \Delta \vDash D \gg P, \theta \quad \Sigma, \cdot \vDash \theta(G), \theta'}{\Sigma, \Delta \vDash G \to D \gg P, \theta'(\theta)}\ \mathsf{SUBGOAL}$$

$$\frac{\Sigma, \Delta_1 \vDash D \gg P, \theta \quad \Sigma, \Delta_2 \vDash \theta(G), \theta'}{\Sigma, \Delta_1, \Delta_2 \vDash G \multimap D \gg P, \theta'(\theta)}\ \mathsf{LIN\text{-}SUBGOAL}$$

$$\frac{\Sigma, \Delta \vDash D_1 \gg P, \theta}{\Sigma, \Delta \vDash D_1 \& D_2 \gg P, \theta}\ \mathsf{AND_{L_1}} \qquad \frac{\Sigma, \Delta \vDash D_2 \gg P, \theta}{\Sigma, \Delta \vDash D_1 \& D_2 \gg P, \theta}\ \mathsf{AND_{L_2}}$$

Figure 3.1: Backward-chaining proof search in CLF

choice depend on the system. Our criteria are slightly different from that proposed by the designers of CLF. We shall discuss these issues in the following subsection.

The function $\mathsf{split}(\cdot)$ used in the rule FWDATOM simply disaggregates S'_D and adds it to Σ and Δ. The function definition of $\mathsf{split}(\cdot)$ is given in Figure 3.3.

The reader is encouraged to refer to López, *et al.* [45] for more details.

3.2.2 Termination of forward-chaining

The intended semantics of forward-chaining mode and in particular the decision to terminate forward-chaining and return to goal-directed backward-chaining are

Goals:

$$\frac{\Sigma, c : D, \Delta \vDash D > \{S_G\}, \theta}{\Sigma, c : D, \Delta \vDash \{S_G\}, \theta}\ \textsf{FWDCLAUSE} \qquad \frac{\Sigma, \Delta \vDash D > \{S_G\}, \theta}{\Sigma, c\hat{:}D, \Delta \vDash \{S_G\}, \theta}\ \textsf{LIN-FWDCLAUSE}$$

$$\frac{\Sigma, \Delta \vDash S_G, \theta}{\Sigma, \Delta \vDash \{S_G\}, \theta}\ \textsf{BACKCHAIN}$$

$$\frac{\Sigma, \Delta_1 \vDash S_{G_1}, \theta_1 \quad \Sigma, \Delta_2 \vDash S_{G_2}, \theta_2}{\Sigma, \Delta_1, \Delta_2 \vDash S_{G_1} \otimes S_{G_2}, \theta_1 \cup \theta_2}\ \textsf{MULTPROD} \qquad \frac{}{\Sigma, \cdot \vDash 1, \cdot}\ \textsf{ONE}$$

$$\frac{\Sigma, \cdot \vDash G, \theta}{\Sigma, \cdot \vDash !G, \theta}\ \textsf{EXP} \qquad \frac{\Sigma, \Delta \vDash [X/x]S_G, \theta}{\Sigma, \Delta \vDash \exists x : A.S_G, \theta}\ \textsf{FWDSOME}$$

Clause:

$$\frac{\Sigma, \Sigma', \Delta, \Delta' \vDash \{S_G\}, \theta}{\Sigma, \Delta \vDash \{S'_D\} > \{S_G\}, \theta}\ \textsf{FWDATOM} \qquad \frac{\Sigma, \Delta \vDash [X/x]D > \{S_G\}, \theta}{\Sigma, \Delta \vDash \Pi x : A.D > \{S_G\}, \theta}\ \textsf{FWDSOME}$$

(where $(\Sigma'; \Delta') = \mathsf{split}(S'_D)$)

$$\frac{\Sigma, \Delta \vDash D > \{S_G\}, \theta \quad \Sigma, \cdot \vDash \theta(G), \theta'}{\Sigma, \Delta \vDash G \rightarrow D > \{S_G\}, \theta'(\theta)}\ \textsf{FWDSUBGOAL}$$

$$\frac{\Sigma, \Delta_1 \vDash D > \{S_G\}, \theta \quad \Sigma, \Delta_2 \vDash \theta(G), \theta'}{\Sigma, \Delta_1, \Delta_2 \vDash G \multimap D > \{S_G\}, \theta'(\theta)}\ \textsf{LIN-FWDSUBGOAL}$$

$$\frac{\Sigma, \Delta \vDash D_1 > \{S_G\}, \theta}{\Sigma, \Delta \vDash D_1 \& D_2 > \{S_G\}, \theta}\ \textsf{FWDAND}_{\textsf{L}_1} \qquad \frac{\Sigma, \Delta \vDash D_2 > \{S_G\}, \theta}{\Sigma, \Delta \vDash D_1 \& D_2 > \{S_G\}, \theta}\ \textsf{FWDAND}_{\textsf{L}_2}$$

Figure 3.2: Forward-chaining proof search in CLF

decisions that are made based on the purpose of the system.

The designers of CLF have proposed two main criteria for termination of forward-chaining. The simplest strategy that is proposed is *quiescence*: the forward-chaining mode ends when *no* forward step is available to be taken. However, in many cases it is more useful to consider an alternative strategy named *saturation*: a state in which forward steps may be available, but they do not cause any change in the state of the contexts. In other words, the saturation criterion disallows forward reasoning steps that either have no effect on the set of available linear and intuitionistic hypotheses,

$$
\begin{array}{rcl}
\mathsf{split}(1) & = & (\cdot;\cdot) \\
\mathsf{split}(!D) & = & (D;\cdot) \\
\mathsf{split}(D) & = & (\cdot;D) \\
\\
\mathsf{split}(S_{D_1} \otimes S_{D_2}) & = & (\Sigma_1, \Sigma_2; \Delta_1, \Delta_2) \\
 & & \mathsf{where}\ (\Sigma_1, \Delta_1) = \mathsf{split}(S_{D_1}) \\
 & & \mathsf{and}\ (\Sigma_2, \Delta_2) = \mathsf{split}(S_{D_2}) \\
\mathsf{split}(\exists x : A.S_D) & = & \mathsf{split}([\mathsf{c}/x]S_D) \\
 & & (\text{where } \mathsf{c} \text{ is a new parameter}))
\end{array}
$$

Figure 3.3: Definition of $\mathsf{split}(\cdot)$

or simply reintroduce intuitionistic hypotheses already known. These strategies are described in more detail in López, *et al.* [45].

For the purpose of representing reductions between NP-complete problems, we shall modify the saturation criteria slightly. Our modified criteria does not allow forward reasoning steps that introduce linear hypotheses already present in the linear context. Since we use linear hypotheses to store problem instances and other additional data, this modified condition greatly simplifies representation of many reductions.

3.3 Representing algorithms in CLF

In this section, we shall give some examples of reductions between some NP-complete problems in CLF. The examples will utilize both the backward and forward-chaining operational semantics of CLF. Appendix A gives complete definitions of NP-complete problems used in this section.

3.3.1 Reduction from SAT to 3-SAT

We have mentioned earlier that we would use intuitionistic and linear contexts to store problem instances. Thus, an instance of SAT is a context consisting of assumptions mentioning the literals appearing in each clause. Thus, we need the following CLF declarations for describing an instance of SAT and 3-SAT.

$$\begin{array}{rcl} \mathsf{variable} & : & \mathsf{type} \\ \mathsf{literal} & : & \mathsf{type} \\ \mathsf{clause} & : & \mathsf{type} \\ \mathsf{term} & : & \mathsf{type} \\ \mathsf{cnfreduction} & : & \mathsf{type} \\ \\ \mathsf{pos} & : & \mathsf{variable} \rightarrow \mathsf{literal} \\ \mathsf{neg} & : & \mathsf{variable} \rightarrow \mathsf{literal} \\ \\ \mathsf{3disjunct} & : & \mathsf{literal} \rightarrow \mathsf{literal} \rightarrow \mathsf{literal} \rightarrow \mathsf{type} \\ \mathsf{ndisjunct} & : & \mathsf{clause} \rightarrow \mathsf{literal} \rightarrow \mathsf{type} \end{array}$$

Now, the boolean formula $(u_1 \vee \bar{u}_2 \vee u_3 \vee \bar{u}_4) \wedge (u_2 \vee u_3 \vee u_4)$ is represented as the following context: $u_1 : \mathsf{variable}, u_2 : \mathsf{variable}, u_3 : \mathsf{variable}, u_4 : \mathsf{variable}, c_1 : \mathsf{clause}, c_2 : \mathsf{clause}; n_1 : \mathsf{ndisjunct}\ c_1\ (\mathsf{pos}\ u_1), n_2 : \mathsf{ndisjunct}\ c_1\ (\mathsf{neg}\ u_2), n_3 : \mathsf{ndisjunct}\ c_1\ (\mathsf{pos}\ u_3), n_4 : \mathsf{ndisjunct}\ c_1\ (\mathsf{neg}\ u_4), n_5 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_2), n_6 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_3), n_7 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_4)$.

We have chosen to represent the ndisjunct assumptions in linear context as it permits an easier solution to the reduction algorithm.

Similarly, representation of the boolean formula $(u_1 \vee u_2 \vee \bar{u}_3) \wedge (\bar{u}_1 \vee u_3 \vee u_4)$ as an instance of 3-SAT is given by: $u_1 : \mathsf{variable}, u_2 : \mathsf{variable}, u_3 : \mathsf{variable}, u_4 : \mathsf{variable}; o_1 : \mathsf{3disjunct}\ (\mathsf{pos}\ u_1)\ (\mathsf{pos}\ u_2)\ (\mathsf{neg}\ u_3), o_2 : \mathsf{3disjunct}\ (\mathsf{neg}\ u_1)\ (\mathsf{pos}\ u_3)\ (\mathsf{neg}\ u_4)$

The reduction algorithm that converts an instance of SAT to that of 3-SAT runs in the initial context corresponds to a valid SAT instance and on termination the final context is the representation of a 3-SAT instance. This algorithm is given below.

The rules convert, term1, term2, and term3 are forward-chaining rules and the rule terminate is a backward-chaining rule. The algorithm runs in two forward-chaining phases. In the first phase, convert is applied as many times as possible until

$$
\begin{array}{rcl}
\mathsf{convert} & : & \mathsf{ndisjunct}\ C\ L_1 \multimap \mathsf{ndisjunct}\ C\ L_2 \multimap \mathsf{ndisjunct}\ C\ L_3 \multimap \mathsf{ndisjunct}\ C\ L_4 \\
& & \multimap \{\exists u : \mathsf{variable}.\mathsf{3disjunct}\ L_1\ L_2\ (\mathsf{pos}\ u) \otimes \mathsf{ndisjunct}\ C\ (\mathsf{neg}\ u) \otimes \\
& & \mathsf{ndisjunct}\ C\ L_3 \otimes \mathsf{ndisjunct}\ C\ L_4\} \\
\mathsf{terminate} & : & (\mathsf{term} \to \{\top\}) \multimap \mathsf{cnfreduction} \\
\mathsf{term1} & : & \mathsf{term} \to \mathsf{ndisjunct}\ C\ L_1 \multimap \mathsf{ndisjunct}\ C\ L_2 \multimap \mathsf{ndisjunct}\ C\ L_3 \\
& & \multimap \{\mathsf{3disjunct}\ L_1\ L_2\ L_3\} \\
\mathsf{term2} & : & \mathsf{term} \to \mathsf{ndisjunct}\ C\ L_1 \multimap \mathsf{ndisjunct}\ C\ L_2 \\
& & \multimap \{\exists u : \mathsf{variable3disjunct}\ L_1\ L_2\ (\mathsf{pos}\ u) \otimes \mathsf{3disjunct}\ L_1\ L_2\ (\mathsf{neg}\ u)\} \\
\mathsf{term3} & : & \mathsf{term} \to \mathsf{ndisjunct}\ C\ L_1 \multimap \{\exists u_1 : \mathsf{variable}\ u_2 : \mathsf{variable} \\
& & \ \mathsf{3disjunct}\ L_1\ (\mathsf{pos}\ u_1)\ (\mathsf{pos}\ u_2) \otimes \\
& & \ \mathsf{3disjunct}\ L_1\ (\mathsf{pos}\ u_1)\ (\mathsf{neg}\ u_2) \otimes \\
& & \ \mathsf{3disjunct}\ L_1\ (\mathsf{neg}\ u_1)\ (\mathsf{pos}\ u_2) \otimes \\
& & \ \mathsf{3disjunct}\ L_1\ (\mathsf{neg}\ u_1)\ (\mathsf{neg}\ u_2)\}
\end{array}
$$

saturation. At this point, every clause from the original instance of SAT has no more than three literals left; the rest of the clause is converted into its 3-SAT form. The terminate step begins the final forward-chaining phase in which only term1, term2 and term3 are applied.

Thus, the execution trace of reduction of the boolean formula $(u_1 \vee u_2 \vee u_3 \vee \bar{u}_4) \wedge (u_2 \vee u_3 \vee u_5)$ to its 3-SAT instance is represented by the expression

$$
\begin{array}{l}
\mathsf{let}\ \{[u, o_1 \otimes n' \otimes n_3 \otimes n_4]\} = \mathsf{convert}\hat{\ } n_1 \hat{\ } n_2 \hat{\ } n_3 \hat{\ } n_4\ \mathsf{in} \\
\mathsf{terminate}\hat{\ }(\lambda t : \mathsf{term}.\mathsf{let}\ \{o_2\} = \mathsf{term1}\hat{\ } n_3 \hat{\ } n_4 \hat{\ } n'\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\quad \mathsf{let}\ \{o_3\} = \mathsf{term1}\hat{\ } n_5 \hat{\ } n_6 \hat{\ } n_7\ \mathsf{in}\ \langle\rangle)
\end{array}
$$

In this case, the initial linear context is given by $n_1 : \mathsf{ndisjunct}\ c_1\ (\mathsf{pos}\ u_1), n_2 : \mathsf{ndisjunct}\ c_1\ (\mathsf{pos}\ u_2), n_3 : \mathsf{ndisjunct}\ c_1\ (\mathsf{pos}\ u_3), n_4 : \mathsf{ndisjunct}\ c_1\ (\mathsf{pos}\ u_4), n_5 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_2), n_6 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_3), n_7 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_5)$. The

$$
\begin{array}{rcl}
\mathsf{boolean} & : & \mathsf{type} \\
\mathsf{true} & : & \mathsf{boolean} \\
\mathsf{false} & : & \mathsf{boolean} \\
\mathsf{assign} & : & \mathsf{variable} \to \mathsf{boolean} \to \mathsf{type} \\
\\
\mathsf{3cnf1} & : & \mathsf{assign}\ V\ \mathsf{true} \to \mathsf{3disjunct}\ (\mathsf{pos}\ V)\ L_2\ L_3 \multimap \{1\} \\
\mathsf{3cnf2} & : & \mathsf{assign}\ V\ \mathsf{true} \to \mathsf{3disjunct}\ L_1\ (\mathsf{pos}\ V)\ L_3 \multimap \{1\} \\
\mathsf{3cnf3} & : & \mathsf{assign}\ V\ \mathsf{true} \to \mathsf{3disjunct}\ L_1\ L_2\ (\mathsf{pos}\ V)\ \multimap \{1\} \\
\mathsf{3cnf4} & : & \mathsf{assign}\ V\ \mathsf{false} \to \mathsf{3disjunct}\ (\mathsf{neg}\ V)\ L_2\ L_3 \multimap \{1\} \\
\mathsf{3cnf5} & : & \mathsf{assign}\ V\ \mathsf{false} \to \mathsf{3disjunct}\ L_1\ (\mathsf{neg}\ V)\ L_3 \multimap \{1\} \\
\mathsf{3cnf6} & : & \mathsf{assign}\ V\ \mathsf{false} \to \mathsf{3disjunct}\ L_1\ L_2\ (\mathsf{neg}\ V)\ \multimap \{1\} \\
\\
\mathsf{sat} & : & \mathsf{clause} \to \mathsf{type} \\
\\
\mathsf{cnf1} & : & \mathsf{sat}\ C \to \mathsf{ndisjunct}\ C\ (\mathsf{pos}\ V) \multimap \{1\} \\
\mathsf{cnf2} & : & \mathsf{sat}\ C \to \mathsf{ndisjunct}\ C\ (\mathsf{neg}\ V) \multimap \{1\} \\
\\
\mathsf{cnf3} & : & \mathsf{assign}\ V\ \mathsf{true} \to \mathsf{ndisjunct}\ C\ (\mathsf{pos}\ V) \multimap \{!\mathsf{sat}\ C\} \\
\mathsf{cnf4} & : & \mathsf{assign}\ V\ \mathsf{false} \to \mathsf{ndisjunct}\ C\ (\mathsf{neg}\ V) \multimap \{!\mathsf{sat}\ C\}
\end{array}
$$

Figure 3.4: Representing proofs of *Yes* instances in CLF

final linear context is given by

$$
\begin{array}{l}
o_1 : \mathsf{3disjunct}\ (\mathsf{pos}\ u)\ (\mathsf{pos}\ u_1)\ (\mathsf{pos}\ u_2), \\
o_2 : \mathsf{3disjunct}\ (\mathsf{neg}\ u)\ (\mathsf{pos}\ u_3)\ (\mathsf{pos}\ u_4), \\
o_3 : \mathsf{3disjunct}\ (\mathsf{pos}\ u_2)\ (\mathsf{pos}\ u_3)\ (\mathsf{pos}\ u_5)
\end{array}
$$

Finally, the rules for representing proofs that certain instances of SAT and 3-SAT are *Yes* instances are shown in Figure 3.4. In this case, the assignment of truth values to boolean variables is done by assumptions $a : \mathsf{assign}\ u\ \mathsf{true}$ or $\mathsf{assign}\ u\ \mathsf{false}$. In this case, the initial context is contains an instance of SAT or 3-SAT and the final context is the empty linear context.

$$
\begin{array}{rcl}
\mathsf{vertex} & : & \mathsf{type} \\
\mathsf{edge} & : & \mathsf{vertex} \rightarrow \mathsf{vertex} \rightarrow \mathsf{type} \\
\mathsf{eq} & : & \mathsf{variable} \rightarrow \mathsf{variable} \rightarrow \mathsf{type} \\
\mathsf{neq} & : & \mathsf{variable} \rightarrow \mathsf{variable} \rightarrow \mathsf{type}
\end{array}
$$

Figure 3.5: Representing instances of graphs

3.3.2 Reduction from 3-SAT to CHROMATIC

For this reduction, an instance of 3-SAT can be represented as described in the previous section. Graphs can be represented in linear context by introducing a new type constructors vertex and edge to represent vertices and edges. We also have eq and neq type constructors as this feature is not provided by the underlying system. These definitions are given in Figure 3.5.

The reduction algorithm that converts an instance of 3-SAT to that of CHROMATIC is given in Figure 3.6. The initial context for this algorithm consists of a series of assumptions of the form 3disjunct L_1 L_2 L_3 corresponding to every clause in conjunctive normal form. The final linear context consists of assumptions of the form edge V_1 V_2 which are the edges in the final graph.

3.3.3 Reduction from SAT to CLIQUE

For this reduction, given the boolean formula F in conjunctive normal form as an instance of SAT, the graph corresponding to an instance of CLIQUE is described by the CLF algorithm in Figure 3.7.

Figure 3.8 gives rules to represent *Yes* instances of CLIQUE. In this case, the assumptions clique and nclique are used to keep track of vertices in the clique. In this case, the initial linear context consists of edges of the graph and the final context is the empty linear context.

$$
\begin{array}{rcl}
\mathsf{var} & : & \mathsf{variable} \to \mathsf{type} \\
\\
\mathsf{ndv} & : & \mathsf{variable} \to \mathsf{vertex} \\
\mathsf{ndv}' & : & \mathsf{variable} \to \mathsf{vertex} \\
\mathsf{ndx} & : & \mathsf{variable} \to \mathsf{vertex} \\
\mathsf{edge}' & : & \mathsf{vertex} \to \mathsf{vertex} \to \mathsf{type} \\
\mathsf{cl} & : & \mathsf{literal} \to \mathsf{literal} \to \mathsf{literal} \to \mathsf{vertex} \\
\\
\mathsf{vertex1} & : & \mathsf{var}\ U \to \{\mathsf{edge}\ (\mathsf{ndv}\ U)\ (\mathsf{ndv}'\ U)\} \\
\mathsf{vertex2}' & : & \mathsf{var}\ U_1 \to \mathsf{var}\ U_2 \to \mathsf{neq}\ U_1\ U_2 \to \{!\mathsf{edge}'\ (\mathsf{ndx}\ U_1)\ (\mathsf{ndx}\ U_2) \\
 & & \qquad \otimes !\mathsf{edge}'\ (\mathsf{ndv}'\ U_1)\ (\mathsf{ndx}\ U_2) \\
 & & \qquad \otimes !\mathsf{edge}'\ (\mathsf{ndv}\ U_1)\ (\mathsf{ndx}\ U_2)\} \\
\mathsf{vertex2} & : & \mathsf{edge}'\ V_1\ V_2 \to \mathsf{edge}'\ V_2\ V_1 \to \{\mathsf{edge}\ V_1\ V_2\} \\
\\
\mathsf{clause1} & : & \mathsf{var}\ U \to \mathsf{3disjunct}\ L_1\ L_2\ L_3 \\
 & & \multimap \{\mathsf{edge}\ (\mathsf{ndv}\ U)\ (\mathsf{cl}\ L_1\ L_2\ L_3) \otimes \mathsf{edge}\ (\mathsf{ndv}'\ U)\ (\mathsf{cl}\ L_1\ L_2\ L_3) \\
 & & \quad \otimes !\mathsf{3disjunct}\ L_1\ L_2\ L_3\} \\
\mathsf{clause2.1} & : & \mathsf{var}\ U \to \mathsf{3disjunct}\ (\mathsf{pos}\ U)\ L_2\ L_3 \\
 & & \qquad \to \mathsf{edge}\ (\mathsf{ndv}\ U)\ (\mathsf{cl}\ (\mathsf{pos}\ U)\ L_2\ L_3) \multimap \{1\} \\
\mathsf{clause2.2} & : & \mathsf{var}\ U \to \mathsf{3disjunct}\ L_1\ (\mathsf{pos}\ U)\ L_3 \\
 & & \qquad \to \mathsf{edge}\ (\mathsf{ndv}\ U)\ (\mathsf{cl}\ L_1 (\mathsf{pos}\ U)\ L_3) \multimap \{1\} \\
\mathsf{clause2.3} & : & \mathsf{var}\ U \to \mathsf{3disjunct}\ L_1\ L_2\ (\mathsf{pos}\ U) \\
 & & \qquad \to \mathsf{edge}\ (\mathsf{ndv}\ U)\ (\mathsf{cl}\ L_1\ L_2\ (\mathsf{pos}\ U)) \multimap \{1\} \\
\mathsf{clause2.4} & : & \mathsf{var}\ U \to \mathsf{3disjunct}\ (\mathsf{neg}\ U)\ L_2\ L_3 \\
 & & \qquad \to \mathsf{edge}\ (\mathsf{ndv}'\ U)\ (\mathsf{cl}\ (\mathsf{neg}\ U)\ L_2\ L_3) \multimap \{1\} \\
\mathsf{clause2.5} & : & \mathsf{var}\ U \to \mathsf{3disjunct}\ L_1\ (\mathsf{neg}\ U)\ L_3 \\
 & & \qquad \to \mathsf{edge}\ (\mathsf{ndv}'\ U)\ (\mathsf{cl}\ L_1 (\mathsf{neg}\ U)\ L_3) \multimap \{1\} \\
\mathsf{clause2.6} & : & \mathsf{var}\ U \to \mathsf{3disjunct}\ L_1\ L_2\ (\mathsf{neg}\ U) \\
 & & \qquad \to \mathsf{edge}\ (\mathsf{ndv}'\ U)\ (\mathsf{cl}\ L_1\ L_2\ (\mathsf{neg}\ U)) \multimap \{1\}
\end{array}
$$

Figure 3.6: Reduction from 3-SAT to CHROMATIC

$$
\begin{array}{rcl}
\mathsf{node} & : & \mathsf{clause} \to \mathsf{literal} \to \mathsf{type} \\
\mathsf{nd} & : & \mathsf{clause} \to \mathsf{literal} \to \mathsf{vertex} \\
\\
\mathsf{nodes} & : & \mathsf{ndisjunct}\ C\ L \multimap \{!\mathsf{node}\ C\ L\} \\
\mathsf{edges} & : & \mathsf{node}\ C_1\ L_1 \to \mathsf{node}\ C_2\ L_2 \to \mathsf{neq}\ C_1\ C_2 \to \{\mathsf{edge}\ (\mathsf{nd}\ C_1\ L_1)\ (\mathsf{nd}\ C_2\ L_2)\} \\
\mathsf{edge}'_1 & : & \mathsf{edge}\ (\mathsf{nd}\ C_1\ (\mathsf{pos}\ U))\ (\mathsf{nd}\ C_2\ (\mathsf{neg}\ U)) \multimap \{\top\} \\
\mathsf{edge}'_2 & : & \mathsf{edge}\ (\mathsf{nd}\ C_1\ (\mathsf{neg}\ U))\ (\mathsf{nd}\ C_2\ (\mathsf{pos}\ U)) \multimap \{\top\}
\end{array}
$$

Figure 3.7: Reduction from SAT to CLIQUE

$$
\begin{array}{rcl}
\mathsf{clique} & : & \mathsf{vertex} \rightarrow \mathsf{type} \\
\mathsf{nclique} & : & \mathsf{vertex} \rightarrow \mathsf{type} \\
\\
\mathsf{edge1} & : & \mathsf{clique}\ V_1 \rightarrow \mathsf{clique}\ V_2 \rightarrow \{\mathsf{edge}'\ V_1\ V_2\} \\
\mathsf{edge2} & : & \mathsf{edge}'\ V_1\ V_2 \multimap \mathsf{edge}\ V_1\ V_2 \multimap \{1\} \\
\mathsf{edge3} & : & \mathsf{edge}\ V_1\ V_2 \multimap \mathsf{nclique}\ V_1 \rightarrow \{1\} \\
\mathsf{edge4} & : & \mathsf{edge}\ V_1\ V_2 \multimap \mathsf{nclique}\ V_2 \rightarrow \{1\}
\end{array}
$$

Figure 3.8: Representing *Yes* instances of CLIQUE

Chapter 4

Complexity analysis of backward-chaining logic programs

In this chapter, we shall present in detail a method for analyzing computational complexity of recursive functions written as backward-chaining logic programs. In this framework, we write recursive functions as relations with well-defined mode-behavior. The results in this chapter will be first presented for the Horn fragment and later extended to hereditary Harrop formulas and the Horn fragment with linear connectives. The term algebra that we will assume when we present the results and related examples will be simply-typed λ-calculus. However, we shall also show that these results are general enough to be applied to the dependently typed term algebra of LF and linear LF with some modifications.

4.1 Background and related work

The task of deciding if a function over binary numbers is computable in polynomial time is well-understood and based on a series of results that date back to

Cobham [15]. Schwichtenberg [68] gives a detailed account of research results on classifying recursive functions into complexity classes.

However, deciding if a general recursive function over arbitrary, possibly higher-order data-types is computable in polynomial time remains difficult, and requires in general a reformulation into a previously established formalism, e.g. as a function in bounded recursion on notation [15], a function in Bellantoni and Cook's algebra [4] or Leivant's algebra [41, 42], or a function typeable in Bellantoni et al. [5] or Hofmann's type systems [34, 35]. In this chapter, we reconcile the simplicity of characterizing polynomial-time functions over *binary numbers* with the expressiveness of general recursive functions over arbitrary domains.

As we have mentioned below, our underlying model is backward-chaining logic programming, where functions are declared as relations. This idea is fundamentally different from Ganzinger and McAllester [24] and Givan and McAllester [28] who have given various criteria for identifying polynomial time predicates for forward-chaining logic programming. Our measure of complexity is captured as the size of the execution derivation, a logical deduction, in terms of the size of the input arguments. We consider only the class of logic programs that implement functions, and we show that our notion of complexity is compatible with the usual one. Furthermore, we give a *sufficient* criterion that decides if a logic program runs in polynomial time.

It is sufficient to require that the sum of the size of arguments passed to the recursive calls not exceed the size of the input arguments of the function. In addition, all calls to auxiliary (non-recursive) functions that take recursively computed arguments as inputs must be shown to be non-size increasing. Aehlig, *et al.* [1] and Hofmann [36] have also used the latter condition to extend Hofmann's polynomial-time type system to include a larger class of functions. If the criterion is satisfied, we show that the logic programming engine will terminate in a number of steps that is bounded by a polynomial in the size of the input.

4.2 Functions as logic programs

We are interested in studying general recursive functions and classifying their running time into complexity classes using syntactic criteria. We think of a recursive function $(y_1, \ldots, y_n) = f(x_1, \ldots, x_m)$ as a predicate $P_f(x_1, \ldots, x_m; y_1, \ldots, y_n)$ that relates input arguments x_i with output arguments y_i. These relations fall into a subclass of well-moded logic programs that compute ground output terms from ground input terms. A *ground term* is a term not containing any free logic variables. In logic programming the underlying model of computation is proof search; and thus a complete computation trace corresponds to a closed proof derivation, which determines ground terms in all output positions. Such logic programs are considered to have a well-defined *mode* behavior. The reader may refer to Section 2.1.2 and Rohwedder and Pfenning [64] for more information on algorithms for identifying *mode correct* logic programs.

4.2.1 Term Algebra

We shall restrict ourselves to simply typed λ-calculus during the presentation of the complexity analysis results. In Section 4.5, we shall extend these results to the dependently typed λ-calculus of LF.

In Section 4.4 we shall show how to extend the results to other term algebras like the dependently typed λ-calculus LF.

$$
\begin{array}{llcl}
\textit{Types} & A, B & ::= & a \mid A \to B \\
\textit{Canonical Terms} & M, N & ::= & \lambda x : A.N \mid R \\
\textit{Atomic Terms} & R & ::= & c \mid x \mid R\ N
\end{array}
$$

where a and c are type and object level constants declared a priori. For studying run-time complexity, it is convenient to consider only canonical terms, i.e. terms without β-redexes. However, many interesting examples with higher-order terms have non-canonical terms. In Section 4.4, we extend the results to non-canonical terms.

4.2.2 Logic programming as model of computation

Logic programming can serve as a model of computation where traces are captured by proof derivations. For simplicity we consider only the Horn fragment in this section, but extend the results to the fragment of hereditary Harrop formulas in Section 4.4. Predicates are given by

$$P(M_1, \ldots, M_m; N_1, \ldots, N_n)$$

where inputs M_i are separated from outputs N_i by a semicolon. The formulation of Horn-logic in terms of goals G and definite clauses D is standard.

$$
\begin{array}{llcl}
\textit{Goals} & G & ::= & \top \mid P \\
\textit{Clauses} & D & ::= & G \supset D \mid \exists x : A.D \mid P \\
\textit{Programs} & \mathcal{F} & ::= & \bullet \mid \mathcal{F}, D
\end{array}
$$

A logic program $\mathcal{F}$ is simply a collection of clauses. Often we find it convenient to reverse the direction of $G \supset D$ and use $D \subset G$ instead. $\supset$ is right-associative. In addition, we always omit the leading $\bullet$ from programs.

$$\frac{}{\mathcal{F} \vDash \top}\ \mathsf{g\_True} \qquad \frac{D \in \mathcal{F} \quad \mathcal{F} \vDash D \gg P}{\mathcal{F} \vDash P}\ \mathsf{g\_Atom}$$

$$\frac{Q \doteq P}{\mathcal{F} \vDash Q \gg P}\ \mathsf{c\_Atom} \qquad \frac{\mathcal{F} \vDash [M/x]D \gg P}{\mathcal{F} \vDash \exists x : A.D \gg P}\ \mathsf{c\_Exists} \qquad \frac{\mathcal{F} \vDash D \gg P \quad \mathcal{F} \vDash G}{\mathcal{F} \vDash G \supset D \gg P}\ \mathsf{c\_Imp}$$

Figure 4.1: Proof search semantics for the Horn fragment

Definition 4.2.1 (Predicate symbol, head of a clause)**.** For a clause D or a goal G, we define predicate symbol of D or G and head of a clause D as given below:

$$\begin{array}{rcl}
\mathsf{symbol}(P(\cdot;\cdot)) & = & P \\
\mathsf{symbol}(\forall x : A.D) & = & \mathsf{symbol}(D) \\
\mathsf{symbol}(G \supset D) & = & \mathsf{symbol}(D)
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{head}(P) & = & P \\
\mathsf{head}(\forall x : A.D) & = & \mathsf{head}(D) \\
\mathsf{head}(G \supset D) & = & \mathsf{head}(D)
\end{array}$$

4.2.3 Function computation through proof search

The proof search semantics of Horn logic is given in Figure 4.1. Given a program $\mathcal{F}$ and a goal G with ground terms in its input positions, the interpreter constructs a derivation of the judgment $\mathcal{F} \vDash G$. In the rule g_Atom an appropriate clause D corresponding to the goal G is selected. It is possible to construct a derivation for the judgment $\mathcal{F} \vDash D \gg P$ only if head of D can be made equal to P. For the sake of our analysis, we assume that an oracle predicts the correct instantiations of the universally quantified formulas (c_Exists). The rule c_Atom unifies the head of a clause Q with the head of the goal P.

In an actual implementation, however, one would postpone non-deterministic choice by employing logic variables that are eventually instantiated by unification, as all logic programs considered here are mode-correct. The proof search rules for LF (Figure 2.1), LLF (Figure 2.9) and CLF (Figure 3.1 and 3.2) give describe the proof search in the latter style.

This logic program implements the Fibonacci function on natural numbers represented in unary:

$$\begin{aligned}\mathcal{F} \;=\;& +(\mathsf{z}, Y; Y), +(X, Y; Z) \supset +(\mathsf{s}\ X, Y; \mathsf{s}\ Z),\\ & \mathsf{fib}(\mathsf{z}; \mathsf{s}\ \mathsf{z}), \mathsf{fib}(\mathsf{s}\ \mathsf{z}; \mathsf{s}\ \mathsf{z}),\\ & +(X, Y; Z) \supset \mathsf{fib}(N; X) \supset \mathsf{fib}(\mathsf{s}\ N; Y) \supset \mathsf{fib}(\mathsf{s}\ (\mathsf{s}\ N); Z)\end{aligned}$$

where the constants z, s, and fib are appropriately defined, and all uppercase variables are of type nat and implicitly universally quantified at the beginning of the respective clause.

For a logic program $\mathcal{F}$, we denote a proof search derivation for a goal G by $\mathcal{D} :: \mathcal{F} \vDash G$ and measure the size of this derivation as the number of inference rules in the derivation. In Section 4.2.5, we show that every rule can be implemented on a random access machine (RAM) in a constant number of steps.

Definition 4.2.2 (Size of proof search derivation)**.** Given a logic program $\mathcal{F}$ and a derivation $\mathcal{D} :: \mathcal{F} \vDash G$, we define the size of $\mathcal{D}$, $\mathsf{sz}(\mathcal{D})$ as the number of rules in $\mathcal{D}$.

4.2.4 Size of terms, goals and clauses

The relevant size functions for the terms from simply typed λ-calculus are defined in Figure 4.2. # counts the number of variables and constants in a term. The size of a goal G or a clause D is defined using $\mathsf{sz_i}(\cdot)$ and $\mathsf{sz_o}(\cdot)$ depending on whether we wish to compute the size of *input* or *output* arguments. $\mathsf{sz_i}(G)$ computes the sum of #-sizes of all the *input* arguments in the goal G and $\mathsf{sz_i}(D)$ computes the sum of #-sizes of all the *input* arguments in predicate P in the clause D. For logic variables, $\#(X)$ is itself an integer variable.

We would like to note that the choice of size function is essentially driven by our method of representation of the terms within the proof search engine.

$$
\begin{array}{rcl}
\#(X) & & \textsf{variable} \\
\#(x) = \#(c) & = & 1 \\
\#(R\ N) & = & \#(R) + \#(N) \\
\#(\lambda x.N) & = & \#(N)
\end{array}
$$

$$
\begin{array}{rclrcl}
\mathsf{sz_u}(\top) & = & 0 & \mathsf{sz_i}(P(M_1, \ldots, M_m; \cdot)) & = & \sum_{i=1}^{m} \#(M_i) \\
\mathsf{sz_u}(G \supset D) & = & \mathsf{sz_u}(D) & \mathsf{sz_o}(P(\cdot; N_1, \ldots, N_n)) & = & \sum_{i=1}^{n} \#(N_i) \\
\mathsf{sz_u}(\forall x : A.D) & = & \mathsf{sz_u}(D) & & &
\end{array}
$$

Figure 4.2: Size function for goals G and clauses D ($\mathsf{u} = \mathsf{i}$ or $\mathsf{u} = \mathsf{o}$)

4.2.5 Translation to a random access machine (RAM)

The following two conditions are sufficient to show that the proof search algorithm from Figure 4.1 can be implemented on a RAM in time proportional to the number of proof search rules in a proof search derivation. The mode-correct logic program

1. must be deterministic and non-backtracking,

2. and the time required to solve the individual unification problem is independent of input or output arguments.

The first condition is satisfied if the cases have non-overlapping patterns and all output positions of recursive calls contain only variables. The second is satisfied if all variables that occur in input arguments in the head of a clause are *linear* (i.e. variables occur exactly once) and form higher-order patterns in the sense of Miller [52]. Linearity guarantees that logical variables are only instantiated once and hence limit the complexity of unification by the size of the pattern. This may sound as a prohibitive restriction as clauses such as $P(x, x; x) \subset \top$ are disallowed. However, those clauses may be *linearized* by explicitly providing an equality predicate equal. The clause $P(x, x; x) \subset \top$ is now replaced by $P(x, y; x) \subset \mathsf{equal}(x, y;) \subset \top$. In practice, we can allow such non-linear clauses when we know that the terms that are bound to the non-linear variables are bounded in size. For examples, if those terms have no constructors and hence are of unit size, the complexity of unification is bounded even if there are duplicate logical variables. The examples that we will discuss later will make this point clearer.

Higher-order patterns are simply-typed β-normal λ-terms whose universally bound variables X are applied exclusively to a sequence of distinct bound variables. Qian [61] has also given a linear time and space unification algorithm for higher-order patterns. Under those two conditions the logic programming engine can be implemented on a RAM without increase in its asymptotic complexity.

Theorem 4.2.1. *Given a logic program $\mathcal{F}$ and a goal G satisfying the conditions given above. If there exists a derivation $\mathcal{D} :: \mathcal{F} \vDash G$, then*

1. *The goal G can be represented on a RAM in size proportional to $\mathsf{sz_i}(G)$.*

2. *The corresponding proof search can be implemented in time proportional to $\mathsf{sz}(\mathcal{D})$.*

Proof. The goal G can be represented on RAM by simply storing the ground terms in the input positions of the goal G. The total size of this input is bound by $\mathsf{sz_i}(G)$.

We shall now show that every rule in Figure 4.1 can be implemented in a constant number of steps, i.e. depending only on the size of the logic program and not its input arguments.

For the rules, $\mathsf{g\_True}$ and $\mathsf{c\_Imp}$, it is clear that the implementation can be done in constant number of steps. Implementing $\mathsf{g\_Atom}$ involves selecting the correct clause based on the inputs to the goal G. This selection is done by matching the inputs of the goal G with the input patterns in the clauses in $\mathcal{F}$. Since the program $\mathcal{F}$ is fixed, the maximal depth of the patterns is known and it is possible to design a hash function which maps every unique pattern to a hash value[1], thus providing a

[1] A simple implementation would assign a unique prime number to every type family. In this

constant time implementation for pattern matching.

During the implementation of c_Exists, we substitute the universally quantified variables by a logic variables which are unified with the ground terms in the rule c_Atom. Since the logic program is mode correct, all logic variables are guaranteed to be ground when the proof search completes. Unification is guaranteed to be decidable and depends only on the size of the program. Moreover, since the program is mode correct and no variable appears more than once in an input position, unification is simply a series of pattern matching operations and hence it runs in time polynomial in the size of the pattern (a constant). The number of such operations per inference rule is bounded by the total number of input positions which is a constant depending only on the logic program $\mathcal{F}$. $\square$

4.3 Conditions for polynomial-time functions

In this section, we describe criteria for classifying recursive functions into the polynomial complexity class, FP, the class of all polynomial time computable recursive functions. These criteria are decidable and can be checked in time depending only on the size of the logic program corresponding to the function. Our criteria are sound. We prove completeness for functions over binary strings; whether these criteria are also complete for arbitrary higher-order data structures is an open problem. A checker implementing these criteria can only have two responses: *yes* and *don't know*.

First, we present a general theorem on integer valued recursive functions given

case, the hash value of the pattern would be product of the prime numbers corresponding to the constituent type families in the pattern.

by

$$
\begin{aligned}
T(x) &= \textstyle\sum_{i=1}^{m} T(g_i(x)) + f(x) && \text{if } x > K \\
T(x) &= b && \text{if } 1 \leq x \leq K
\end{aligned}
\tag{4.1}
$$

where $x \in \mathbb{Z}^+$ and there exists functions $g_i(\cdot)$ (not defined using $T(\cdot)$) for all $i = 1, \ldots, m$ such that $g_i(x) \in \mathbb{Z}^+$ and $g_i(x) < x$ for all $x \in \mathbb{Z}^+$, each $f(x)$ is an integer valued function defined on $\mathbb{Z}^+$ (not defined using $T(\cdot)$), b and K are positive integers; and m is an positive integer constant.

Corollary 4.3.1 (Verma [70])**.** *Given a recursive function $T(x)$ defined in equation 4.1. If $f(x)$ is a monotonically increasing function such that $f(x) > 0$ for all $1 \leq x \leq K$, and $x \geq \sum_{i=1}^{m} g_i(x)$, then there exists a constant $c \geq 1$ such that $T(x) \leq cx^2 f(x)$ for all $x \geq 1$.*

Proof. (Poswolsky, Shah and Trifonov, 2005)

There exists a $d > 0$ such that $f(x) \geq d$ for all $1 \leq x \leq K$. Choose $c = max\{1, b/d\}$.

We shall prove by induction

Base case: When $1 \leq x \leq K$, $T(x) = b = (b/d)\, d \leq cf(x) \leq cx^2 f(x)$.

Induction case: Let $x_i = g_i(x)$.

When $x > K$,

$$\begin{aligned}
T(x) &= \sum_{i=1}^{m} T(x_i) + f(x) \\
&\leq \sum_{i=1}^{m} c x_i^2 f(x_i) + f(x) \\
&\qquad \text{(Using the principle of Strong induction} \\
&\qquad\qquad T(x_i) \leq c x_i^2 f(x_i)) \\
&\leq \sum_{i=1}^{m} c x_i^2 f(x) + c f(x) \\
&\qquad (\because x_i \leq x \Rightarrow f(x_i) \leq f(x) \text{ and } c \geq 1) \\
&= c f(x)(\sum_{i=1}^{m} x_i^2 + 1) \\
&\leq c x^2 f(x) \\
&\quad (\text{When } m = 1,\ x_1^2 + 1 \leq x^2,\ \because x_1 < x) \\
&\quad (\text{When } m > 1,\ \textstyle\sum_{i=1}^{m} x_i^2 + 1 \leq (\sum_{i=1}^{m} x_i)^2 \leq x^2, \\
&\qquad \because x \geq \textstyle\sum_{i=1}^{m} x_i)
\end{aligned}$$

$\square$

For example, if $T(x) = T(\lfloor x/3 \rfloor) + T(\lfloor x/4 \rfloor) + x$, then $T(x) = O(x^3)$ as $x \geq \lfloor x/3 \rfloor + \lfloor x/4 \rfloor$. On the other hand, we know that $T(x) = T(x-1) + T(x-2) + 1$ when $x \geq 2$ and $T(0) = T(1) = 1$ is not a polynomial. In this case, $x \not\geq (x-1) + (x-2)$.

Corollary 4.3.1 can be generalized to a set of functions

$$\mathcal{T} = \{T_1(\cdot), T_2(\cdot), \ldots, T_k(\cdot)\}$$

where each $T_i(\cdot)$ is defined as

$$
\begin{aligned}
T_i(x) &= \textstyle\sum_{j=1}^{m_i} T_{l_j}(g_{ij}(x)) + f_i(x) && \text{if } x > K_i \\
T_i(x) &= b_i && \text{if } 1 \leq x \leq K_i
\end{aligned}
\tag{4.2}
$$

where m_i, K_i and b_i are positive integer constants, each $l_j \in \{1, \ldots, k\}$, every $f_i(x)$ is an integer-valued function defined on $\mathbb{Z}^+$ (not defined using $T(\cdot)$), $x \in \mathbb{Z}^+$ and there exists functions $g_{ij}(\cdot)$ (not defined using $T(\cdot)$) such that $g_{ij}(x) \in \mathbb{Z}^+$ and $g_{ij}(x) < x$.

Theorem 4.3.2. *Given a set of recursive functions $\mathcal{T} = \{T_1(\cdot), T_2(\cdot), \ldots, T_k(\cdot)\}$ such that each function is given by equation 4.2. If for all $i = 1, \ldots, k$:*

1. *$f_i(\cdot)$ are monotonically increasing functions such that $f_i(x) > 0$ for all $1 \leq x \leq K_i$.*
2. *$x \geq \sum_{j=1}^{m_i} g_{ij}(x)$*

then there exists a constant $c \geq 1$ such that $T_i(x) \leq cx^2F(x)$ for all $x \geq 1$ where $F(x) = \mathsf{max}(f_1(x), f_2(x), \ldots, f_k(x))$.

Proof. For all $i = 1, \ldots, k$, there exists $d_i > 0$ such that $f_i(x) \geq d_i$ for all $1 \leq x \leq K_i$. Choose $c = max\{1, b_1/d_1, \ldots, b_k/d_k\}$ and $F(x) = max\{f_1(x), \ldots, f_k(x)\}$.

We shall prove this theorem using the induction hypothesis: $\forall i = 1, \ldots, k. y \sqsubset x \Rightarrow T_i(y) \leq cy^2F(y)$

Base Case: For any $i = 1, \ldots, k$: When $1 \leq x \leq K_i$, $T_i(x) = b_i = (b_i/d_i)d_i \leq cf_i(x) \leq cx^2F(x)$.

Induction Case: Let $x_{ij} = g_{ij}(x)$.

For any $i = 1, \ldots, k$: When $x > K$,

$$
\begin{aligned}
T_i(x) &= \sum_{j=1}^{m_i} T_{l_i}(x_{ij}) + f_i(x) \\
&\leq \sum_{j=1}^{m_i} c x_{ij}^2 F(x_{ij}) + f_i(x) \\
&\qquad \text{(Using the principle of Strong induction} \\
&\qquad\qquad T_{l_i}(x_{ij}) \leq c x_{ij}^2 F(x_{ij})) \\
&\leq \sum_{j=1}^{m_i} c x_{ij}^2 F(x_{ij}) + F(x) \\
&\qquad (\because \forall i = 1, \ldots, k. F(x) \geq f_i(x)) \\
&\leq \sum_{j=1}^{m_i} c x_{ij}^2 F(x) + cF(x) \\
&\qquad (\because x_{ij} \leq x \Rightarrow F(x_{ij}) \leq F(x) \text{ and } c \geq 1) \\
&\leq cF(x)(\sum_{j=1}^{m_i} x_{ij}^2 + 1) \\
&\leq cx^2 F(x)
\end{aligned}
$$

(When $m = 1$, $x_{i1}^2 + 1 \leq x^2$, $\because x_{i1} < x$)

(When $m > 1$, $\sum_{j=1}^{m} x_{ij}^2 + 1 \leq (\sum_{j=1}^{m} x_{ij})^2 \leq x^2$,

$\because x \geq \sum_{j=1}^{m} x_{ij}$)

$\square$

We present our result in two stages. In Section 4.3.1 we present the *basic criterion* which captures the essence of our solution. It is based on the syntactic structure of the logic programs. The functions identified by this criterion satisfy the following condition: the sum of the sizes of the recursive input arguments to the recursive calls is less than the original recursive input arguments. Thus, we are generalizing the results of Corollary 4.3.1 and Theorem 4.3.2 to higher-order data structures by defining an appropriate size function. In Section 4.3.3, we first show that Cobham's function class [15] is a special case of this criteria. Later in Section 4.3.4, we will also extend our criteria to identify functions where size of the output is bounded by a polynomial.

4.3.1 Basic criteria

We generalize Corollary 4.3.1 and Theorem 4.3.2 to mutually recursive functions on arbitrary simply-typed λ-terms.

Definition 4.3.1 (goals)**.** Given a clause D, we define the set $\mathsf{goals}(D)$ as given below.

$$\begin{array}{rcl} \mathsf{goals}(P) & = & \{\} \\ \mathsf{goals}(G \supset D) & = & \{G\} \cup \mathsf{goals}(D) \\ \mathsf{goals}(\exists x : A.D) & = & \mathsf{goals}([X/x]D) \end{array}$$

Definition 4.3.2 (GOALS)**.** Given a clause D and a predicate P with the restriction that $\mathsf{symbol}(D) = \mathsf{symbol}(P)$, and a derivation $\mathcal{D} :: \mathcal{F} \models D \gg P$, we define the set $\mathsf{GOALS}(\mathcal{D})$ as given below.

$$\begin{array}{rcl} \mathsf{GOALS}\left(\dfrac{}{\mathcal{F} \models P \gg P} \right) & = & \{\} \\ \mathsf{GOALS}\left(\dfrac{\overset{\mathcal{D}_1}{\mathcal{F} \models D \gg P} \quad \overset{\mathcal{D}_2}{\mathcal{F} \models G}}{\mathcal{F} \models G \supset D \gg P} \right) & = & \{\mathcal{D}_2\} \cup \mathsf{GOALS}(\mathcal{D}_1) \\ \mathsf{GOALS}\left(\dfrac{\overset{\mathcal{D}'}{\mathcal{F} \models [M/x]D \gg P}}{\mathcal{F} \models \exists x : A.D \gg P} \right) & = & \mathsf{GOALS}(\mathcal{D}') \end{array}$$

We would like to note that all terms that appear in $\mathsf{GOALS}(\mathcal{D})$ are *ground*. This is not true for terms that appear in $\mathsf{goals}(D)$, which may contain logic variables.

Mutual recursion

Definition 4.3.3. Given a logic program $\mathcal{F}$, a set S of predicate symbols and predicate symbols $P_f, P_g \in S$. If there exists a clause $D \in \mathcal{F}$ such that $\mathsf{symbol}(D) = P_f$ and there exist a goal $G \in \mathsf{goals}(D)$ such that $\mathsf{symbol}(G) = P_g$, then we say that predicate symbol P_f *calls* symbol P_g denoted by $P_f \rightharpoondown P_g$.

Similarly, if $P_f \rightharpoondown^* P_g$, if there exists predicate symbols $P_1, \ldots, P_k$ such that $P_f \rightharpoondown P_1 \rightharpoondown P_2 \ldots \rightharpoondown P_k \rightharpoondown P_g$.

Definition 4.3.4 (Mutually recursive predicate symbols)**.** Given a logic program $\mathcal{F}$, a set S of predicate symbols is said to be mutually recursive if and only if for any two predicate symbols $P_f, P_g \in S$, both $P_f \rightharpoondown^* P_g$ and $P_g \rightharpoondown^* P_f$ are true.

Basic criteria

Figure 4.3 shows a deductive system for identifying logic programs corresponding to polynomial time functions. We say that a logic program $\mathcal{F}$ and a corresponding set S of mutually recursive predicate symbols computes a polynomial-time function, if we can construct a proof of the judgment $\vdash_S \mathcal{F}\mathsf{poly_b}$ using the rules given in Figure 4.3. The deductive system checks that every clause $D \in \mathcal{F}$ satisfies our polynomial time criteria and the corresponding judgment is given by $\vdash_S \Delta/D$ poly, where Δ is the list of subgoals. Initially, Δ is empty; the subgoals of D are added to Δ and they are eventually used in the base rule (rule $\mathsf{b\_Atom}$).

For the sake of clarity, given a program clause D and a set S of mutually recursive predicate symbols corresponding to a function, we will refer to subgoals G such that $\mathsf{symbol}(G) \in S$ as *recursive function calls* and subgoals G such that $\mathsf{symbol}(G) \notin S$ as *auxiliary function calls.*

Informally speaking, these conditions require that every program clause D satisfy the following properties:

1. The sum of the sizes of the inputs to all recursive function calls is no greater than the size of the input to the function. (Rule $\mathsf{b\_Atom}$)

2. The size of the input to a recursive function call is strictly less than the size of the input to the function. (Rule $\mathsf{b\_Imp1}$)

3. All auxiliary function calls are polynomial-time computable functions and the sizes of the inputs to those function calls are bounded by a polynomial in the size of the input to the function. (Rule $\mathsf{b\_Imp2}$)

In our deductive system, we have omitted proofs of these conditions, but they could be implemented in standard theorem provers using an implementation of Peano's arithmetic. In the rule, $\mathsf{b\_Imp2}$, the simplification of the term $\mathsf{sz_i}(G)$ in this rule involves replacing all variables $\#(X)$ where X is a output logic variable with $\#(Y)$ where Y is an input logic variable. For example, it may be known that the relationship is non-size-increasing, i.e $\#(X) \leq \#(Y)$. The polynomial $f_G(\cdot)$ is constructed by verifying that the simplified expression $\mathsf{sz_i}(G)$ is a polynomial in the input logic variables. In Section 4.3.4, we extend this idea to allow input and output logic variables to be related by a polynomial, i.e. $\#(X) \leq p(\#(Y))$, where $p(\cdot)$ is a polynomial. If any output logic variables cannot be replaced using a relation involving just the input logic variables, then this condition cannot be proven.

The main result of this chapter is shown in Theorem 4.3.6 and it uses the Lemmas given below.

Lemma 4.3.3. *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$.*

Given a predicate P and a clause $D \in \mathcal{F}$ such that $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$, if $\mathcal{D} :: \mathcal{F} \vDash D \gg P$, then $\mathsf{sz}(\mathcal{D}) = \sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})} \mathsf{sz}(\mathcal{D}_G) + C_D$ where C_D is a

Programs:

$$\frac{}{\vdash_S \bullet\ \mathsf{poly_b}}\ \mathsf{b\_empty} \qquad \frac{\mathsf{symbol}(D) \notin S \quad \vdash_S \mathcal{F}\ \mathsf{poly_b}}{\vdash_S \mathcal{F}, D\ \mathsf{poly_b}}\ \mathsf{b\_clause1}$$

$$\frac{\mathsf{symbol}(D) \in S \quad \vdash_S \bullet/D\ \mathsf{poly_b} \vdash_S \mathcal{F}\ \mathsf{poly_b}}{\vdash_S \mathcal{F}, D\ \mathsf{poly_b}}\ \mathsf{b\_clause2}$$

Clauses:

$$\frac{}{\vdash_S \Delta/P\ \mathsf{poly_b}}\ \mathsf{b\_Atom}\left\langle \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \mathsf{sz_i}(G) \leq \mathsf{sz_i}(P) \right\rangle$$

$$\frac{\vdash_S \Delta, G/D\ \mathsf{poly_b} \quad \mathsf{symbol}(G) \in S}{\vdash_S \Delta/G \supset D\ \mathsf{poly_b}}\ \mathsf{b\_Imp1}\langle \mathsf{sz_i}(G) < \mathsf{sz_i}(D)\rangle$$

$$\frac{\vdash_S \Delta, G/D\ \mathsf{poly_b} \quad \mathsf{symbol}(G) \notin S \quad \vdash_T \mathcal{F}\ \mathsf{poly_b}}{\vdash_S \Delta/G \supset D\ \mathsf{poly_b}}\ \mathsf{b\_Imp2}\langle \mathsf{sz_i}(G) < f_G(\mathsf{sz_i}(D))\rangle$$

(where T is a set of mutually recursive predicate symbols such that $\mathsf{symbol}(G) \in T$ and $f_G(\cdot)$ is a polynomial)

$$\frac{\vdash_S \Delta/[X/x]D\ \mathsf{poly_b}}{\vdash_S \Delta/\forall x : A.D\ \mathsf{poly_b}}\ \mathsf{b\_Forall}(\cdot \vdash X : A)$$

Figure 4.3: Basic criteria for identifying for polynomial time functions

constant depending only on the structure of D and not its input terms. Moreover, $\mathsf{sz_i}(D) = \mathsf{sz_i}(P)$.

Proof. We shall prove by induction on the size of derivation $\mathcal{D}$.

(Base) Case: When the derivation $\mathcal{D}$ is given by

$$\frac{Q \doteq P}{\mathcal{F} \vDash Q \gg P}\ \mathsf{c\_Atom}$$

, $\mathsf{sz}(\mathcal{D}) = 1$ and hence the theorem is true. Also, $\mathsf{sz_i}(Q) = \mathsf{sz_i}(P)$.

Case: When the derivation $\mathcal{D}$ is given by

$$\frac{\mathcal{F} \vDash \overset{\mathcal{D}_1}{D} \gg P \quad \mathcal{F} \overset{\mathcal{D}_2}{\vDash} H}{\mathcal{F} \vDash H \supset D \gg P}\ \mathsf{c\_Imp}$$

By induction hypothesis,

$$\mathsf{sz}(\mathcal{D}_1) = \sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D}_1)} \mathsf{sz}(\mathcal{D}_G) + C_D.$$

Hence,

$$\begin{array}{rcl} \mathsf{sz}(\mathcal{D}) & = & \mathsf{sz}(\mathcal{D}_1) + \mathsf{sz}(\mathcal{D}_2) + 1 \\ \mathsf{sz}(\mathcal{D}) & = & \displaystyle\sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D}_1)} \mathsf{sz}(\mathcal{D}_G) + C_D + \mathsf{sz}(\mathcal{D}_2) + 1 \\ \mathsf{sz}(\mathcal{D}) & = & \displaystyle\sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})} \mathsf{sz}(\mathcal{D}_G) + C_{H \supset D} \\ & & (\text{where } C_{H \supset D} = C_D + 1) \end{array}$$

By induction hypothesis, $\mathsf{sz_i}(D) = \mathsf{sz_i}(P)$. Hence, $\mathsf{sz_i}(H \supset D) = \mathsf{sz_i}(P)$.

Case: When the derivation $\mathcal{D}$ is given by

$$\dfrac{\overset{\mathcal{D}'}{\mathcal{F} \vDash [M/x]D \gg P}}{\mathcal{F} \vDash \exists x : A.D \gg P}\ \mathsf{c\_Exists}$$

The proof of this case is also similar to the above cases, if we define $C_{\exists x:A.D} = C_{[M/x]D} + 1$.

$\square$

Lemma 4.3.4. *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$. Given a predicate P and a clause $D \in \mathcal{F}$ with no logic variables such that* $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$ *and* $\mathcal{E} :: \vdash_S \Delta/D\ \mathsf{poly_b}$, *if* $\mathcal{D} :: \mathcal{F} \vDash D \gg P$, *then*

1. *For all* $\mathcal{D}_G :: \mathcal{F} \vDash G \in \mathsf{GOALS}(\mathcal{D})$*, if* $\mathsf{symbol}(G) \in S$ *then* $\mathsf{sz_i}(G) < \mathsf{sz_i}(P)$ *and if* $\mathsf{symbol}(G) \in T \neq S$*, then* $\vdash_T \mathcal{F}\ \mathsf{poly_b}$ *and* $\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(P))$*.*

2. $\sum_{\substack{\mathcal{D}_G :: \mathcal{F} \vDash G \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(G) \in S}} \mathsf{sz_i}(G) + \sum_{G \in \Delta} \mathsf{sz_i}(G) \leq \mathsf{sz_i}(P)$.

Proof. We shall prove by induction on the size of the derivation $\mathcal{E}$ and $\mathcal{D}$.

(Base) Case: When the derivation $\mathcal{E}$ is given by

$$\frac{}{\vdash_S \Delta/P\ \mathsf{poly_b}}\ \mathsf{b\_Atom}\left\langle \sum_{\substack{G \in \Delta \\ \mathsf{symbol}(G) \in S}} \mathsf{sz_i}(G) \leq \mathsf{sz_i}(P) \right\rangle$$

In this case, $\mathsf{GOALS}(\mathcal{D}) = \{\}$ and the second condition is true because the side-condition of $\mathsf{b\_Atom}$ is true.

Case: When the derivation $\mathcal{E}$ is given by

$$\frac{\overset{\mathcal{E}'}{\vdash_S \Delta, G/D\ \mathsf{poly_b}} \quad \mathsf{symbol}(G) \in S}{\vdash_S \Delta/G \supset D\ \mathsf{poly_b}}\ \mathsf{b\_Imp1}\langle \mathsf{sz_i}(G) < \mathsf{sz_i}(D) \rangle$$

In this case, the derivation $\mathcal{D}$ is given by

$$\frac{\overset{\mathcal{D}_1}{\mathcal{F} \vDash D \gg P} \quad \overset{\mathcal{D}_2}{\mathcal{F} \vDash G}}{\mathcal{F} \vDash G \supset D \gg P}$$

and $\mathsf{GOALS}(\mathcal{D})$ is given by $\mathsf{GOALS}(\mathcal{D}_1) \cup \{\mathcal{D}_2\}$.

By induction hypothesis, condition 1 is true for all $\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D}_1)$.

Moreover, for $\mathcal{D}_2$ the condition $\mathsf{sz_i}(G) < \mathsf{sz_i}(D)$ is true. By Lemma 4.3.3, this implies that $\mathsf{sz_i}(G) < \mathsf{sz_i}(P)$.

For the second condition, by induction hypothesis we know that,

$$\sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}_1)\\ \mathsf{symbol}(H)\in S}} \mathsf{sz_i}(H) + \sum_{H\in\Delta,G} \mathsf{sz_i}(H) \leq \mathsf{sz_i}(P)$$

Now the left hand side can be rearranged as shown below.

$\sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}_1)\\ \mathsf{symbol}(H)\in S}} \mathsf{sz_i}(H) + \sum_{H\in\Delta} \mathsf{sz_i}(H) + \mathsf{sz_i}(G) = \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \mathsf{sz_i}(H) + \sum_{H\in\Delta} \mathsf{sz_i}(H)$

Thus, the second condition is true as well.

Case: When the derivation $\mathcal{E}$ is given by

$$\frac{\overset{\mathcal{E}'}{\vdash_S \Delta, G/D\ \mathsf{poly_b}} \quad \mathsf{symbol}(G) \notin S \quad \vdash_T \mathcal{F}\ \mathsf{poly_b}}{\vdash_S \Delta/G \supset D\ \mathsf{poly_b}}\ \mathsf{b\_Imp2}\langle \mathsf{sz_i}(G) < f_G(\mathsf{sz_i}(D))\rangle$$

This analysis for this case is similar to that of the previous case.

Case: When the derivation $\mathcal{E}$ is given by

$$\frac{\overset{\mathcal{E}'}{\vdash_S \Delta/[X/x]D\ \mathsf{poly_b}}}{\vdash_S \Delta/\forall x : A.D\ \mathsf{poly_b}}\ \mathsf{b\_Forall}(\cdot \vdash X : A)$$

and the derivation $\mathcal{D}$ is given by

$$\frac{\overset{\mathcal{D}'}{\mathcal{F} \vDash [M/x]D \gg P}}{\mathcal{F} \vDash \exists x : A.D \gg P}$$

Let $\mathcal{E}''$ be given by the derivation obtained by substituting M for the logic variable X in the derivation $\mathcal{E}'$.

Conditions 1 and 2 follow by induction hypothesis on $\mathcal{E}''$ and $\mathcal{D}$.

$\square$

Lemma 4.3.5. *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$. Given a predicate P and a non-trivial goal G (not a $\top$) , if $\mathcal{D} :: \mathcal{F} \vDash G$, then there exists a clause $D \in \mathcal{F}$ such that $\mathsf{symbol}(D) = \mathsf{symbol}(P) \in S$ and a sub-derivation $\mathcal{D}' :: \mathcal{F} \vDash D \gg P$ such that $\mathsf{sz}(\mathcal{D}) = \mathsf{sz}(\mathcal{D}') + 1$. Also, $\mathsf{sz_i}(P) = \mathsf{sz_i}(G)$ and $\mathsf{sz_o}(P) = \mathsf{sz_o}(G)$.*

Proof. Given a non-trivial goal G, it is a predicate and the only rule that is applicable is $\mathsf{g\_Atom}$. Now identifying the right clause D corresponding to that goal is independent of the input arguments to the goals. Thus, $\mathcal{D}$ is of the form given below:

$$\frac{D \in \mathcal{F} \quad \mathcal{F} \vDash \overset{\mathcal{D}'}{D} \gg P}{\mathcal{F} \vDash P}\ \mathsf{g\_Atom}$$

$\mathsf{sz_i}(G) = \mathsf{sz_i}(P)$ and $\mathsf{sz_i}(P) = \mathsf{sz_i}(G)$ follow because of the structure of G. $\square$

Theorem 4.3.6 (Basic Criteria)**.** *Given a program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$ such that $\vdash_S \mathcal{F}\ \mathsf{poly_b}$, then there exists a monotonically increasing polynomial $p(\cdot)$ such that for all goals G: if $\mathsf{symbol}(G) \in S$ and $\mathcal{D} :: \mathcal{F} \vDash G$, then $\mathsf{sz}(\mathcal{D}) \leq p(\mathsf{sz_i}(G))$.*

Proof. Using Lemma 4.3.5, we know that there exists a derivation $\mathcal{D}' :: \mathcal{F} \vDash D \gg P$ such that

$$\mathsf{sz}(\mathcal{D}) = \mathsf{sz}(\mathcal{D}') + 1.$$

Using Lemma 4.3.3, we know that

$$\mathsf{sz}(\mathcal{D}') = \sum_{\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D}')} \mathsf{sz}(\mathcal{D}_G) + C_D.$$

Hence,

$$\begin{aligned}
\mathsf{sz}(\mathcal{D}) &= \sum_{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}')} \mathsf{sz}(\mathcal{D}_H) + C_D + C_G \\
&= \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\in S}} \mathsf{sz}(\mathcal{D}_H) + C_D + 1 \\
&\quad + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\notin S}} \mathsf{sz}(\mathcal{D}_H)
\end{aligned}$$

By Lemma 4.3.4, for goals H such that $\mathsf{symbol}(H) \in T_H \neq S$, $\vdash_{T_H} \mathcal{F}$ $\mathsf{poly_b}$. Hence, by induction,

$$\begin{aligned}
\mathsf{sz}(\mathcal{D}) &\leq \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\in S}} \mathsf{sz}(\mathcal{D}_H) + C_D + 1 \\
&\quad + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\notin S}} f_{T_H}(\mathsf{sz_i}(H)) \\
&\leq \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\in S}} \mathsf{sz}(\mathcal{D}_H) + C_D + 1 \\
&\quad + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\notin S}} f_{T_H}(f_H(\mathsf{sz_i}(G))) \\
&\quad (\text{Using Lemma 4.3.4, } \mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P)) \\
&\quad \leq f_H(\mathsf{sz_i}(G)) \text{ when } \mathsf{symbol}(H) \notin S))
\end{aligned}$$

Let us define

$$F(\mathsf{sz_i}(H)) = \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D}')\\ \mathsf{symbol}(H)\notin S}} f_{T_H}(f_H(\mathsf{sz_i}(H))) + 1 + C_D^{max}$$

where $C_D^{max} = max\{C_D | D \in \mathcal{F}\}$.

We shall prove by induction on $\mathsf{sz_i}(G)$ that the polynomial $p(x) = x^2F(x)$. The theorem follows by applying Theorems 4.3.2. $\square$

According to Theorem 4.2.1, if the logic program $\mathcal{F}$ computes a function that satisfies the conditions given in Section 4.2.5, the proof derivation $\mathcal{D}$ can be implemented on a RAM in time proportional to $\mathsf{sz}(\mathcal{D})$. Moreover, $\mathsf{sz}(\mathcal{D})$ is bounded by a polynomial in the size of the input $\mathsf{sz_i}(G)$. Therefore, we can conclude that $\mathcal{F}$ is polynomial-time computable.

Example 4.3.1 (Combinators)**.** The combinators $c ::= \mathsf{S} \mid \mathsf{K} \mid \mathsf{MP}\ c_1\ c_2$ that are prevalent in programming language theory are represented as constructors of type comb. We study the complexity of the bracket abstraction algorithm ba, which converts a parametric combinator M (a representation-level function of type $\mathsf{comb} \to \mathsf{comb}$) into a combinator with one less parameter (of type comb) to which we refer as M'. The bracket abstraction algorithm is expressed by a predicate relating M and M'. Let $\mathcal{F}$ be defined as the following program.

$$
\begin{aligned}
&\mathsf{ba}\ (\lambda x : \mathsf{comb}.\, x; \mathsf{MP}\ (\mathsf{MP}\ \mathsf{S}\ \mathsf{K})\ \mathsf{K}),\\
&\mathsf{ba}\ (\lambda x : \mathsf{comb}.\, \mathsf{K}; \mathsf{MP}\ \mathsf{K}\ \mathsf{K}),\\
&\mathsf{ba}\ (\lambda x : \mathsf{comb}.\, \mathsf{S}; \mathsf{MP}\ \mathsf{K}\ \mathsf{S}),\\
&\mathsf{ba}\ (\lambda x : \mathsf{comb}.\, \mathsf{MP}\ (C_1\ x)\ (C_2\ x); \mathsf{MP}\ (\mathsf{MP}\ \mathsf{S}\ D_1)\ D_2)\\
&\quad \subset\ \mathsf{ba}\ (\lambda x : \mathsf{comb}.\, C_1\ x; D_1)\\
&\quad \subset\ \mathsf{ba}\ (\lambda x : \mathsf{comb}.\, C_2\ x; D_2).
\end{aligned}
$$

It is easy to see that $\sum_{i=1}^{2} \#(\lambda x : \mathsf{comb}.\, C_i\ x) < \#(\lambda x : \mathsf{comb}.\, \mathsf{MP}\ (C_1\ x)\ (C_2\ x))$, and hence $\vdash_{\mathsf{ba}} \mathcal{F}\ \mathsf{poly_b}$.$\square$

4.3.2 Functions with inputs from outputs of auxiliary functions

When recursive function calls receive inputs from outputs of certain auxiliary functions, we may be unable to verify the first condition in our *basic criteria* directly. In such cases, we will need additional properties that relate outputs of those auxiliary functions to their inputs. The non-size increasing property described in detail later in Section 4.3.4 could suffice. But, in general, the user or the theorem prover could use any other property that may be known to be true regarding that auxiliary function.

Example 4.3.2 (Greatest Common Divisor)**.** Consider the algorithm for computing greatest common divisor of two positive integers represented in unary notation. The logic program $\mathcal{F}$ corresponding to this algorithm is given in Figure 4.4.

For the function $\mathsf{compare}(x, y; t)$, t is true if $x < y$ and t is false otherwise.

Suppose we also know the following two properties about $\mathsf{compare}$ and $\mathsf{subtract}$,

1. $\mathsf{sz_o}(\mathsf{compare}(x, y; t)) = 1$

2. $\mathsf{sz_o}(\mathsf{subtract}(x, y; w)) = \#(x) - \#(y)$

These properties could be proved automatically in a theorem prover or simply be provided by the user. Later, in Section 4.3.4, we give a simple *sufficient* criteria for identifying non-size increasing functions. Similar criteria could be developed for identifying other properties for functions under consideration.

Now, gcd $(x, y; z)$ and gcd' $(t, x, y; z)$ are mutually recursive functions. For clauses $D \in \mathcal{F}$ such that $\mathsf{symbol}(D) \in \{\mathsf{gcd}'\}$, it can be shown that $\vdash_{\{\mathsf{gcd},\mathsf{gcd}'\}} \bullet/D\ \mathsf{poly_b}$. Here, we need to prove that $\#(\mathsf{true}) + \#(x) + \#(y) \geq \#(w) + \#(y)$. Since, $\mathsf{sz_o}(\mathsf{subtract}(y, x; w)) = \#(y) - \#(x)$, this inequality can be shown to be true.

$$\begin{array}{ll}
\mathsf{gcd}\ (\mathsf{z}, y; y), & \mathsf{subtract}(x, \mathsf{z}; x), \\
\mathsf{gcd}\ (x, \mathsf{z}; x), & \mathsf{subtract}(\mathsf{z}, y; \mathsf{z}), \\
\mathsf{gcd}\ (\mathsf{s}\ x, \mathsf{s}\ y; z) & \mathsf{subtract}\ (\mathsf{s}\ x, \mathsf{s}\ y; z) \\
\quad \subset \mathsf{compare}(x, y, t) & \quad \subset \mathsf{subtract}\ (x, y; z), \\
\quad \subset \mathsf{gcd}'(t, x, y; z), & \\
 & \mathsf{gcd}'\ (\mathsf{true}, x, y; z) \\
\mathsf{compare}\ (\mathsf{z}, y; \mathsf{true}), & \quad \subset \mathsf{subtract}\ (y, x; w) \\
\mathsf{compare}\ (x, \mathsf{z}; \mathsf{false}), & \quad \subset \mathsf{gcd}\ (x, w; z), \\
\mathsf{compare}\ (\mathsf{s}\ x, \mathsf{s}\ y; t) & \mathsf{gcd}'\ (\mathsf{false}, x, y; z) \\
\quad \subset \mathsf{compare}\ (x, y; t), & \quad \subset \mathsf{subtract}\ (x, y; w) \\
 & \quad \subset \mathsf{gcd}\ (w, y; z).
\end{array}$$

Figure 4.4: Greatest Common Divisor

For clauses $D \in \mathcal{F}$ such that $\mathsf{symbol}(D) \in \{\mathsf{gcd}\}$, it can also be shown that $\vdash_{\{\mathsf{gcd},\mathsf{gcd}'\}} \bullet/D\ \mathsf{poly_b}$. In this case, we need to prove that $\#(\mathsf{s}\ x) + \#(\mathsf{s}\ y) \geq \#(t) + \#(x) + \#(y)$. This is clearly true, since it is known that $\#(t) = 1$.

Hence, $\vdash_{\{\mathsf{gcd},\mathsf{gcd}'\}} \mathcal{F}\ \mathsf{poly_b}$ can be shown to be true.

4.3.3 Completeness on functions over natural numbers

Cobham [15] gave a characterization of polynomial-time computable functions as the least class of functions containing constant, projection, successor, and the smash function $2^{|x|\cdot|y|}$ (where $|x|$ is the length of x); and closed under ordinary composition and *bounded recursion on notation* as defined below:

Definition 4.3.5 (Bounded recursion on notation)**.** Let g, h_0, h_1 and k be functions in the class. The function f is defined by bounded recursion on notation if

$$\begin{aligned}
f(0, x_1, \ldots, x_n) &= g(x_1, \ldots, x_n) \\
f(2y, x_1, \ldots, x_n) &= h_0(y, x_1, \ldots, x_n, f(y, x_1, \ldots, x_n)) \\
f(2y+1, x_1, \ldots, x_n) &= h_1(y, x_1, \ldots, x_n, f(y, x_1, \ldots, x_n))
\end{aligned}$$

and $f(y, x_1, \ldots, x_n) \leq k(y, x_1, \ldots, x_n)$.

Of the elementary functions, constant, projection and successor can be implemented without any recursion. Consider a direct implementation of bounded recursion. In this case, we have a bound on the size of the recursive call which we can inductively assume to be a polynomial. Since polynomials are closed under composition, we can show that the total size of the inputs to h_0 and h_1 are polynomials (side condition to the rule pc_Imp2). Hence, the implementation is within our *basic criteria*. The smash function can be implemented using bounded recursion and so, it satisfies our basic criteria. The case for composition is similar – we know a polynomial bound on the size of the the output of the functions being composed.

Therefore, the Cobham's functions can be implemented in our logic programming language and they always satisfy our basic criteria. It is possible to show a similar result for Cobham's functions when defined over binary strings.

4.3.4 Polynomial time functions with bounded recursion

It is clear from the discussion in Section 4.3.3 that our *basic criteria* are unable to identify functions that have function calls which use as inputs, outputs of other recursive functions unless we know an *apriori* bound on the size of those outputs. Based on the ideas first introduced by Caseiro [10], Aehlig, *et al.* [1] and Hofmann [36] have developed type systems for identifying functions which recurse on their *safe* inputs and yet remain within polynomial-time. Such functions have the property that any function that recurses on a recursively computed value is non-size increasing. Essentially, this property ensures that the size of the output of the function is bounded. In this section, we shall extend our *basic criteria* using their idea to identify functions which have bounded output.

Non-size increasing functions

We say that a function f is non-size increasing if and only if the sum of the sizes of the output arguments is never greater than the sizes of its input arguments by more than an additive constant, i.e. $\mathsf{sz_o}(G) \leq \mathsf{sz_i}(G) + C$, where C is an integer independent of the input variables of G. The concept of multiplicity defined below will be used in building a formal deductive system to identify non-size increasing functions.

Definition 4.3.6 (Multiplicity)**.** Given a clause D, a goal $G \in \mathsf{goals}(D)$ the α and β_G multiplicities of D are defined as follows.

1. $\alpha(D)$ is defined as the maximum number of times any input variable in $\mathsf{head}(D)$ appears in the output positions of $\mathsf{head}(D)$.

2. $\beta_G(D)$ is defined as the maximum number of times any output variable in G appears in the output positions of $\mathsf{head}(D)$.

3. $\gamma(D)$ is defined as the sum of the sizes of all the term constants that appear in output positions of $\mathsf{head}(D)$.

For example, the α, β and γ multiplicities for the program corresponding to addition are as given below:

$$
\begin{aligned}
\alpha(\exists N_1 \exists N_2 \exists M. +(N_1, M; N_2) \supset +(\mathsf{s}N_1, M; \mathsf{s}N_2)) &= 0 \\
\beta_{+(N_1,M;N_2)}(\exists N_1 \exists N_2 \exists M. +(N_1, M; N_2) \supset +(\mathsf{s}N_1, M; \mathsf{s}N_2)) &= 1 \\
\gamma(\exists N_1 N_2 M. +(N_1, M; N_2) \supset +(\mathsf{s}N_1, M; \mathsf{s}N_2)) &= 1
\end{aligned}
$$

corresponding to the second declaration of addition + operation are given by 0, 1 and 1 respectively. Similarly, for a clause of the form $P(N; \mathsf{c}NN)$, $\alpha(P(N; \mathsf{c}NN))$ is given by 2 and $\gamma(P(N; \mathsf{c}NN))$ is given by 1.

The following lemma relates the size of the output of a logic program in terms of its input.

Lemma 4.3.7. *Given a program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$. Given a predicate P and a clause $D \in \mathcal{F}$ such that* $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$. *If $\mathcal{D} :: \mathcal{F} \vDash D \gg P$, then*

$$\mathsf{sz_o}(P) = \alpha(D)\mathsf{sz_i}(P) + \sum_{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})} \beta_G(D)\mathsf{sz_o}(H) + \gamma(D).$$

Proof. The size of output of a clause D given by $\mathsf{sz_o}(D)$ consists of three kinds of terms: a fixed number of term constants (accounted for by $\gamma(D)$), the input variables of D and the output variables of the subgoals G of D ($G \in \mathsf{goals}(D)$). By taking into account the α and β_G multiplicities of the variables and the fact that the output terms of P are unified with the output variables of D the theorem follows. □

The judgment corresponding to the non-size increasing property is written as $\vdash_S \mathcal{F}$ nsi and the corresponding deductive system is given in Figure 4.5. The deductive system ensures that the following conditions hold for all program clauses D:

1. For functions which make recursive function calls, the sum of the contribution to the output of the function due to the original inputs given by $\alpha(D)\mathsf{sz_i}(D)$, and due to outputs from the subgoal calls and the constant terms given by

$$\sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G) + \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\notin S}} \beta_G(P)\mathsf{sz_o}(G) + \gamma(D)$$

   is less than the input to the function $\mathsf{sz_i}(D)$. Additionally, $\sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P) = 1$. (Rule $\mathsf{nsi\_Atom_1}$)

2. For functions without recursive function calls and base cases of recursive func-

tions, the sum of the contribution to the output of the function due to the original inputs given by $\alpha(D)\mathsf{sz_i}(D)$, and due to outputs from the subgoal calls and the constant terms given by

$$\sum_{\substack{G\in\Delta\\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G) + \sum_{\substack{G\in\Delta\\ \mathsf{symbol}(G)\notin S}} \beta_G(P)\mathsf{sz_o}(G) + \gamma(D)$$

differs from the input to the function $\mathsf{sz_i}(D)$ by an additive constant C. (Rule $\mathsf{nsi\_Atom_2}$)

3. The sum of all input sizes of recursive calls is less than the input to the function. (Rule $\mathsf{nsi\_Imp1}$)

4. All auxiliary function calls are non-size increasing. (Rule $\mathsf{nsi\_Imp2}$)

These conditions are sufficient to ensure that the predicate corresponding to the clause D is non-size increasing.

Lemma 4.3.8. *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$.*

Given a predicate P and a clause $D \in \mathcal{F}$ such that $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$ and $\vdash_S \Delta/D\mathsf{nsi}$. If $\mathcal{D} :: \mathcal{F} \vDash D \gg P$, then

1. *For all $\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})$, if $\mathsf{symbol}(D) \in S$ then $\mathsf{sz_i}(G) < \mathsf{sz_i}(P)$.*

2. *For all $\mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})$, if $\mathsf{symbol}(D) \notin S$ then $\vdash_T \mathcal{F}$ nsi.*

3. $\sum_{\substack{G\in\Delta'\\ \mathsf{symbol}(G)\in S}} \beta_G(D)\mathsf{sz_i}(G) + \sum_{\substack{G\in\Delta'\\ \mathsf{symbol}(G)\notin S}} \beta_G(D)\mathsf{sz_o}(G) + \gamma(D) \leq (1-\alpha(D))\mathsf{sz_i}(P)$ *where $\Delta' = \Delta \cup \{G | \mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})\}$ and $\sum_{\substack{G\in\Delta'\\ \mathsf{symbol}(G)\in S}} \beta_G(D) = 1$ when $\exists G \in \Delta'$ such that $\mathsf{symbol}(G) \in S$.*

4. $\sum_{\substack{G\in\Delta'\\ \mathsf{symbol}(G)\notin S}} \beta_G(D)\mathsf{sz_o}(G) + \gamma(D) \leq (1-\alpha(D))\mathsf{sz_i}(P)$ *where $\Delta' = \Delta \cup \{G | \mathcal{D}_G \in \mathsf{GOALS}(\mathcal{D})\} + C$ when $\nexists G \in \Delta'$ such that $\mathsf{symbol}(G) \in S$.*

Programs:

$$\frac{}{\vdash_S \bullet\ \mathsf{nsi}}\ \mathsf{nsi\_empty}$$

$$\frac{\mathsf{symbol}(D) \notin S \quad \vdash_S \mathcal{F}\ \mathsf{nsi}}{\vdash_S \mathcal{F}, D\ \mathsf{nsi}}\ \mathsf{nsi\_clause1} \qquad \frac{\mathsf{symbol}(D) \in S \quad \vdash_S \bullet/D\ \mathsf{nsi} \quad \vdash_S \mathcal{F}\ \mathsf{nsi}}{\vdash_S \mathcal{F}, D\ \mathsf{nsi}}\ \mathsf{nsi\_clause2}$$

Clauses:

$$\frac{\vdash_S \Delta\ \mathsf{recursive}}{\vdash_S \Delta/P\ \mathsf{nsi}}\ \mathsf{nsi\_Atom}_1 \left\langle \phi_1 \wedge \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P) = 1 \right\rangle$$

$$(\text{where } \phi_1 \equiv \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G) + \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\notin S}} \beta_G(P)\mathsf{sz_o}(G) + \gamma(P) \leq (1-\alpha(P))\mathsf{sz_i}(P))$$

$$\frac{\vdash_S \Delta\ \mathsf{non\text{-}recursive}}{\vdash_S \Delta/P\ \mathsf{nsi}}\ \mathsf{nsi\_Atom}_2 \left\langle \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\notin S}} \beta_G(P)\mathsf{sz_o}(G) + \gamma(D) \leq (1-\alpha(P))\mathsf{sz_i}(P) + C \right\rangle$$

(C is a constant depending on the logic program $\mathcal{F}$)

$$\frac{\vdash_S \Delta, G/D\ \mathsf{nsi} \quad \mathsf{symbol}(G) \in S}{\vdash_S \Delta/G \supset D\ \mathsf{nsi}}\ \mathsf{nsi\_Imp1}\langle \mathsf{sz_i}(G) < \mathsf{sz_i}(D)\rangle$$

$$\frac{\vdash_S \Delta, G/D\ \mathsf{nsi} \quad \mathsf{symbol}(G) \notin S \quad \vdash_T \mathcal{F}\ \mathsf{nsi}}{\vdash_S \Delta/G \supset D\ \mathsf{nsi}}\ \mathsf{nsi\_Imp2}$$

(where T is a set of mutually recursive predicate symbols such that $\mathsf{symbol}(G) \in T$)

$$\frac{\vdash_S \Delta/D\ \mathsf{nsi}}{\vdash_S \Delta/\forall x : A.D\ \mathsf{nsi}}\ \mathsf{nsi\_Forall}$$

$$\frac{\mathsf{symbol}(G) \in S}{\vdash_S G\ \mathsf{recursive}} \qquad \frac{\mathsf{symbol}(G) \in S}{\vdash_S \Delta, G\ \mathsf{recursive}} \qquad \frac{\mathsf{symbol}(G) \notin S \quad \vdash_S \Delta\ \mathsf{recursive}}{\vdash_S \Delta, G\ \mathsf{recursive}}$$

$$\frac{\mathsf{symbol}(G) \notin S}{\vdash_S G\ \mathsf{non\text{-}recursive}} \qquad \frac{\mathsf{symbol}(G) \notin S \quad \vdash_S \Delta\ \mathsf{non\text{-}recursive}}{\vdash_S \Delta, G\ \mathsf{non\text{-}recursive}}$$

Figure 4.5: *Sufficient* conditions for non-size increasing functions.

Proof. The proof is by induction on the size of the derivation $\mathcal{D}$ and is similar to the proof of Lemma 4.3.4. □

Theorem 4.3.9 (Non-size increasing functions). *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$ such that $\vdash_S \mathcal{F}$ nsi. For all goals G, if $\mathcal{D} :: \mathcal{F} \vDash G$, then $\mathsf{sz_o}(G) \leq \mathsf{sz_i}(G) + C$ where C is a constant depending only on the logic program $\mathcal{F}$.*

Proof. We shall prove by induction, first on the call graph generated by the mutually recursive functions and then on the size of the derivation.

Using Lemma 4.3.5, we know that there exists a derivation $\mathcal{D}' :: \mathcal{F} \vDash D \gg P$ such that $\mathsf{sz_i}(G) = \mathsf{sz_i}(P)$ and $\mathsf{sz_o}(G) = \mathsf{sz_o}(P)$.

From Lemma 4.3.7, we know that

$$\begin{aligned}
\mathsf{sz_o}(G) &= \alpha(D)\mathsf{sz_i}(G) + \sum_{\mathcal{D}_H :: \mathcal{F} \vDash H \in \mathsf{GOALS}(\mathcal{D})} \beta_H(D)\mathsf{sz_o}(H) \\
&\quad + \gamma(D) \\
&= \alpha(D)\mathsf{sz_i}(G) + \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \vDash H \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz_i}(H) + \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \vDash H \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(H) \notin S}} \beta_H(D)\mathsf{sz_o}(H) \\
&\quad + \gamma(D)
\end{aligned}$$

When the function has recursive calls and $\mathsf{symbol}(H) \in S$, we know that $\mathsf{sz_i}(H) < \mathsf{sz_i}(P)$ (Lemma 4.3.8). So, by induction hypothesis, $\mathsf{sz_o}(H) \leq \mathsf{sz_i}(H) + C$. (Note that we also know by induction hypothesis on the call graph that for $\mathsf{symbol}(H) \notin S$, $\mathsf{sz_o}(H) \leq \mathsf{sz_i}(H) + C_H$ where C_H is a constant.)

Hence,

$$\begin{aligned}
\mathsf{sz_o}(G) &= \alpha(D)\mathsf{sz_i}(G) + \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \vDash H \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz_i}(H) + C \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \vDash H \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(H) \in S}} \beta_H(D) \\
&\quad + \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \vDash H \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(H) \notin S}} \beta_H(D)\mathsf{sz_o}(H) + \gamma(D) \\
&\leq \mathsf{sz_i}(G) + C \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \vDash H \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(H) \in S}} \beta_H(D) \\
&= \mathsf{sz_i}(G) + C
\end{aligned}$$

This is because,

1.

$$\sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_i}(H) + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\notin S}} \beta_H(D)\mathsf{sz_o}(H) + \gamma(D) \leq (1-\alpha(D))\mathsf{sz_i}(G)$$

2. $\sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \beta_H(D) = 1$

are implied by $\vdash_S \mathcal{F}$ nsi and $\mathsf{sz_i}(G) = \mathsf{sz_i}(P)$ (Lemma 4.3.8).

When the function has no recursive calls,

$$\sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_i}(H) = \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \beta_H(D) = 0.$$

In this case,

$$\begin{aligned}
\mathsf{sz_o}(G) &= \alpha(D)\mathsf{sz_i}(G) + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_i}(H) \\
&\quad + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\notin S}} \beta_H(D)\mathsf{sz_o}(H) + \gamma(D) \\
&= \alpha(D)\mathsf{sz_i}(G) + \sum_{\substack{\mathcal{D}_H::\mathcal{F}\vDash H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\notin S}} \beta_H(D)\mathsf{sz_o}(H) + \gamma(D) \\
&\leq \mathsf{sz_i}(G) + C
\end{aligned}$$

Here the relevant condition implied by Lemma 4.3.8 is given by,

$$\sum_{\substack{\mathcal{D}_H::\mathcal{F}\models H\in\mathsf{GOALS}(\mathcal{D})\\ \mathsf{symbol}(H)\notin S}} \beta_H(D)\mathsf{sz_o}(H)\gamma(D) \leq (1-\alpha(D))\mathsf{sz_i}(G) + C.$$

□

Dependence Paths

The definitions of *dependence paths* given below assist us in keeping track of outputs of function calls when they are used as inputs to other function calls.

Definition 4.3.7. Given a clause D, and goals G and H in the clause, $H \leftsquigarrow_m G$ iff variables of G in output positions appear in input positions of H and no variable of G appears more than m times in H.

Definition 4.3.8 (Dependence Path)**.** Given a clause D and goals $H = G_0, G_1, \ldots, G_n = G \in \mathsf{goals}(D)$, a dependence path from G to H of length n denoted by $H \leftsquigarrow G$ is a sequence of goal and positive integer pairs $(G_1, m_1), \ldots, (G_n = G, m_n)$ such that for each pair of goals G_i, G_{i+1} for $i = 0, \ldots, n-1$, $G_i \leftsquigarrow_{m_{i+1}} G_{i+1}$. The width of this dependence path is defined as $\Pi_{i=1}^{n} m_i$.

For example, consider the example of Fibonacci numbers from Section 4.2.3. In this case, there are two dependence paths each of length 1 from $\mathsf{fib}(N;X)$ to $+(X,Y;Z)$ and from $\mathsf{fib}(\mathsf{s}\ N;\ Y)$ to $+(X,Y;Z)$.

It is worth noting that dependence paths are a structural property of a logic program and hence identifying dependence paths is independent of any of the inputs to the program.

Definition 4.3.9 (Set of Dependence Paths)**.** Given a clause D and two goals $G, H \in \mathsf{goals}(D)$, $H \lhd^* G$ is the set of all dependence paths from G to H

$$\frac{\vdash_S H \lhd D}{\vdash_S H \lhd \forall x : A.D}\ \mathsf{dp\_Forall} \qquad \frac{\vdash_S H \lhd D}{\vdash_S H \lhd G \supset D}\ \mathsf{dp\_Imp1}\langle H \not\leftsquigarrow G\rangle$$

$$\frac{\mathsf{symbol}(G) \in S}{\vdash_S H \lhd G \supset D}\ \mathsf{dp\_Imp2}\langle H \leftsquigarrow G\rangle \quad \frac{\mathsf{symbol}(G) \notin S \quad \vdash_S G \lhd D}{\vdash_S H \lhd G \supset D}\ \mathsf{dp\_Imp3/1}\langle H \leftsquigarrow G\rangle$$

$$\frac{\mathsf{symbol}(G) \notin S \quad \vdash_S H \lhd D}{\vdash_S H \lhd G \supset D}\ \mathsf{dp\_Imp3/2}\langle H \leftsquigarrow G\rangle$$

$$\frac{\vdash_S H \not\lhd D}{\vdash_S H \not\lhd \forall x : A.D}\ \mathsf{ndp\_Forall} \qquad \frac{\vdash_S H \not\lhd D}{\vdash_S H \not\lhd G \supset D}\ \mathsf{ndp\_Imp1}\langle H \not\leftsquigarrow G\rangle$$

$$\frac{\mathsf{symbol}(G) \notin S \quad \vdash_S G \not\lhd D \quad \vdash_S H \not\lhd D}{\vdash_S H \not\lhd G \supset D}\ \mathsf{ndp\_Imp2}\langle H \leftsquigarrow G\rangle$$

Figure 4.6: Proving existence and non-existence of dependence paths

For a clause D and a goal H, we define a judgment $\vdash_S H \lhd D$ which is provable if and only if there exists a goal $G \in \mathsf{goals}(D)$ such that $\mathsf{symbol}(G) \in S$ and there is a dependence path from G to H. Similarly, we define the judgment $\vdash_S H \not\lhd D$. Figure 4.6 gives the deductive systems corresponding to these judgments.

Criteria for functions with bounded recursion

Now we can define an extended version of the conditions given in Figure 4.3; the corresponding judgment is given by $\vdash_S \mathcal{F}\ \mathsf{poly_{br}}$. In this case, $\vdash_S \mathcal{F}\ \mathsf{poly_{\{b,br\}}}$ means that either $\vdash_S \mathcal{F}\ \mathsf{poly_b}$ or $\vdash_S \mathcal{F}\ \mathsf{poly_{br}}$ is true.

These conditions are given in Figure 4.7 below and they generalize the conditions given earlier. In this case, we distinguish between functions that have function calls that use output of a recursive function call and functions that do not. We require that in the former case, the function calls which use output of a recursive call are non-size increasing in addition to being polynomial-time computable (compare rules br_Imp2/1 and br_Imp2/2). The conditions ensure that the size of the output of the logic programs which satisfy these criteria is polynomially bounded in their input. In the rule

Programs:

$$\frac{}{\vdash_S \bullet\ \mathsf{poly_{br}}}\ \mathsf{br\_empty}$$

$$\frac{\mathsf{symbol}(D) \in S \quad \vdash_S \bullet/D\ \mathsf{poly_{br}} \quad \vdash_S \mathcal{F}\ \mathsf{poly_{br}}}{\vdash_S \mathcal{F}, c : D\ \mathsf{poly_{br}}}\ \mathsf{br\_clause1} \qquad \frac{\mathsf{symbol}(D) \notin S \quad \vdash_S \mathcal{F}\ \mathsf{poly_{br}}}{\vdash_S \mathcal{F}, c : D\ \mathsf{poly_{br}}}\ \mathsf{br\_clause2}$$

Clauses:

$$\frac{}{\vdash_S \Delta/P\ \mathsf{poly_{br}}}\ \mathsf{br\_Atom}\left\langle \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G) + \sum_{\substack{H\in\Delta \\ \mathsf{symbol}(H)\notin S}} \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S \\ p\in H\lhd^* G}} \beta_H(P)\mathsf{sz_i}(G)\mathsf{width}(p) \le \mathsf{sz_i}(P)\right\rangle$$

$$\frac{\vdash_S \Delta/[X/x]D\ \mathsf{poly_{br}}}{\vdash_S \Delta/\forall x : A.D\ \mathsf{poly_{br}}}\ \mathsf{br\_Forall} \qquad \frac{\vdash_S \Delta, G/D\ \mathsf{poly_{br}} \quad \mathsf{symbol}(G) \in S}{\vdash_S \Delta/G \supset D\ \mathsf{poly_{br}}}\ \mathsf{br\_Imp1}\langle \mathsf{sz_i}(G) < \mathsf{sz_i}(D)\rangle$$

$$\frac{\vdash_S \Delta, G/D\ \mathsf{poly_{br}} \quad \mathsf{symbol}(G) \notin S \quad \vdash_S G \lhd D \quad \vdash_T \mathcal{F}\ \mathsf{nsi} \quad \vdash_T \mathcal{F}\ \mathsf{poly_{\{br,u\}}}}{\vdash_S \Delta/G \supset D\ \mathsf{poly_{br}}}\ \mathsf{br\_Imp2/1}$$

(where T is a set of mutually recursive predicate symbols such that $\mathsf{symbol}(G) \in T$)

$$\frac{\vdash_S \Delta, G/D\ \mathsf{poly_{br}} \quad \mathsf{symbol}(G) \notin S \quad \vdash_S G \not\lhd D \quad \vdash_T \mathcal{F}\ \mathsf{poly_{\{br,u\}}}}{\vdash_S \Delta/G \supset D\ \mathsf{poly_{br}}}\ \mathsf{br\_Imp2/2}$$

(where T is a set of mutually recursive predicate symbols such that $\mathsf{symbol}(G) \in T$)

Figure 4.7: Criteria for identifying polynomial-time functions with bounded recursion

pc_Atom we require that the sum of all the inputs to the recursive calls is not larger than the original input. We require that we count the inputs to those recursive calls whose outputs have been used either as input to other function calls or in the final output (with corresponding multiplicities). Thus, the sum $\sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S}} \beta_G(P)\mathsf{sz_i}(G)$ accounts for the first case and $\displaystyle\sum_{\substack{H\in\Delta \\ \mathsf{symbol}(H)\notin S}} \sum_{\substack{G\in\Delta \\ \mathsf{symbol}(G)\in S \\ p\in H\lhd^* G}} \beta_H(P)\mathsf{sz_i}(G)\mathsf{width}(p)$ for the second.

This ensures that the input arguments to goal H are polynomial in the original input arguments of the clause D. Hence, the third condition of our basic criteria (rule br_Imp2 in Figure 4.3) is satisfied.

Lemma 4.3.10 and Theorem 4.3.11 give the correctness results for the deductive system given in Figure 4.7.

Lemma 4.3.10 (Bounded Recursion)**.** *Given a logic program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$. Given a predicate P and a clause $D \in \mathcal{F}$ such that* $\mathsf{symbol}(P) = \mathsf{symbol}(D) \in S$ *and* $\vdash_S \Delta/D\ \mathsf{poly_{br}}$.

If $\mathcal{D} :: \mathcal{F} \vDash D \gg P$*, then*

1. *For all* $\mathcal{D}_G :: \mathcal{F} \vDash G \in \mathsf{GOALS}(\mathcal{D})$*, if* $\mathsf{symbol}(G) \in S$*, then* $\mathsf{sz_i}(G) < \mathsf{sz_i}(P)$*.*

2. *For all* $\mathcal{D}_G :: \mathcal{F} \vDash G \in \mathsf{GOALS}(\mathcal{D})$*, if* $\mathsf{symbol}(G) \in T \neq S$ *and* $\mathcal{E} :: \vdash G \lhd D$*, then there exists a polynomial* $f_G(\cdot)$ *(depending only on* G*) such that*

$$\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(P)) + \sum_{\substack{\mathcal{D}_H :: \mathcal{F} \vDash H \in \mathsf{GOALS}(\mathcal{D}) \\ \mathsf{symbol}(H) \in S \\ p \in G \lhd^* H}} \mathsf{sz_o}(H)\mathsf{width}(p)$$

and $\vdash_T \mathcal{F}$ nsi.

3. *For all* $\mathcal{D}_G :: \mathcal{F} \vDash G \in \mathsf{GOALS}(\mathcal{D})$*, if* $\mathsf{symbol}(G) \in T \neq S$ *and* $\mathcal{E} :: \vdash G \not\lhd D$*, then there exists a polynomial* $f_G(\cdot)$ *such that* $\mathsf{sz_i}(G) \leq f_G(\mathsf{sz_i}(D))$ *and* $\vdash_S \mathcal{F}$ $\mathsf{poly_{br}}$.

4. $$\left(\sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S}} \sum_{\substack{G \in \Delta' \\ \mathsf{symbol}(G) \in S \\ p \in H \lhd^* G}} \beta_H(D)\mathsf{sz_i}(G)\mathsf{width}(p) \right) + \left(\sum_{\substack{G \in \Delta' \\ \mathsf{symbol}(G) \in S}} \beta_G(D)\mathsf{sz_i}(G) \right) \leq \mathsf{sz_i}(P)$$

   where

$$\Delta' = \Delta \cup \{G | \mathcal{D}_G :: \mathcal{F} \vDash G \in \mathsf{GOALS}(\mathcal{D})\}.$$

Proof. Let $\Delta'' = \{G | \mathcal{D}_G :: \mathcal{F} \vDash G \in \mathsf{GOALS}(\mathcal{D})\}$.

For $G \in \Delta''$, if $\mathsf{symbol}(G) \in T$ and $\mathcal{E} :: \vdash_S G \not\lhd D$, then all terms that appear in input positions in G are either the terms from input positions of D or from output positions of goals H such that $\mathcal{E}' :: \vdash_S H \not\lhd D$. We can show by induction that for such goals $\mathsf{sz}_\mathsf{o}(H) \leq p(\mathsf{sz}_\mathsf{i}(D))$ for some polynomial $p(\cdot)$. Note that this induction is based on the partial ordering produced by $\rightharpoondown$ extended to sets of mutually recursive predicate symbols. Hence there exists a polynomial $f_G(\cdot)$ such that $\mathsf{sz}_\mathsf{i}(G) \leq f_G(\mathsf{sz}_\mathsf{i}(D))$.

For $G \in \Delta''$, if $\mathsf{symbol}(G) \in T$ and $\mathcal{E} :: \vdash_S G \lhd D$, then all terms that appear in input positions in G are either from input positions of D, output positions of goals H such that $\mathcal{E}' :: \vdash_S H \ntriangleleft D$ or form output positions of goals H such that $\mathcal{E}' :: \vdash_S H \lhd D$.

For the first two cases, we have already shown that there exists a polynomial $f'_G(\cdot)$ that bounds the total contribution to $\mathsf{sz}_\mathsf{i}(G)$ due to the terms that satisfy the conditions of these two cases. Thus,

$$\mathsf{sz}_\mathsf{i}(G) \leq f'_G(\mathsf{sz}_\mathsf{i}(D)) + \sum_{\substack{H \in \Delta'' \\ \vdash_S H \lhd D \\ G \leftsquigarrow_m H}} m\mathsf{sz}_\mathsf{o}(H).$$

We shall bound the contribution due to the third case using induction on the length of dependence paths ending in a goal I such that $\mathsf{symbol}(I) \in S$. For the base case (length of dependence paths is 1), we have,

$$
\begin{aligned}
\mathsf{sz_i}(G) &\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta''\\ \vdash_S H\lhd D\\ G\leftsquigarrow_m H}} m\mathsf{sz_o}(H) \\
&\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta''\\ \mathsf{symbol}(H)\in S\\ \vdash_S H\lhd D\wedge G\leftsquigarrow_m H}} m\mathsf{sz_o}(H) + \\
&\quad \sum_{\substack{H\in\Delta''\\ \mathsf{symbol}(H)\notin S\\ \vdash_S H\lhd D\wedge G\leftsquigarrow_m H}} m\mathsf{sz_o}(H) \\
&\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta''\\ \mathsf{symbol}(H)\in S\\ \vdash_S H\lhd D\wedge G\leftsquigarrow_m H}} m\mathsf{sz_o}(H) + 0 \\
&\quad \text{(As all dependence paths have length 1)} \\
&\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta''\\ \mathsf{symbol}(H)\in S\\ p\in G\lhd^* H}} \mathsf{width}(p)\mathsf{sz_o}(H)
\end{aligned}
$$

In this case $f_G(\cdot) = f'_G(\cdot)$.

For the induction case,

$$
\begin{aligned}
\mathsf{sz_i}(G) &\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta''\\ \vdash_S H\lhd D\\ G\leftsquigarrow_m H}} m\mathsf{sz_o}(H) \\
&\leq f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H\in\Delta''\\ \mathsf{symbol}(H)\notin S\\ \vdash_S H\lhd D\wedge G\leftsquigarrow_m H}} m(\mathsf{sz_i}(H) + C) \\
&\quad + \sum_{\substack{H\in\Delta''\\ \mathsf{symbol}(H)\in S\\ \vdash_S H\lhd D\wedge G\leftsquigarrow_m H}} m\mathsf{sz_o}(H) \\
&\quad \text{(For } \mathsf{symbol}(H)\in U \text{ and } \vdash_S H\lhd D, \vdash_U \mathcal{F}\ \mathsf{nsi} \\
&\quad \text{and using Theorem 4.3.9.)} \qquad (4.3)
\end{aligned}
$$

By induction hypothesis,

$$\mathsf{sz_i}(H) \leq f'_H(\mathsf{sz_i}(D)) + \sum_{\substack{I \in \Delta'' \\ \mathsf{symbol}(I) \in S \\ q \in H \lhd^* I}} \mathsf{sz_o}(I)\mathsf{width}(q) \tag{4.4}$$

where $f'_H(\cdot)$ is a polynomial.

By substituting right side of equation 4.4 for $\mathsf{sz_i}(H)$ in equation 4.3 we get,

$$\begin{aligned}
\mathsf{sz_i}(G) &\leq f_G(\mathsf{sz_i}(D)) + \sum_{\substack{H \in \Delta'' \\ \mathsf{symbol}(H) \in S \\ \vdash_S H \lhd D \wedge G \leftrightsquigarrow_m H}} m\mathsf{sz_o}(H) \\
&\quad + \sum_{\substack{H \in \Delta'' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \lhd D \wedge G \leftrightsquigarrow_m H}} \sum_{\substack{I \in \Delta'' \\ \mathsf{symbol}(I) \in S \\ q \in H \lhd^* I}} m\mathsf{width}(q)\mathsf{sz_o}(I) \\
&\leq f_G(\mathsf{sz_i}(D)) + \sum_{\substack{H \in \Delta'' \\ \mathsf{symbol}(H) \in S \\ \vdash_S H \lhd D \wedge G \leftrightsquigarrow_m H}} m\mathsf{sz_o}(H) \\
&\quad + \sum_{\substack{I \in \Delta'' \\ \mathsf{symbol}(I) \in S \\ r \in G \lhd^* I \wedge \mathsf{length}(r) > 1}} \mathsf{width}(r)\mathsf{sz_o}(I) \\
&\leq f_G(\mathsf{sz_i}(D)) + \sum_{\substack{I \in \Delta'' \\ \mathsf{symbol}(I) \in S \\ r \in G \lhd^* I}} \mathsf{width}(r)\mathsf{sz_o}(I)
\end{aligned}$$

where $f_G(\mathsf{sz_i}(D))$ is a polynomial and is given by

$$f'_G(\mathsf{sz_i}(D)) + \sum_{\substack{H \in \Delta'' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \lhd D \wedge G \leftrightsquigarrow_m H}} m\left(f'_H(\mathsf{sz_i}(D)) + C\right).$$

The remaining cases of the proof is by induction on the size of the derivation $\mathcal{D}$ is quite similar to the proof of Lemma 4.3.4. □

Theorem 4.3.11 (Bounded Recursion). *Given a program $\mathcal{F}$ and a set S of mutually recursive predicate symbols from $\mathcal{F}$ such that $\vdash_S \mathcal{F}$ $\mathsf{poly_{br}}$, then there exists monoton-*

ically increasing polynomials $p(\cdot)$ and $p'(\cdot)$ such that for all goals G: if $\mathsf{symbol}(G) \in S$ and $\mathcal{D} :: \mathcal{F} \models G$, then $\mathsf{sz_o}(G) \leq p(\mathsf{sz_i}(G))$ and $\mathsf{sz}(\mathcal{D}) \leq p'(\mathsf{sz_i}(G))$.

Proof. Let the derivation $\mathcal{D}$ be given by

$$\frac{D \in \mathcal{F} \quad \mathcal{F} \models \overset{\mathcal{D}'}{D} \gg P}{\mathcal{F} \models P}$$

and

$$\Delta' = \{H | \mathcal{D}_H :: \mathcal{F} \models H \in \mathsf{GOALS}(\mathcal{D})\}$$

By Lemma 4.3.5 and Lemma 4.3.7, we know that,

$$\begin{aligned}
\mathsf{sz_o}(G) &\leq \alpha(D)\mathsf{sz_i}(G) + \sum_{H \in \Delta'} \beta_H(D)\mathsf{sz_o}(H) + C \\
&\leq \alpha(D)\mathsf{sz_i}(G) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz_o}(H) \\
&\quad + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S}} \beta_H \mathsf{sz_o}(H) + C \\
&\leq \alpha(D)\mathsf{sz_i}(D) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz_o}(H) \\
&\quad + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \Gamma' \vdash_S H \lhd D}} \beta_H(D)\mathsf{sz_o}(H) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \Gamma' \vdash_S H \ntriangleleft D}} \beta_H(D)\mathsf{sz_o}(H) \\
&\quad + C'_m
\end{aligned}$$

In this case, C is the total size of the term constants appearing in output positions of D. Clearly, it is a constant (depending only on $\mathcal{F}$). Let C'_m be the maximum among all such constants.

By Lemma 4.3.10, for goals $H \in \Delta'$ such that $\mathsf{symbol}(H) \in T$, $\vdash_T \mathcal{F}$ nsi if

$\vdash_S H \lhd D$ and $\vdash_T \mathcal{F}$ $\mathsf{poly_{br}}$ if $\vdash_S H \not\lhd D$.

By Theorem 4.3.9, $\mathsf{sz_o}(H) \leq \mathsf{sz_i}(H) + C'$ in the former case, where C' is a constant. In the latter case, we can show by induction on the call graph of $\mathcal{F}$ rooted at S that $\mathsf{sz_o}(H) \leq p_T(\mathsf{sz_i}(H))$ where $p_T(\cdot)$ is a polynomial depending only on T. Hence,

$$\begin{aligned}
\mathsf{sz_o}(G) \;\; \leq \;\; & \alpha(D)\mathsf{sz_i}(P) + \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_o}(H) \\
& + \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H \lhd D}} \beta_H(D)\mathsf{sz_i}(H) + \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H \not\lhd D}} \beta_H(D) p_T(\mathsf{sz_i}(H)) \\
& + C'_m + C' \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H \lhd D}} \beta_H(D) \\
\leq \;\; & \alpha(D)\mathsf{sz_i}(G) + \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_o}(H) \\
& + \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H \lhd D}} \beta_H(D)\mathsf{sz_i}(H) \\
& + \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H \not\lhd D}} \beta_H(D) p_T(f_H(\mathsf{sz_i}(G))) + C_m \\
& \text{(By Lemma 4.3.10, } \mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P)) \leq f_H(\mathsf{sz_i}(G))) \\
& \text{(and } C_m = C'_m + C' \textstyle\sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H \lhd D}} \beta_H(D)) \\
\leq \;\; & F_1(\mathsf{sz_i}(G)) + \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H \lhd D}} \beta_H(D)\mathsf{sz_i}(H) \\
& + \sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\in S}} \beta_H(D)\mathsf{sz_o}(H) \qquad (4.5) \\
& \text{(where } F_1(\mathsf{sz_i}(G)) = \alpha(D)\mathsf{sz_i}(G) + C_m \\
& \quad + \textstyle\sum_{\substack{H\in\Delta' \\ \mathsf{symbol}(H)\notin S \\ \vdash_S H \not\lhd D}} \beta_H(D) p_T(f_H(\mathsf{sz_i}(G))))
\end{aligned}$$

By Lemma 4.3.10, we have

$$\mathsf{sz_i}(H) \leq f'_H(\mathsf{sz_i}(P)) + \sum_{\substack{I \in \Delta' \\ \mathsf{symbol}(I) \in S \\ p \in H \lhd^* I}} \mathsf{sz_o}(I)\mathsf{width}(p)$$

where $f'_H(\cdot)$ is a polynomial.

Substituting in equation 4.5, we get,

$$\begin{aligned}
\mathsf{sz_o}(G) \quad \leq \quad & F_1(\mathsf{sz_i}(G)) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz_o}(H) \\
& + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \lhd D}} \beta_H(D) f'_H(\mathsf{sz_i}(D)) \\
& + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \lhd D}} \beta_H(D) \left(\sum_{\substack{I \in \Delta' \\ \mathsf{symbol}(I) \in S \\ p \in H \lhd^* I}} \mathsf{sz_o}(I)\mathsf{width}(p) \right) \\
\leq \quad & F(\mathsf{sz_i}(G)) + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \in S}} \beta_H(D)\mathsf{sz_o}(H) \\
& + \sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \lhd D}} \sum_{\substack{I \in \Delta' \\ \mathsf{symbol}(I) \in S \\ p \in H \lhd^* I}} \beta_H(D)\mathsf{sz_o}(I)\mathsf{width}(p) \\
& (\text{where } F(\mathsf{sz_i}(D)) = F_1(\mathsf{sz_i}(D) \\
& \qquad + \textstyle\sum_{\substack{H \in \Delta' \\ \mathsf{symbol}(H) \notin S \\ \vdash_S H \lhd D}} \beta_H(D) f'_H(\mathsf{sz_i}(D)))
\end{aligned}$$

Now, by Lemma 4.3.10, we know that,

$$\left(\sum_{\substack{G \in \Delta \\ \mathsf{symbol}(G) \in S}} \alpha_G \mathsf{sz_i}(G) \right) + \left(\sum_{\substack{H \in \Delta \\ \mathsf{symbol}(H) \notin S}} \sum_{\substack{G \in \Delta \\ \mathsf{symbol}(G) \in S \\ p \in H \lhd^* G}} \alpha_H \mathsf{sz_i}(G)\mathsf{width}(p) \right) \leq \mathsf{sz_i}(P)$$

Choose polynomial $p(x) = x^2 F(x)$ and the remainder of the proof follows by induction on $\mathsf{sz_o}(G)$. It is similar to the proofs of Corollary 4.3.1 and Theorem 4.3.2.

$\mathsf{mergesort}(\mathsf{nil}; \mathsf{nil})$
$\mathsf{mergesort}(\mathsf{cons}\ x\ xs; w)$
$\quad\subset \mathsf{split}(\mathsf{cons}\ x\ xs; y, z)$
$\quad\subset \mathsf{mergesort}(y; y_1)$
$\quad\subset \mathsf{mergesort}(z; z_1)$
$\quad\subset \mathsf{merge}(y_1, z_1; w)$

$\mathsf{split}(\mathsf{nil}; \mathsf{nil}, \mathsf{nil})$
$\mathsf{split}(\mathsf{cons}\ x\ \mathsf{nil}; \mathsf{cons}\ x\ \mathsf{nil}, \mathsf{nil})$
$\mathsf{split}(\mathsf{cons}\ x\ (\mathsf{cons}\ y\ xs); \mathsf{cons}\ x\ x_1, \mathsf{cons}\ y\ y_1)$
$\quad\subset \mathsf{split}(xs; x_1, y_1)$

$\mathsf{merge}(\mathsf{nil}, w; w)$
$\mathsf{merge}(w, \mathsf{nil}; w)$
$\mathsf{merge}(\mathsf{cons}\ x\ xs, \mathsf{cons}\ y\ ys; \mathsf{cons}\ u\ z)$
$\quad\subset \mathsf{compare}(x, y; t)$
$\quad\subset \mathsf{merge}'(t, \mathsf{cons}\ x\ xs, \mathsf{cons}\ y\ ys; u, v, w)$
$\quad\subset \mathsf{merge}(v, w; z)$

$\mathsf{merge}'(\mathsf{true}, \mathsf{cons}\ x\ xs, \mathsf{cons}\ y\ ys; x, xs,$
$\qquad \mathsf{cons}\ y\ ys)$
$\mathsf{merge}'(\mathsf{false}, \mathsf{cons}\ x\ xs, \mathsf{cons}\ y\ ys; y,$
$\qquad \mathsf{cons}\ x\ xs, ys)$

Figure 4.8: Merge Sort

To prove that $\mathsf{sz}(\mathcal{D}) \leq p'(\mathsf{sz_i}(G))$, we shall first need to show that for all $H \in \Delta'$ such that $\mathsf{symbol}(H) \notin S$, $\mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P))$ for some polynomial $f_H(\cdot)$. All terms that appear in input positions of H are either sub-terms of the terms in input positions of D or from output positions of other goals H. When $\mathsf{symbol}(H) \in S$, we have already proved that $\mathsf{sz_o}(H) \leq p_1(\mathsf{sz_i}(G))$ and when $\mathsf{symbol}(H) \in T \neq S$, we know that $\vdash_T \mathcal{F}\ \mathsf{poly_{br}}$ and hence $\mathsf{sz_o}(H) \leq p_2(\mathsf{sz_i}(G))$ for some monotonically increasing polynomials $p_1(\cdot)$ and $p_2(\cdot)$. Hence, $\mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P))$ for some polynomial $f_H(\cdot)$. Now it is possible to show that the conditions given in Figure 4.3 are satisfied. The condition b_Atom is always true if pp_Atom is true and the condition b_Imp2 is true as $\mathsf{sz_i}(H) \leq f_H(\mathsf{sz_i}(P))$. $\square$

Example 4.3.3 (Merge Sort)**.** Consider a representation of a list using the constants nil and cons. The logic program $\mathcal{F}$ corresponding to merge sort is given in Figure 4.8.

In this example $\mathsf{compare}(x, y; t)$, t is true if $x < y$ and t is false otherwise (clauses are given in Figure 4.4). It is not hard to see that $\vdash_{\mathsf{compare}} \mathcal{F}\ \mathsf{poly_b}$.

It is also clear that $\vdash_{\mathsf{split}} \mathcal{F}\ \mathsf{poly_b}$ as $\#(xs) \leq \#(\mathsf{cons}\ (\mathsf{cons}\ y\ xs))$ for the third declaration of split. The predicate merge' is also in polynomial time as it is not recursive. We can also check that $\vdash_{\mathsf{merge}'} \mathcal{F}\ \mathsf{nsi}$. In this case, the side condition of $\mathsf{nsi\_Atom_2}$ is satisfied because $\alpha(\cdot) = 1$ and $\beta_G(\cdot) = 0$ for both declarations of merge'. In fact, we can show that $\mathsf{sz_o}(\mathsf{merge}'(G)) = \mathsf{sz_i}(\mathsf{merge}'(G)) - 2$ when given some input through a goal G

We can also show that $\vdash_{\mathsf{merge}} \mathcal{F}\ \mathsf{poly_b}$. For this we need to show that $\#(v) + \#(w) \leq \#(\mathsf{cons}\ x\ xs) + \#(\mathsf{cons}\ y\ ys)$. It is true because merge' is non-size increasing and we know that $1 + \#(\mathsf{cons}\ x\ xs) + \#(\mathsf{cons}\ y\ ys) - 2 = \#(u) + \#(v) + \#(w)$. We can also show that merge is non-size increasing. Here $\alpha(\mathsf{merge}'(\cdot)) = \alpha(\mathsf{merge}(\cdot)) = 1$ and we need to show that $\#(\mathsf{cons}) + \#(u) + \#(v) + \#(w) \leq \#(\mathsf{cons}\ x\ xs) + \#(\mathsf{cons}\ y\ ys)$. This follows from the fact that merge' is non-size increasing.

Finally, it needs to be shown that $\vdash_{\mathsf{mergesort}} \mathcal{F}\ \mathsf{poly_{br}}$ as the outputs y_1 and z_1 of $\mathsf{mergesort}$ are given as inputs to the predicate merge. In this case, $\beta_{\mathsf{mergesort}}(\cdot) = 0$ for both the $\mathsf{mergesort}$ subgoals and $\beta_{\mathsf{merge}}(\cdot) = 1$ for the second declaration of $\mathsf{mergesort}$. There are also two dependence paths of $\mathsf{length} = 1$ from $\mathsf{mergesort}$ to merge. Thus, this conditions in Figure 4.7 require that merge is non-size increasing and $\#(y) + \#(z) \leq \#(\mathsf{cons}\ x\ xs)$. This follows from split being non-size increasing.$\square$

4.3.5 Decidability

The formal deductive systems presented in Figures 4.3, 4.5 and 4.7 are terminating if the side conditions can be proved or disproved. These side conditions are simply

$$\frac{}{\mathcal{F} \vDash \top}\ \mathsf{g\_True} \qquad \frac{D \in \mathcal{F} \quad \mathcal{F} \vDash D \gg P}{\mathcal{F} \vDash P}\ \mathsf{g\_Atom} \qquad \frac{\mathcal{F}, D \vDash G}{\mathcal{F} \vDash D \supset G}\ \mathsf{g\_Imp}$$

$$\frac{c \ \mathsf{new} \quad \mathcal{F} \vDash [c/x]G}{\mathcal{F} \vDash \forall x : A.G}\ \mathsf{g\_Forall}$$

$$\frac{}{\mathcal{F} \vDash P \gg P}\ \mathsf{c\_Atom} \qquad \frac{\mathcal{F} \vDash [M/x]D \gg P}{\mathcal{F} \vDash \exists x : A.D \gg P}\ \mathsf{c\_Exists} \qquad \frac{\mathcal{F} \vDash D \gg P \quad \mathcal{F} \vDash G}{\mathcal{F} \vDash G \supset D \gg P}\ \mathsf{c\_Imp}$$

Figure 4.9: Proof search semantics for the Hereditary Harrop formulas

multi-variable inequalities which depend only on the input variables of the function and output variables of the function calls. We have commented in Section 4.3.2 on some techniques to eliminate output variables in the conditions. After we have removed all the output variables, we simply need to check that the resulting expression is a polynomial and that the inequality holds. Since the expression is defined over positive integer variables and coefficients, these conditions can be checked easily in modern theorem provers. Therefore, these deductive systems are decidable.

4.4 Extending to Hereditary Harrop Formulas

Hereditary Harrop formulas [33, 51] which allow embedded implications by extending Horn goals G as shown below.

$$\begin{array}{llll} \textit{Goals} & G & ::= & \top \mid P \mid \forall x : A.G \mid D \supset G \\ \textit{Clauses} & D & ::= & G \supset D \mid \exists x : A.D \mid P \end{array}$$

The proof search semantics are extended as shown in Figure 4.9. The embedded implication is operationally interpreted as extending the logic program dynamically during proof-search. Thus, a logic program with Hereditary Harrop formulas is polynomial time if we can ensure that all embedded implications satisfy the polynomial time conditions that we have presented so far.

Example 4.4.1 (β-redexes)**.** Since the arguments to predicates P have to be in canonical form, it is not possible to represent functions such as eval which simplify a term in lambda-calculus to its β-normal form.

$$\begin{aligned}
&\mathsf{eval}\ (\mathsf{lam}\ E)\ (\mathsf{lam}\ E) \subset \top,\\
&\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ V \subset \mathsf{eval}\ E_1\ (\mathsf{lam}\ E_1') \subset \mathsf{eval}\ E_2\ V_2 \subset \mathsf{eval}\ (E_1'\ V_2)\ V
\end{aligned}$$

However, such predicates can be represented by defining a predicate $\mathsf{subst}^{A,B}$: $(A \to B) \to A \to B$ which performs the substitution explicitly and computes the canonical form. For example, if $A = B = \mathsf{exp}$ then $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$ (written as subst^1 for clarity) is given by

$$\begin{aligned}
&\mathsf{subst}^1(\lambda x.x, V; V) \subset \top,\\
&\mathsf{subst}^1(\lambda x.\mathsf{app}\ (E_1 x)\ (E_2 x), V; (\mathsf{app}\ (E_1')\ (E_2')))\\
&\qquad \subset \mathsf{subst}^1(\lambda x.(E_1 x), V; E_1') \subset \mathsf{subst}^1(\lambda x.(E_2 x), V; E_2'),\\
&\mathsf{subst}^1(\lambda x.\mathsf{lam}\ (\lambda y.(E\ x\ y))), V; \mathsf{lam}\ (\lambda y.(E' y)))\\
&\qquad \subset (\forall y : \mathsf{exp}.\mathsf{subst}^1(\lambda x.y, V; y) \supset \mathsf{subst}^1(\lambda x.(E\ x\ y), V; (E'\ y)))
\end{aligned}$$

In this case, we observe that for logic program $\mathcal{F}$ corresponding to $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$, $\vdash_{\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}} \mathcal{F}\ \mathsf{poly_b}$ because the first declaration is non-recursive, $\sum_{i=1}^{2} \#(\lambda x.(E_i x)) < \#(\lambda x.\mathsf{app}\ (E_1 x)\ (E_2 x))$ in the second declaration, and the embedded implication in the third declaration in non-recursive.

On the other hand, when $A = \mathsf{exp} \to \mathsf{exp}$ and $B = \mathsf{exp}$ then $\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}$

(written as subst^2 for clarity) is given by

$$\begin{aligned}
&\mathsf{subst}^2(\lambda f.f, V; V) \subset \top,\\
&\mathsf{subst}^2(\lambda f.(\mathsf{app}\ (E_1\ f)\ (E_2\ f)), V; \mathsf{app}\ E_1'\ E_2')\\
&\qquad \subset \mathsf{subst}^2(\lambda f.(E_1 f), V; E_1') \subset \mathsf{subst}^2(\lambda f.(E_2 f), V; E_2'),\\
&\mathsf{subst}^2(\lambda f.\mathsf{lam}\ \lambda y.(E\ f\ y), V; \mathsf{lam}\ \lambda y.(E'\ y))\\
&\qquad \subset (\forall y : \mathsf{exp}.\mathsf{subst}^2(\lambda f.y, V; y) \supset \mathsf{subst}^2(\lambda f.(E\ f\ y), V; (E'\ y)),\\
&\mathsf{subst}^2(\lambda f.f\ (E f), V; E'')\\
&\qquad \subset \mathsf{subst}^2(\lambda f.E\ f, V; E') \subset \mathsf{subst}^1(\lambda x.V x, E'; E'')
\end{aligned}$$

In this case, the first three declarations satisfy the polynomial time conditions we have described so far. In the fourth declaration, output term E' from the recursive call $\mathsf{subst}^{\mathsf{exp}\to\mathsf{exp},\mathsf{exp}}$ is provided as input to $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$. It is easy to see that Stage 1 conditions do not hold for this case because, it is not possible to determine the run time of $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$ as we do not know the size of its input E'. Stage 2 conditions do not hold either because, $\mathsf{subst}^{\mathsf{exp},\mathsf{exp}}$ is a size-increasing function. Now the $\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ V$ is changed to

$$\begin{aligned}
&\mathsf{eval}\ (\mathsf{app}\ E_1\ E_2)\ V\\
&\ \subset \mathsf{eval}\ E_1\ (\mathsf{lam}\ E_1') \subset \mathsf{eval}\ E_2\ V_2 \subset \mathsf{subst}^{A,\mathsf{exp}}(E_1', V_2; E_1'') \subset \mathsf{eval}\ (E_1''\ V)
\end{aligned}$$

where an appropriate $\mathsf{subst}^{A,\mathsf{exp}}$ is chosen.

Therefore, when $A = \mathsf{exp}$ we know that β-reduction is a polynomial time operation, but when A is a higher-order type, our conditions can no longer guarantee that β-reduction is in polynomial time.$\Box$

Example 4.4.2 (Combinators cont'd)**.** Recall the bracket abstraction algorithm from Example 4.3.1 that is used in the conversion from λ-expressions into combinators. We follow standard practice and define a new type exp together with the two constructors app of type $\mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$ and lam of type $(\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp}$. Using our syntax, extend the program $\mathcal{F}$ from Example 4.3.1 to a program $\mathcal{F}'$ by the following new declarations.

$$
\begin{aligned}
&\mathsf{convert}(\mathsf{app}\ E_1\ E_2; \mathsf{MP}\ C_1\ C_2) \subset \mathsf{convert}(E_1; C_1) \subset \mathsf{convert}(E_2; C_2),\\
&\mathsf{convert}(\mathsf{lam}\ E); D)\\
&\quad \subset \ \ (\forall x : \mathsf{exp}.\, \forall y : \mathsf{comb}.\, \mathsf{ba}(\lambda z : \mathsf{comb}.\, y; \mathsf{MP}\ \mathsf{K}\ y)\\
&\qquad \supset \mathsf{convert}(y; z) \supset \mathsf{convert}\ (E\ x; C\ y))\\
&\quad \subset \mathsf{ba}\ (\lambda y : \mathsf{comb}.\, C\ y; D)
\end{aligned}
$$

We observe that $\vdash_{\mathsf{convert}} \mathcal{F}'\ \ \mathsf{poly}_{\mathsf{br}}$ because the first declaration satisfies

$$\sum_{i=1}^{2} \#(E_i) < \#(\mathsf{app}\ E_1\ E_2)$$

and each embedded implication in the second is non-recursive.

Furthermore $\#(E\ x) < \#(\mathsf{lam}\ E)$ because E is applied to a parameter x (and not an arbitrary term). In addition, $\vdash_{\mathsf{ba}} \mathcal{F}\ \mathsf{nsi}$ by rule $\mathsf{nsi\_Atom}_1$ where $\alpha(\cdot) = 0$ and $\beta_{\mathsf{ba}}(\cdot) = 1$ for the two recursive calls, and hence the dynamic extension of the bracket abstraction algorithm ba is non-size increasing.$\square$

4.5 Extending to Logical framework LF

The operational semantics of LF is essentially that of hereditary Harrop formulas extended with a dependently-typed term algebra.

For a dependently-typed term algebra, we use a size function similar to that for simply-typed terms, i.e. size of a term is number of variables and constants in the term. This size function directly corresponds to the simplest representation of the terms within the proof search engine. As noted by Necula and Lee [57], and Reed [63], dependently-typed terms have a high degree of redundancy. Significant size reductions can be achieved by identifying and eliminating redundant sub-terms in LF. A size function which does not take into account sizes of such redundant terms can also be used. In such cases, we should ensure that the algorithm that reconstructs the redundant terms should have running time independent of the size of the terms reconstructed.

4.5.1 Polynomial-time reductions

In Section 2.3, we have given LF reductions of several NP-complete problems. We shall now show that a polynomial-time checker based on the results in this chapter can identify those reductions.

Reduction from SAT to 3-SAT

The reduction is given in Figure 2.5 (page 22). The predicate $\mathsf{literal}$ is clearly polynomial-time as the size of the boolean formula strictly decreases in the recursive call lnew. Moreover, the β-substitution $F\ v$, the variable v is first-order of type v. Thus, the size of $F\ v$ is $\#(F)$ as the β-substitution is simply renaming the old bound variable to v.

For the main reduction described by conv, the only recursive clauses are $\mathsf{conv}\wedge$ and $\mathsf{conv_n}$. The clause $\mathsf{conv}\wedge$ is polynomial-time as $\#(F_1) + \#(F_2) < \#(F_1 \wedge F_2) = 1+\#(F_1)+\#(F_2)$. In the clause $\mathsf{conv_n}$, size of the initial input is $\#(F_1 \vee F_2 \vee F_3 \vee F) = 3+\#(F_1)+\#(F_2)+\#(F_3)+\#(F)$. And the input to the recursive call is $\#(((\mathsf{neg}\ v) \vee F_3 \vee F)) = 4 + \#(F_3) + \#(F)$. Since, F_1 and F_2 are non-trivial boolean formulas, $\#(F_1) \geq 0$ and $\#(F_2) \geq 0$. Hence, $\#(F_1 \vee F_2 \vee F_3 \vee F) \geq \#(((\mathsf{neg}\ v) \vee F_3 \vee F))$. The other non-recursive functions are clearly polynomial-time in the size of the original input.

Reduction from VERTEX COVER to FEEDBACK ARC SET

The reduction from VERTEX COVER to FEEDBACK ARC SET is given in Figure 2.6 (page 23). This reduction has embedded implication in the form of the clause relate $u\ v\ w$. This clause is non-recursive and so is clearly within polynomial-time. The recursive clauses are conv1 and conv2. In the case of conv2, the size of input in the recursive call $\#(G)$ is clearly less than the size of the original input $\#(\text{newe}\ \lambda e.G) = 1 + \#(G)$. For the clause conv1, we would like to note that size of $G\ u$ is $\#(G)$, as it represents a first-order β-substitution.

Reduction from DIRECTED HAMILTON CIRCUIT to UNDIRECTED HAMILTON CIRCUIT

The reduction from DIRECTED HAMILTON CIRCUIT to UNDIRECTED HAMILTON CIRCUIT is given in Figure 2.7 (page 24). The analysis of this reduction is similar to that of reduction from VERTEX COVER to FEEDBACK ARC SET. The embedded implication introduced is non-recursive and the size of the input to the recursive calls in the clauses conv1 and conv2 is less than the original input.

4.6 Extending to linear Logical framework LLF

Linear logical framework LLF extends LF with connectives from linear logic and provides appropriate operational semantics for these connectives. The operational semantics given in Figure 2.9 highlight the crucial role played by linear assumptions in the context Δ. These linear assumptions play the role of resources which have to be consumed eventually. Since LLF is an extension of LF, the polynomial-time results presented so far extend to LLF.

However identifying more interesting cases would require us to develop additional criteria. The linear embedded implication in LLF is really a postponed computation that has to be executed at some point in future (unless goal $\top$ is proved in the program). This produces an interesting change in the polynomial-time results we have discussed so far. If a clause has linear subgoals, then the linear context needs to contain corresponding clauses that unify with that subgoal. Moreover, those clauses are used up during the proof search. Thus, the number of the linear assumptions initially present also restrict the computational complexity. Hence, a program can be polynomial-time computable even if it may not satisfy our criteria that the sum of the sizes of the input to recursive calls is less than the original input. Such programs have linear subgoals and do not add more clauses of the same type that are used for subgoal evaluation. The reduction from 3-SAT to CHROMATIC that we will discuss below has such clauses. They do not satisfy our criterion but are yet polynomial-time computable. Of course, if a program does satisfy the polynomial-time criterion we have developed, then we need not check these issues.

Moreover, if a clause has a subgoal of the form $G_1 \& G_2$, this subgoal is similar to having two separate goals G_1 and G_2 and the polynomial-time results need to take this into account.

It is non-trivial to incorporate these ideas into our polynomial-time criterion. In fact, determining how a clause changes its linear context is a crucial issue in extending our polynomial-time results to LLF. We will not develop this formally here and leave it for future work.

$$
\begin{aligned}
\mathsf{v2c\_v} \quad : \quad & (\mathsf{var}\ U \multimap \mathsf{vars2clique}\ (G1 + G2 + G3 + G4)) \\
& \leftarrow \mathsf{vars2clique}\ G1\ \&\ \mathsf{connectX}\ V\ G_2 \\
& \quad\ \&\ \mathsf{connectX}\ V'\ G_3\ \&\ \mathsf{connectV}\ X\ G_4 \\
& \leftarrow \mathsf{relate}\ U\ V\ V'\ X. \\
\mathsf{clique\_v} \quad : \quad & (\mathsf{var}\ U \multimap \mathsf{clique}\ (G + G')) \\
& \leftarrow \mathsf{clique}\ G\ \&\ \mathsf{connectX}\ X\ G' \\
& \leftarrow \mathsf{relate}\ U\ \_\ \_\ X.
\end{aligned}
$$

Figure 4.10: Clauses from encoding of clique and vars2clique in the reduction from 3-SAT to CHROMATIC

Reduction from 3-SAT to CHROMATIC

This reduction is given in Figures 2.14–2.19. In Figure 2.19, the arguments C and C' are given simply for proving properties about the function later and need not be considered for the purposes of complexity analysis. Similarly, the continuation K plays no role in recursion and needs to be ignored for the moment as well. In the example, K accumulates the boolean clauses F which are used later. Thus, the function is syntactically similar to *tail recursive* functions and K is in fact intermediate computation and not an input. The criteria we have developed bunch all the input arguments together and will not be able to detect this difference. However, a variant where *recursive* and *non-recursive* inputs are distinguished will succeed in identifying this function. Alternatively, we could have put the boolean clauses F in the linear context and accessed them later instead of gathering them in a continuation. That implementation would also be detected by our criterion.

Of the remaining functions, conv′, conv″, conv‴ (Figures 2.16–2.18) satisfy the property that size of input strictly decreases during recursion and hence they can be shown to be polynomial-time computable.

In the functions clique and vars2clique (Figures 2.14 and 2.15), the size of inputs also decreases during recursive calls. However, in the case of vars2clique, output U of one auxiliary function var is input to relate U and output V' and X of relate are inputs to auxiliary functions connectX and connectV respectively. The case with clique is similar. The relevant clauses from clique and vars2clique are repeated in Figure 4.10. It can be checked that the clauses var and relate are non-recursive and hence are single step computations. It can also be shown that the size of their respective outputs is 1.

The functions connectX and connectV fail the polynomial-time criteria that we have developed as the size of the input never decreases during the recursive calls. However, the point to note here is that these functions have a linear subgoal var U. Since these clauses were introduced only in the conv function, their number is bounded by the size of the original input. Moreover, the functions connectX and connectV do not introduce any linear assumptions. Hence, even though the size of the input never decreases, the functions connectX and connectV are polynomial time functions. We have not developed these ideas formally in this dissertation, but leave it for future work.

Thus, this example illustrates the potential for extending the criteria we have developed for identifying polynomial-time computations for complex reductions in LLF using linear contexts.

Chapter 5

Complexity analysis of forward chaining logic programs

In this chapter, we shall develop criteria for identifying polynomial time forward-chaining logic programs. The criteria are based on properties of computation traces of forward-chaining logic programs. The results will be presented for a forward-chaining variant of the Horn fragment with simply-typed λ-calculus terms. We will also describe informally how these results can be applied to a full-fledged system like concurrent logical framework (CLF).

5.1 Related work

Givan and McAllester [28] have defined the concept of *local* rule sets and shown that forward-chaining in *local* rule sets always terminates in polynomial-time. While they show that *locality* is undecidable in general, several subclasses of locality such as *bounded locality* are decidable.

Ganzinger and McAllester [24, 25] have provided a relationship between running time of a forward-chaining logic program and the number of *prefix firings* by the clauses of logic programs. Informally, *prefix firings* of a clause is the total number of times that a clause can be selected for execution by the forward-chaining engine.

We believe that the results that we present in this chapter are closely related to both the notions of *locality* and *prefix firings* of a clause. However, we will not explore these connections in this dissertation and leave it for future work.

5.2 Forward-chaining fragment of CLF

In Chapter 3, we gave a complete description of Concurrent Logical Framework (CLF). We will focus on a restricted forward-chaining component for the development of our complexity criteria given below.

$$
\begin{array}{lccl}
\textit{Programs} & \mathcal{F} & ::= & \bullet \mid D, \mathcal{P} \\
\textit{Antecedents} & E & ::= & \bullet \mid P; E \mid !P; E \mid \exists x : A.E \\
\textit{Clauses} & D & ::= & E \Rightarrow S \\
\textit{Assertion} & P & ::= & a \mid P N \\
\textit{Conclusion} & S & ::= & S_1 \otimes S_2 \mid \top \mid \exists u : A.S \mid !P \mid P
\end{array}
$$

The clauses are denoted by D and the antecedents are given in E. We distinguish between intuitionistic $!P$ and linear antecedents P. The latter can be used only once while the former can be used multiple times. This is a forward-chaining fragment of CLF is essentially the Horn fragment with insertion and deletion. We have modeled the insertion and deletion aspect of the operational semantics using linear logic.

The assertions P are generated using the type constructors a and terms N. For the moment, we will restrict the terms N and term variables x to the simply-typed λ-calculus that we used in the previous chapter and also disallow non-canonical terms. This system is quite expressive enough to represent the reductions between NP-complete problems.

For example, an implementation of Fibonacci numbers in this framework would be given as $\mathcal{F} = \mathsf{fib}\ (\mathsf{s}\ (\mathsf{s}\ N)) \Rightarrow \mathsf{fib}\ (\mathsf{s}\ N) \otimes \mathsf{fib}\ N, \mathsf{fib}\ \mathsf{z} \Rightarrow \mathsf{val}\ \mathsf{z}, \mathsf{fib}\ \mathsf{s}\ \mathsf{z} \Rightarrow \mathsf{val}\ \mathsf{z}, \mathsf{val}\ \mathsf{z}; \mathsf{sum}\ V \Rightarrow \mathsf{sum}\ (\mathsf{s}\ V)$. The linear initial context is $\mathsf{fib}\ N, \mathsf{sum}\ \mathsf{z}$ and the final linear context is simply $\mathsf{sum}\ M$ – where M is the computed value of the function at N.

5.2.1 Function computation through forward-chaining

Our primary interest in studying forward-chaining logic programs is to be able to represent functions – in particular reductions between NP-complete problems. The initial input is given in the intuitionistic context Γ and the linear context Δ. The final contexts contain the output produced by the logic program. Since we are interested in function computation, we would like to ensure that all executions of the logic program on the same input yield the same output. Hence, we will restrict ourselves to programs without any non-determinism, i.e. there cannot be two rules which can be successfully applied at stage during forward-chaining computation or even a single rule cannot be applied in two different ways to produce different outcomes.

The selected program clauses modify the database according to the operational semantics described in Figure 5.1. The forward-chaining engine constructs a derivation of the judgment $\mathcal{F} \vDash \Gamma, \Delta$. The engine selects program clauses from the program $\mathcal{F}$ until *saturation* is reached (See section 3.2.2). The judgment $\mathcal{F} \vDash E \gg \Gamma, \Delta$ is provable if the assertions in E are provable under the contexts Γ and Δ, i.e. the database of known assertions. The linear context Δ needs to be empty when there are no assertions left to prove. The judgment $\mathcal{F} \vDash S > \Gamma, \Delta$ decomposes the conclusion S and adds it to the contexts Γ and Δ. The definition of split is given in

$$\frac{\mathcal{F}, D \vDash D > \Gamma, \Delta}{\mathcal{F}, D \vDash \Gamma, \Delta}\ \mathsf{CLAUSE} \qquad \frac{\mathcal{F} \vDash E \gg \Gamma, \Delta_1 \quad \mathcal{F} \vDash S > \Gamma, \Delta_2}{\mathcal{F} \vDash E \Rightarrow S > \Gamma, \Delta_1, \Delta_2}\ \mathsf{ANTCDNT}$$

$$\frac{}{\mathcal{F} \vDash \bullet \gg \Gamma, \cdot}\ \mathsf{EMPTY} \qquad \frac{\mathcal{F} \vDash [M/x]E \gg \Gamma, \Delta}{\mathcal{F} \vDash \exists x : A.E \gg \Gamma, \Delta}\ \mathsf{EXISTS}$$

$$\frac{P \doteq Q \quad \mathcal{F} \vDash E \gg \Gamma, u : Q, \Delta}{\mathcal{F} \vDash !P; E \gg \Gamma, u : Q, \Delta}\ \mathsf{I\text{-}ASSRT} \qquad \frac{P \doteq Q \quad \mathcal{F} \vDash E \gg \Gamma, \Delta}{\mathcal{F} \vDash P; E \gg \Gamma, \Delta, u : Q}\ \mathsf{L\text{-}ASSERT}$$

$$\frac{\mathcal{F} \vDash \Gamma, \Gamma', \Delta, \Delta'}{\mathcal{F} \vDash S > \Gamma, \Delta}\ \mathsf{ATOM} \qquad \text{(where } (\Gamma; \Delta') = \mathsf{split}(S))$$

Figure 5.1: Operational Semantics of the forward-chaining fragment (See Figure 5.2 for definition of split)

$$\begin{array}{rcl}
\mathsf{split}(\top) & = & (\cdot; \cdot) \\
\mathsf{split}(!P) & = & (D; \cdot) \\
\mathsf{split}(P) & = & (\cdot; D) \\
\\
\mathsf{split}(S_{D_1} \otimes S_{D_2}) & = & (\Gamma_1, \Gamma_2; \Delta_1, \Delta_2) \\
 & & \mathsf{where}\ (\Gamma_1, \Delta_1) = \mathsf{split}(S_{D_1}) \\
 & & \mathsf{and}\ (\Gamma_2, \Delta_2) = \mathsf{split}(S_{D_2}) \\
\mathsf{split}(\exists x : A.S_D) & = & \Gamma, c : A, \Delta \\
 & & \mathsf{where}\ \Gamma; \Delta = \mathsf{split}([\mathsf{c}/x]S_D)
\end{array}$$

Figure 5.2: Definition of $\mathsf{split}(\cdot)$

Figure 5.2.

For the sake of our analysis, we will assume that the engine will predict the correct instantiations of the universally quantified variable in the rule EXISTS. In an actual implementation, however, the variable would employ logic variables that would be instantiated by unification in the rules I-ASSRT and L-ASSRT. If unification fails, the processing will be backtracked to the point where the failed clause was selected by the corresponding CLAUSE rule.

In the next section, we shall show that each rule given in Figure 5.1 can be implemented in a constant number of steps.

Definition 5.2.1 (Size of the forward-chaining derivation)**.** Given a logic program $\mathcal{F}$ and a derivation $\mathcal{D} :: \mathcal{F} \vDash \Gamma, \Delta$, we define size of $\mathcal{D}$, $\mathsf{sz}(\mathcal{D})$ as the number of rules in $\mathcal{D}$.

5.2.2 Size of contexts

The size of a context (intuitionistic or linear), denoted by $\mathsf{sz}(\Gamma)$ or $\mathsf{sz}(\Delta)$, is defined as the total number of symbols present in all the assertions in the context (excluding the separator symbols like ,). The sizes of simply-typed λ terms is given in Figure 4.2. We have seen that the boolean formula $(u_1 \vee \bar{u}_2 \vee u_3 \vee \bar{u}_4) \wedge (u_2 \vee u_3 \vee u_4)$ is represented by the context $u_1 : \mathsf{variable}, u_2 : \mathsf{variable}, u_3 : \mathsf{variable}, u_4 : \mathsf{variable}, c_1 : \mathsf{clause}, c_2 : \mathsf{clause}; n_1 : \mathsf{ndisjunct}\ c_1\ (\mathsf{pos}\ u_1), n_2 : \mathsf{ndisjunct}\ c_1\ (\mathsf{neg}\ u_2), n_3 : \mathsf{ndisjunct}\ c_1\ (\mathsf{pos}\ u_3), n_4 : \mathsf{ndisjunct}\ c_1\ (\mathsf{neg}\ u_4), n_5 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_2), n_6 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_3), n_7 : \mathsf{ndisjunct}\ c_2\ (\mathsf{pos}\ u_4)$. The length of the intuitionistic context is 6 and the length of the linear context is 28. Note that we are ignoring the u_i's when computing the size of the intuitionistic context and the n_i's when computing the size of the linear context.

5.2.3 Translation to a random access machine (RAM)

We would restrict ourselves to a subset of logic programs which satisfies the following properties.

1. deterministic
2. and the time required to solve the individual unification problem is independent of the contexts.

We have already discussed the need for considering only deterministic logic programs. The case of limiting ourselves to only those programs where every unique unification problem is a constant is similar to that for the backward-chaining case – we would like to show that every forward-chaining step can be implemented on a RAM in constant time. Thus, we require that all patterns are *higher-order patterns* and all assertions P do not have multiple occurrences of a variable. However, the rules $\mathsf{I\text{-}ASSRT}$ and $\mathsf{L\text{-}ASSRT}$ cannot be implemented in constant number of steps (depending on the size of the logic program only) as they require selection over the intuitionistic context Γ and the linear context Δ.

Oracle based forward-chaining

However, if we are given an oracle which returns the correct the set of clauses for every instantiation of $\mathsf{I\text{-}ASSRT}$ and $\mathsf{L\text{-}ASSRT}$, these rules can be implemented in a constant number of steps. It is sufficient that the oracle return the correct instantiation in time that is polynomial in the size of the initial contexts.

Theorem 5.2.1. *Given a logic program $\mathcal{F}$ and initial intuitionistic and linear contexts Γ and Δ satisfying the conditions given above and an oracle as described above. If there exists a derivation $\mathcal{D} :: \mathcal{F} \vDash \Gamma, \Delta$, then*

1. *The intuitionistic and linear contexts can be represented on a RAM in size proportional to the size of those contexts.*

2. *The corresponding forward-chaining procedure can be implemented on a random access machine in time proportional to $\mathsf{sz}(\mathcal{D})(|\mathcal{F}|+1+p(\mathsf{sz}(\Gamma)+\mathsf{sz}(\Delta)))$, where $|\mathcal{F}|$ is the number of clauses in the logic program $\mathcal{F}$ and $p(\cdot)$ is a polynomial.*

Proof. The contexts can be stored on a RAM by simply storing the assertions. The total size of this input is proportional to the size of the contexts.

We shall now show that every rule in Figure 5.1 can be implemented in a constant number of steps, i.e. depends only on the size of the patterns in the logic program

$\mathcal{F}$.

For the rules EMPTY and EXISTS, it is clear that the implementation can be done in constant number of steps. Implementing ATOM involves disaggregating the conclusion S. This process takes time proportional to the maximum size of the conclusion in the program $\mathcal{F}$ – a constant. Implementing CLAUSE involves selecting a clause from the list of clauses in the program $\mathcal{F}$.

The implementation of ANTCDNT does not require us to split the linear context, but we pass the current linear context to the first judgment of the $\mathcal{F} \vDash E \gg \Gamma, \Delta_1$. This judgment succeeds only if the linear context is empty. Hence, we can pass the remaining linear context to the second judgment $\mathcal{F} \vDash S > \Gamma, \Delta_2$. Thus, we never need to non-deterministically split the linear context Δ in implementing this rule.

During the implementation of EXISTS, we substitute the existentially quantified variable by logic variables which are unified in the rules I-ASSRT and L-ASSRT. Unification is guaranteed to be decidable and depends only on the size of the program clauses. Moreover, since all assertions in the contexts are *ground*, unification is a series of pattern matching operations and hence it runs in time polynomial in the size of the pattern (a constant).

If the unification fails at any point, the forward-chaining is backtracked to the most recent CLAUSE rule and another clause is selected for analysis. The number of rules backtracked is bounded by the size of the largest clause in the program $\mathcal{F}$. Since there are at most $|\mathcal{F}|$ clauses, at most $|\mathcal{F}|$ clauses are selected at any CLAUSE rule. Thus, given a derivation $\mathcal{D}$, there are at most $|\mathcal{F}|$ CLAUSE rules unaccounted for every CLAUSE rule.

Moreover, the oracle returns the correct set of clauses in polynomial-time as a function of the initial contexts.

Hence, the forward-chaining process can be implemented in time proportional to

$\mathsf{sz}(\mathcal{D})(|\mathcal{F}| + 1 + p(\mathsf{sz}(\Gamma) + \mathsf{sz}(\Delta)))$ given the oracle. $\square$

5.3 Classification of forward-chaining programs

In this section, we will develop a criterion for identifying polynomial time and non-polynomial time forward-chaining logic programs. The criteria are based on distinguishing the program clauses into two distinct classes, viz. *inductive* clauses and *non-inductive* clauses. In general, *inductive* clauses cause a logic program have super-polynomial time execution time, unless it can be additionally shown that execution of those clauses reduces the size of the context. We will describe these ideas in more detail below.

5.3.1 Inductive and non-inductive clauses

Consider the operational semantics of forward-chaining presented earlier. The initial intuitionistic and linear contexts contain the initial input. The forward-chaining engine selects clauses, verifies that the antecedents hold under the context and adds the conclusions to the contexts. This process is continued until selection of a clause does not add any new conclusions can be added to the context.

Thus, we can have executions where conclusions produced during execution of a clause are used as antecedents later during execution of another clause.

A *non-inductive* clause D is a clause whose conclusions are never used as antecedents for another execution of the same clause.

On the other hand, an *inductive* clause D can have executions where antecedent during an execution is a conclusion that was produced during a previous execution of the same clause. We shall restrict ourselves to programs with only these two kind of clauses. However, we conjecture that the results would extend to the general case as well.

We would like to note here that specification of the initial context under which the program is to be run determines whether a clause is inductive or not.

For example, consider the logic program for computing a Fibonacci number. It consists of the following four clauses:

$$\begin{aligned}
d_1 &: \mathsf{fib}\ (\mathsf{s}\ (\mathsf{s}\ N)) \Rightarrow \mathsf{fib}\ (\mathsf{s}\ N) \otimes \mathsf{fib}\ N \\
d_2 &: \mathsf{fib}\ \mathsf{z} \Rightarrow \mathsf{val}\ \mathsf{z} \\
d_3 &: \mathsf{fib}\ \mathsf{s}\ \mathsf{z} \Rightarrow \mathsf{val}\ \mathsf{z} \\
d_4 &: \mathsf{val}\ \mathsf{z}; \mathsf{sum}\ V \Rightarrow \mathsf{sum}\ (\mathsf{s}\ V)
\end{aligned}$$

The first and the last program clauses are *inductive*, while the second and the third program clauses are *non-inductive*.

5.3.2 Input to a logic program clause

We define input to a logic program clause as all members of the input context which unify successfully with at least one antecedent of the program clause. The sum of the sizes of those inputs is the sum of the input for the execution of the program clause.

For example, the initial input to the Fibonacci function shown above could just be the context $\mathsf{fib}\ (\mathsf{s}\ (\mathsf{s}\ \mathsf{z}))$. After first execution of d_1, the input context becomes, $\mathsf{fib}\ (\mathsf{s}\ \mathsf{z}), \mathsf{fib}\ \mathsf{z}$. Clearly, the total size of the input has increased, though size of each individual member of the context is smaller.

5.3.3 Criteria for polynomial-time logic programs

A forward-chaining logic program under a well-defined input context computes a polynomial-time function if one of the following two conditions holds for every logic program clause.

We give formal proofs of these conditions in Theorems 5.3.1 and 5.3.2.

- *All clauses are non-inductive.*
- *All clauses are inductive and the size of the input reduces after an execution of the clause.*

We can consider two consecutive executions of a clause as a recursive call. Thus, the condition requiring the size of the input context to decrease in two consecutive executions is similar to our condition for backward-chaining programs described in the previous chapter.

Modified criteria

In reality, there can be *inductive* clauses that terminate in polynomial-time (in the size of the initial input) even when the input size increases after each execution. For example, consider the variant of the clause d_4 from Fibonacci number example:

$$\mathsf{val}\ \mathsf{z}; \mathsf{sum}\ V \Rightarrow \mathsf{sum}\ (\mathsf{s}\ (\mathsf{s}\ (\mathsf{s}\ V)))$$

In this case, the total size of the context increases by 1 units after every execution of the clause. Yet the clause terminates in polynomially many steps in the size of the initial input. We can take into account such clauses if we modify our size condition slightly. Instead of comparing the total size of the input with that of the output, we make the comparison only between types generated from the same type family. If there is a size decrease in at least one type family, the clause is bound to terminate in polynomial time. In the above example, while the size of types generated by sum increases, the types generated by val decrease. The proof of Theorem 5.3.2 will hold in this case as well.

We also conjecture that it is sufficient to ensure that the clauses are *inductive*, but leave a complete formal proof for future work.

For clauses with no linear assertions or conclusions, it may be difficult to make such a claim. In such cases, we may need to develop additional criteria or adapt previous results by Ganzinger and McAllester [24], and Givan and McAllester [28]. However, since we are primarily concerned with representing reductions between NP-complete problems in this dissertation, such cases are very rare.

When every clause of the logic program has at least one linear antecedent, it suffices to focus only on the linear antecedents and conclusions to determine if the clauses are *inductive* or *non-inductive*. Similarly, we need to focus only on the linear antecedents and conclusions to determine the size of the input to the clause. The proof of this condition is very similar to that of the conditions given above and we have sketched it in Theorem 5.3.3.

Theorem 5.3.1 (Non-inductive)**.** *Given a logic program $\mathcal{F}$ with no clauses having empty antecedents, and initial input as intuitionistic context Γ and linear context Δ. If all program clauses D in the logic program $\mathcal{F}$ are non-inductive and there exists a derivation $\mathcal{D} :: \mathcal{F} \vDash \Gamma, \Delta$, then $\mathsf{sz}(\mathcal{D})$ is a polynomial in $\mathsf{sz}(\Gamma)$ and $\mathsf{sz}(\Delta)$.*

Proof. Let $|\mathcal{F}|$ is the number of program clauses in $\mathcal{F}$.

The forward-chaining execution described by the derivation $\mathcal{D}$ can be restructured as a directed graph. Each node of the graph denotes execution of a program clause and there is a directed edge between two nodes when conclusion of a clause is used as antecedent for the second clause. Since all clauses are non-inductive, this graph can be partitioned into $|\mathcal{F}|$ groups $N_0, N_1, \ldots, N_{|\mathcal{F}|-1}$ such that nodes in group N_i use at least one conclusion from group N_{i-1}. The nodes in group N_i can additionally use the initial context or the conclusions from the previous nodes. In effect, this synchronizes the asynchronous forward-chaining execution.

We will try to bound the number of nodes in each group N_i by a polynomial. The main restrictions are the number of program clauses $|\mathcal{F}|$, the fact that a linear assumption can be used only once, and the maximum number of antecedents used or conclusions generated by any program clause is a constant.

Let x_i be the total size of intuitionistic context and y_i be the total size of the linear context before any clause in group N_i is executed. We know that $x_0 = \mathsf{sz}(\Gamma)$ and $y_0 = \mathsf{sz}(\Delta)$.

Since each member of the linear context can be used at most once, there can be at most y_0 nodes using the linear assumptions. Moreover, let k be the maximum number of intuitionistic assumptions in any program clause in $\mathcal{F}$. The number of nodes using intuitionistic assumptions is bounded by $\binom{x_0}{l_1} l_1! + \ldots + \binom{x_0}{l_{|\mathcal{F}|}} l_{|\mathcal{F}|}!$, where $l_i \leq k$ is the number of intuitionistic assumptions in the ith clause. In other words, a clause with l_i intuitionistic assumptions cannot be instantiated more than $\binom{x_0}{l_i} l_i!$ times. Since $\binom{x_0}{l_i} l_i! \leq \frac{x_0^{l_i}}{l_i!} l_i! \leq x_0^k$, this sum can be bounded by $|\mathcal{F}|x_0^k$.

Therefore, $|N_0| \leq |\mathcal{F}|x_0^k + y_0$.

Using a similar argument, we can show that $|N_i| \leq y_i + |\mathcal{F}|x_i^k$.

To determine the bound on x_i and y_i, we need to note that all program clauses add only finitely many more new members to the linear and intuitionistic contexts. Let $C \geq 1$ be the constant such that for any program clause, if x is the size of all the antecedents to a program clause, then Cx is the total size of the conclusions. Thus, $x_i \leq x_{i-1} + C(x_{i-1} + y_{i-1})|N_{i-1}|$ and $y_i \leq y_{i-1} + C(x_{i-1} + y_{i-1})|N_{i-1}|$.

Thus, we can now show that $|N_1|$ is bounded by

$$
\begin{aligned}
|N_1| &\leq |\mathcal{F}|x_1^k + y_1 \\
&\leq |\mathcal{F}|((x_0 + C(x_0 + y_0)|N_0|)^k + y_0 + C(x_0 + y_0)|N_0|
\end{aligned}
$$

Since $|N_0| \leq |\mathcal{F}|x_0^k + y_0$, it can be shown that $|N_1|$ is a polynomial in x_0 and y_0.

We will skip rest of the technical details, but it can be shown that each of the N_i's is a polynomial in x_0 and y_0. The result can be guessed from the observation that x_0 and y_0 always appear as base in any exponentiation in the final term. The total number of clauses used in the computation is bounded by $\sum_{i=0}^{|\mathcal{F}|-1} |N_i|$ and this sum is also a polynomial in x_0 and y_0.

Since the sizes of the contexts x_i and y_i remain a polynomial in the initial size x_0 and y_0, an oracle to find the correct sets of antecedents for every clause can be determined in polynomial time in the size of the initial contexts. Further, each node corresponds to at most a constant number of steps in the derivation $\mathcal{D}$. $\square$

Theorem 5.3.2 (Inductive)**.** *Given a logic program $\mathcal{F}$ with no clauses having empty antecedents, and initial input as intuitionistic context Γ and linear context Δ. If the following conditions hold:*

1. *Every program clause D in the logic program $\mathcal{F}$ is either non-inductive or inductive.*

2. *The size of the input context strictly decreases after an execution of any inductive clause,*

and there exists a derivation $\mathcal{D} :: \mathcal{F} \vDash \Gamma, \Delta$, then $\mathsf{sz}(\mathcal{D})$ is a polynomial in $\mathsf{sz}(\Gamma)$ and $\mathsf{sz}(\Delta)$.

Proof. The proof is similar to that of the previous theorem. We can partition the nodes in the execution graph into groups in a similar manner. However, we will group all consecutive executions of the same clause together. Now we can partition the nodes of this modified execution graph into $|\mathcal{F}|$ groups $N_0, N_1, \ldots, N_{|\mathcal{F}|-1}$. As before, nodes in group N_i use at least one conclusion from group N_{i-1}. In addition, the can use conclusions from previous groups and also conclusions produced by previous executions of same self-inductive clause in their own group.

We will try to bound the number of nodes in each group N_i by a polynomial. Let x_i be the total size of intuitionistic context and y_i be the total size of the linear context before any clause in group N_i is executed. We know that $x_0 = \mathsf{sz}(\Gamma)$ and $y_0 = \mathsf{sz}(\Delta)$.

Using arguments similar to that given in the previous proof, we can show that $|N_i| \leq y_i + |\mathcal{F}|x_i^k$.

The relation between x_i, y_i and their predecessors is slightly different as we have to take into account the multiple executions at the nodes in each group. However, we know that the size of the input strictly decreases after an execution of the clause. Hence, the total number such executions in each group N_i is bounded by $y_i|N_i|$. Thus, we have $x_i \leq x_{i-1} + C(x_{i-1} + y_{i-1})y_{i-1}|N_{i-1}|$ and $y_i \leq y_{i-1} + C(x_{i-1} + y_{i-1})y_{i-1}|N_{i-1}|$. The constant C is the constant factor such that for any program clause, if x is the sum of the sizes of the antecedents then Cx bounds the sum of the sizes of the conclusions. This constant depends solely on the logic program $\mathcal{F}$.

The total number of clauses is given by the sum $\sum_{i=0}^{|\mathcal{F}|-1} y_i|N_i|$ and this sum can be shown to be a polynomial in x_0 and y_0. This sum is the number of clauses executed during the forward chaining computation.

Since the sizes of the contexts x_i and y_i remain a polynomial in the initial size x_0 and y_0, an oracle to find the correct sets of antecedents for every clause can be determined in polynomial time in the size of the initial contexts. The theorem follows because each node corresponds to at most a constant number of steps in the derivation $\mathcal{D}$. $\square$

Theorem 5.3.3 (Linear antecedents)**.** *Given a logic program $\mathcal{F}$ with no clauses having empty antecedents and every clause having at least one linear antecedent,* *and initial input as intuitionistic context* Γ *and linear context* Δ*. If the following conditions hold:*

1. *Every program clause* D *in the logic program* $\mathcal{F}$ *is either non-inductive or inductive when restricted to linear antecedents and conclusions.*

2. *The size of the linear input context strictly decreases after an execution of any inductive clause,*

and there exists a derivation $\mathcal{D} :: \mathcal{F} \vDash \Gamma, \Delta$*, then* $\mathsf{sz}(\mathcal{D})$ *is a polynomial in* $\mathsf{sz}(\Gamma)$ *and* $\mathsf{sz}(\Delta)$*.*

Proof. (*Sketch*) The proof is similar to the proof of Theorems 5.3.2 with the difference that size of each group N_i is simply bounded by the linear context under which it runs. Thus, $N_i \leq y_i$. The rest of the proof is similar. $\square$

5.3.4 Exponential-time logic programs

The case when inductive clauses do not decrease the size of the input in an execution loop (two executions of the same inductive clause) can lead to a non-terminating program. However, consider the example given below.

$$\mathsf{double}\ (\mathsf{s}\ N) \Rightarrow \mathsf{double}\ N \otimes \mathsf{double}\ N$$
$$\mathsf{double}\ \mathsf{z} \Rightarrow \mathsf{val}\ \mathsf{z}$$

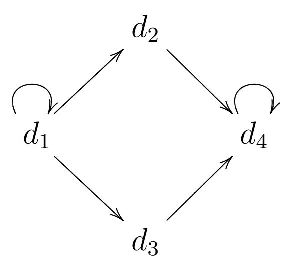
In this case, the first program clause is clearly inductive and increases the size of the input context. But, the program is terminating. Although the total input context size increases, the input size for each clause execution has decreased. Such programs, while terminating, are generally take exponentially many steps to terminate. We will not discuss the exponential case in detail in this dissertation, but leave it for future work.

5.4 Identifying non-inductive and inductive clauses

In this section, we will give a simple algorithm to identify non-inductive and inductive clauses in a logic program.

The algorithm constructs a *dependency graph* where nodes of the graph correspond to the clauses in the logic program and there is a directed edge between two nodes if conclusion of the first clause can be used as an antecedent of the second clause.

Consider the example of Fibonacci number given earlier. The dependency graph corresponding to the logic program is shown below.



Thus, given a dependency graph it is easy to check by doing a breath-first search if the graph has any cycles. If the graph has no cycles, then all the clauses are *non-inductive* and if all the cycles are only loops at nodes then those clauses are *inductive*.

5.4.1 Constructing the dependency graph

The dependency graph can be constructed if for every pair of clauses in the logic program, it is possible to determine whether conclusion of the first can appear as an antecedent in the second. There are two ways to check if this condition holds:

- Ensure that conclusions and antecedents of the two clauses are constructed from two distinct sets of type constructors.

- Unification between conclusions of the first clause and antecedents of the second always fails. If it terminates with flex-flex pairs, there can be an assignment to the logic variables that could cause the conclusions and antecedents to unify.

Since our clauses only have *higher-order patterns*, unification is guaranteed to terminate in polynomial-time. Moreover, the first condition always holds when the second condition holds. However, it is easier to check the first condition than the second. On the other hand, we risk classifying clauses incorrectly if we use the first condition.

5.4.2 Proving that size of the input reduces

The task of proving whether size of input to an *inductive* clause increases after its execution requires us to identify the assertions from the conclusion which may appear in the antecedent of the same clause. We need to take into account sizes of only these assertions. The rest of the assertions are of no consequence for this clause execution. This relationship can be determined by a procedure similar to that used during construction of the dependency graph.

Thus, for the clause d_1 of Fibonacci number, the following inequality needs to be proved or disproved:

$$\#(\mathsf{fib}\ (\mathsf{s}\ (\mathsf{s}\ N))) \geq \#(\mathsf{fib}\ (\mathsf{s}\ N)) + \#(\mathsf{fib}\ N)$$

Such inequalities are similar to those we encountered in the previous chapter while developing complexity analysis criteria for backward-chaining logic programs. They are multi-variable inequalities and they can be easily verified to be true or false by using an implementation of Peano's arithmetic in a standard theorem prover like Twelf.

Formally, these conditions can be represented as shown below.

Definition 5.4.1. Given a forward-chaining logic program clause $E \Rightarrow S$, we define the sets $\mathsf{antecedents}(E)$ and $\mathsf{conclusions}(S)$ as shown below.

$$\begin{array}{rcl}
\mathsf{antecedents}(\bullet) & = & \phi \\
\mathsf{antecedents}(P; E) & = & \{P\} \cup \mathsf{antecedents}(E) \\
\mathsf{antecedents}(!P; E) & = & \mathsf{antecedents}(E) \\
\mathsf{antecedents}(\exists x : A.E) & = & \mathsf{antecedents}([X/x]E)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{conclusions}(P) & = & \{P\} \\
\mathsf{conclusions}(!P) & = & \{P\} \\
\mathsf{conclusions}(\top) & = & \top \\
\mathsf{conclusions}(\exists u : A.S) & = & \mathsf{conclusions}([c/x]S) \\
\mathsf{conclusions}(S_1 \otimes S_2) & = & \mathsf{conclusions}(S_1) \cup \mathsf{conclusions}(S_2)
\end{array}$$

The condition ensuring that the size of the input decrease can now be written as $\sum_{P \in \mathsf{antecedents}(E)} \#(P) \geq \sum_{P \in \mathsf{conclusions}(S)} \#(P)$.

The analysis is similar even when we restrict ourselves to the linear context only or consider inputs and outputs generated by each type constructors separately.

5.5 Examples of NP-complete reductions

Consider the example of reduction from SAT to CLIQUE from Section 3.3.3. We have reproduced the reduction below for convenience.

Although, the logic program is in CLF, it is purely forward-chaining. Hence, it can be converted into our Horn fragment based language. It can be easily checked that every clause is *non-inductive*.

Similarly, all the clauses in the reduction from 3-SAT to CHROMATIC described in Section 3.3.2 are also *non-inductive*.

$$
\begin{array}{rcl}
\mathsf{node} & : & \mathsf{clause} \rightarrow \mathsf{literal} \rightarrow \mathsf{type} \\
\mathsf{nd} & : & \mathsf{clause} \rightarrow \mathsf{literal} \rightarrow \mathsf{vertex} \\
\\
\mathsf{nodes} & : & \mathsf{ndisjunct}\ C\ L \multimap \{!\mathsf{node}\ C\ L\} \\
\mathsf{edges} & : & \mathsf{node}\ C_1\ L_1 \rightarrow \mathsf{node}\ C_2\ L_2 \rightarrow \mathsf{neq}\ C_1\ C_2 \rightarrow \{\mathsf{edge}\ (\mathsf{nd}\ C_1\ L_1)\ (\mathsf{nd}\ C_2\ L_2)\} \\
\mathsf{edge}'_1 & : & \mathsf{edge}\ (\mathsf{nd}\ C_1\ (\mathsf{pos}\ U))\ (\mathsf{nd}\ C_2\ (\mathsf{neg}\ U)) \multimap \{\top\} \\
\mathsf{edge}'_2 & : & \mathsf{edge}\ (\mathsf{nd}\ C_1\ (\mathsf{neg}\ U))\ (\mathsf{nd}\ C_2\ (\mathsf{pos}\ U)) \multimap \{\top\}
\end{array}
$$

5.6 Extending to Horn fragment with priorities

Priorities allow finer control over execution of program clauses before *saturation*. In the operational semantics we have chosen, the program clauses are selected randomly and if all the antecedents can be proven in the context, the rule is executed. With priorities, the higher priority clauses are selected before any lower priority clause is selected. We will assume that all priorities are constants.

Priorities often allow easier representation of forward-chaining algorithms. For example, consider the algorithm for reduction of an instance of SAT to 3-SAT described in Section 3.3.1. Since CLF does not support any well-defined notion of priorities, we need to use a combination of forward-chaining and backward-chaining to represent the algorithm. With priorities, the algorithm would be written in CLF as shown in Figure 5.3 with priorities. The program clause convert has a higher priority than the clauses term1, term2 and term3. Without priorities, this algorithm would be non-deterministic as all four program clauses could be executed when the linear context has more than four ndisjunct assertions.

The criteria we have developed for identifying polynomial-time forward-chaining logic program applies to the case when program clauses have priorities as well. The proofs of Theorems 5.3.1 and 5.3.2 upper bound the total number of program clauses that can be executed under a given initial context. Since introduction of rule prior-

$$\begin{array}{rcl}
\mathsf{convert} & : & \mathsf{ndisjunct}\ C\ L_1; \mathsf{ndisjunct}\ C\ L_2; \mathsf{ndisjunct}\ C\ L_3; \mathsf{ndisjunct}\ C\ L_4 \\
& & \Rightarrow \exists u : \mathsf{variable}.\mathsf{3disjunct}\ L_1\ L_2\ (\mathsf{pos}\ u) \otimes \mathsf{ndisjunct}\ C\ (\mathsf{neg}\ u) \otimes \\
& & \mathsf{ndisjunct}\ C\ L_3 \otimes \mathsf{ndisjunct}\ C\ L_4 \\
\mathsf{term1} & : & \mathsf{ndisjunct}\ C\ L_1; \mathsf{ndisjunct}\ C\ L_2; \mathsf{ndisjunct}\ C\ L_3 \\
& & \Rightarrow \mathsf{3disjunct}\ L_1\ L_2\ L_3 \\
\mathsf{term2} & : & \mathsf{ndisjunct}\ C\ L_1; \mathsf{ndisjunct}\ C\ L_2 \\
& & \Rightarrow \exists u : \mathsf{variable3disjunct}\ L_1\ L_2\ (\mathsf{pos}\ u) \otimes \mathsf{3disjunct}\ L_1\ L_2\ (\mathsf{neg}\ u) \\
\mathsf{term3} & : & \mathsf{ndisjunct}\ C\ L_1 \Rightarrow \exists u_1 : \mathsf{variable}\ u_2 : \mathsf{variable} \\
& & \ \mathsf{3disjunct}\ L_1\ (\mathsf{pos}\ u_1)\ (\mathsf{pos}\ u_2) \otimes \\
& & \ \mathsf{3disjunct}\ L_1\ (\mathsf{pos}\ u_1)\ (\mathsf{neg}\ u_2) \otimes \\
& & \ \mathsf{3disjunct}\ L_1\ (\mathsf{neg}\ u_1)\ (\mathsf{pos}\ u_2) \otimes \\
& & \ \mathsf{3disjunct}\ L_1\ (\mathsf{neg}\ u_1)\ (\mathsf{neg}\ u_2)
\end{array}$$

Figure 5.3: Reduction from SAT to 3-SAT in Horn fragment with rule priorities (all antecedents are linear)

ities can only reduce the number of possible program clauses that can be executed at any stage of the computation, the proofs remain valid.

In the example given above, the program clauses term1, term2 and term3 are non-inductive. On the other hand, the program clause convert is self-inductive. However, the size of the input reduces between two consecutive executions of the clause as the following inequality holds: $\#(\mathsf{ndisjunct}\ C\ L_1) + \#(\mathsf{ndisjunct}\ C\ L_2) + \#(\mathsf{ndisjunct}\ C\ L_3) + \#(\mathsf{ndisjunct}\ C\ L_4) \geq \#(\mathsf{ndisjunct}\ C\ (\mathsf{pos}\ u)) + \#(\mathsf{ndisjunct}\ C\ L_3) + \#(\mathsf{ndisjunct}\ C\ L_4)$.

Similarly, consider the algorithm for checking if a given graph is bipartite. The nodes of a bipartite graph can be partitioned into two subsets A and B such that edges do not connect any pair of nodes in the same subset.

The algorithm in forward-chaining Horn fragment with priorities is given below. The program clause p_1 has the lowest priority and the other program clauses all have higher priorities. The initial linear context consists of the vertices and edges of the graph as assertions of type vertex V and edge $U\ V$. We have also two labels constants

a and b. After termination, the nodes belonging to each subset are identified. If the graph is not bipartite, this program assigns at least one node to both subsets.

$$
\begin{array}{lcl}
p_1 & : & \mathsf{vertex}\ V \Rightarrow !\mathsf{label}\ V\ a \\
p_2 & : & \mathsf{edge}\ U\ V; !\mathsf{label}\ U\ a \Rightarrow !\mathsf{label}\ V\ b \\
p_3 & : & \mathsf{edge}\ U\ V; !\mathsf{label}\ V\ a \Rightarrow !\mathsf{label}\ U\ b \\
p_4 & : & \mathsf{edge}\ U\ V; !\mathsf{label}\ U\ b \Rightarrow !\mathsf{label}\ V\ a \\
p_5 & : & \mathsf{edge}\ U\ V; !\mathsf{label}\ V\ b \Rightarrow !\mathsf{label}\ U\ a
\end{array}
$$

Since all clauses in this program have at least one linear antecedent, we can restrict ourselves to linear context only. Under this restriction, all the program clauses are *non-inductive*. And hence, the program terminates in polynomial-time.

5.7 Extending to Concurrent Logical Framework CLF

The primary challenge in developing complexity analysis criteria involves combining our results for backward-chaining and forward-chaining logic programs into a unified framework. We informally sketch these ideas in this section. We shall illustrate them through the example reduction from SAT to 3-SAT described in detail in Section 3.3.1. It is shown here in Figure 5.4 for convenience.

Since CLF is a combination of forward-chaining and backward-chaining logic programming models, we need to check the corresponding complexity conditions.

1. *Backward chaining conditions:* Every clause in the logic program satisfies the backward-chaining conditions described in the previous chapter. The conditions need to be modified slightly to include the monadic subgoals $\{S\}$. The operational semantics of a monadic subgoal requires forward-chaining until *saturation* followed by backward-chaining computation on the synchronous type

$$
\begin{array}{rcl}
\textsf{convert} & : & \textsf{ndisjunct}\ C\ L_1 \multimap \textsf{ndisjunct}\ C\ L_2 \multimap \textsf{ndisjunct}\ C\ L_3 \multimap \textsf{ndisjunct}\ C\ L_4 \\
& & \multimap \{\exists u : \textsf{variable.3disjunct}\ L_1\ L_2\ (\textsf{pos}\ u) \otimes \textsf{ndisjunct}\ C\ (\textsf{neg}\ u) \otimes \\
& & \textsf{ndisjunct}\ C\ L_3 \otimes \textsf{ndisjunct}\ C\ L_4\} \\
\textsf{terminate} & : & (\textsf{term} \rightarrow \{\top\}) \multimap \textsf{cnfreduction} \\
\textsf{term1} & : & \textsf{term} \rightarrow \textsf{ndisjunct}\ C\ L_1 \multimap \textsf{ndisjunct}\ C\ L_2 \multimap \textsf{ndisjunct}\ C\ L_3 \\
& & \multimap \{\textsf{3disjunct}\ L_1\ L_2\ L_3\} \\
\textsf{term2} & : & \textsf{term} \rightarrow \textsf{ndisjunct}\ C\ L_1 \multimap \textsf{ndisjunct}\ C\ L_2 \\
& & \multimap \{\exists u : \textsf{variable3disjunct}\ L_1\ L_2\ (\textsf{pos}\ u) \otimes \textsf{3disjunct}\ L_1\ L_2\ (\textsf{neg}\ u)\} \\
\textsf{term3} & : & \textsf{term} \rightarrow \textsf{ndisjunct}\ C\ L_1 \multimap \{\exists u_1 : \textsf{variable}\ u_2 : \textsf{variable} \\
& & \textsf{3disjunct}\ L_1\ (\textsf{pos}\ u_1)\ (\textsf{pos}\ u_2) \otimes \\
& & \textsf{3disjunct}\ L_1\ (\textsf{pos}\ u_1)\ (\textsf{neg}\ u_2) \otimes \\
& & \textsf{3disjunct}\ L_1\ (\textsf{neg}\ u_1)\ (\textsf{pos}\ u_2) \otimes \\
& & \textsf{3disjunct}\ L_1\ (\textsf{neg}\ u_1)\ (\textsf{neg}\ u_2)\}
\end{array}
$$

Figure 5.4: Reduction from SAT to 3-SAT

S.

2. *Forward chaining conditions:* The forward-chaining clauses, i.e clauses whose head is a monadic type $\{S\}$ should satisfy the forward-chaining conditions.

We would like to note here that any program clause can only introduce a finite number of assumptions (and other program clauses) during a single backward-chaining step. Similarly, forward-chaining parts of the executions terminate in polynomial time and the final context is a polynomially bounded in the size of the initial context. Thus, the contexts always remain polynomial in the size of the initial input and the initial context.

In the example of the reduction from SAT to CHROMATIC, the forward-chaining rules term1, term2 and term3 are *non-inductive*. The rule convert is *self-inductive*, but the size of the context generated by the type family ndisjunct decreases. The backward-chaining rule terminate is non-recursive and hence trivially satisfies our conditions.

Chapter 6

Conclusion and Future work

We began this dissertation with the goal of developing a formal framework for representing and reasoning about NP-complete problems. The main requirements that we noted as necessary for such a system were:

1. Ability to represent NP-complete problems and problem instances,

2. Ability to represent reductions between NP-complete problems and proofs that the reductions represent polynomial-time algorithms,

3. Ability to represent proofs that the reduction is *correct*, i.e. the reduction maps *Yes* instance of the first problem to *Yes* instance of the second problem and vice-versa.

We chose logical framework LF as a starting point to better understand the strengths and limitations of current systems. Logical framework LF provided with a good set of basic features for representing NP-complete problems and their reductions. The *proofs as programs* paradigm of LF and the Elf language also allowed us to represent the associated proofs.

We encountered two major difficulties in using LF in its original form. First, it is hard to represent reductions in LF that require multiple iterations over some data structure. Second, representing mathematical entities in as LF terms imposes an artificial ordering on them. Due to this limitation, simple operations can have very complex LF encodings.

Linear logical framework LLF addressed some of these difficulties by providing a linear context that allowed some control over how data was accessed. Concurrent Logical Framework CLF, with its forward-chaining semantics and linear contexts turned out to be ideal for representing reduction algorithms.

Logical frameworks currently lack any support for identifying polynomial-time reductions and representing the corresponding proof. In this dissertation, we have focused on this problem in depth and developed syntactic, decidable criteria for identifying polynomial-time algorithms. The criteria, while not complete, are quite intuitive and generally correspond to the most natural way a programmer might choose to represent reductions for NP-complete problems. We have described the results in detail for Horn fragment, and informally described their extensions for the logical framework LF, the linear logical framework LLF and the concurrent logical framework CLF.

The task of representing proofs of correctness is well understood for logical frameworks LF and LLF (although there is not formal implementation of LLF meta-logic). However, the meta-theory of concurrent logical framework CLF has yet to be developed. In fact, we are not aware of any formal reasoning system for forward-chaining logic programs. The development of such a system would be an important step forward in achieving the goal of having a digital library for NP-complete problems.

We would also like to use the logical frameworks LF, LLF to store the proof of the fact that a reduction is a polynomial time reduction. Such a proof would store the deductive systems that we have described in Chapter 4 along with the proofs of side conditions. It is also worth studying if the criteria described in Chapter 5 can be stored in a meta-theoretic framework for CLF.

Most of the NP-complete problems that we have focused in this dissertation have problem instances that are described either using boolean formulas or graphs. While there are literally hundreds of problems that fall within this category, it leaves out many interesting classes of problems. Of the Karp's 21 problems, the problems that we have not considered are SUBSET SUM and JOB SEQUENCING. These problems have conditions involving arithmetic operations. Most theorem provers, including LF and its variants, do not have a natural way to represent these operations. An obvious approach would involve implementing Peano's arithmetic within these theorem provers and using it to express such the arithmetic operations. We have not studied the advantages and limitations of this approach in this dissertation, but it is a fruitful area for future research.

A useful feature that we have not discussed in this dissertation is the ability of the system to automatically or semi-automatically search if a given NP-complete problem and its instances reduce to a well-known NP-complete problem already in the library. Given a description of NP-complete problem, such a search feature would find an algorithm that converts every well-defined instance of some *known* problem to a well-defined instance of the given problem. In addition, it would also have to provide a proof that such the algorithm maps the *Yes* instances of the *known* problem to *Yes* instances of the given problem and vice-versa.

6.1 Applications of static complexity analysis

The syntactic criteria for identifying polynomial-time executable logic programs that we have described in this dissertation can be adapted to solve problems that arise in a variety of environments. We shall described some of them below.

Query optimization in databases: Query optimizers often have to decide between multiple equivalent versions of a given query. A query optimizer that could identify polynomial-time executable queries would be an important development for implementation of query languages with features such as recursion. Deductive databases use *bottom-up* evaluation strategies based on the forward-chaining model we have described here. Considerable work has been done *bottom-up* evaluation strategies and source-to-source transformations that make such *bottom-up* evaluation strategies more efficient [6, 55]. Our results could be applied to improve these results as well.

Type-inference systems: It is essential to have efficient type-inference systems. Therefore, identifying connections between our results and known results on tractability of type-inference systems would be a fruitful area for future research.

Advanced programming environments: The syntactic criteria developed in this dissertation can be adapted to a variety of different functional and logic programming languages. Thus, it is not unrealistic to imagine programming environments that would provide user with real-time assistance in identifying polynomial and super-polynomial programs. The most obvious use of this feature would be in educational environments, but a suitable implementation could also be useful in advanced domain-specific programming environments like *Mathematica*.

Chapter 7

Appendix

A Karp's 21 NP-complete problems

Definition A.1 (SAT)**.** Given a set $U = \{u_1, u_2, \ldots, u_n\}$ of Boolean variables and a conjunctive normal form formula $f = c_1 \wedge c_2 \wedge \ldots c_m$ on Boolean variables such that $c_i = l_{i1} \vee l_{i2} \vee \ldots \vee l_{ik_i}$, $\forall i = 1, \ldots, m$ and $k_i \in \mathbb{Z}$, and $l_{i1}, l_{i2}, \ldots, l_{ik_i} \in U \cup \overline{U}$ where $\overline{U} = \{\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n\}$.

QUESTION: Is there a truth assignment to the Boolean variables such that every clause in f is satisfied?

Definition A.2 (0-1 INTEGER PROGRAMMING)**.** Given a finite set X of pairs $(\bar{x}, b)$, where $\bar{x}$ is an m-tuple of integers and b is an integer, an m-tuple $\bar{c}$ of integers, and an integers B.

QUESTION: Is there an m-tuple $\bar{y}$ of integers such that $\bar{x} \cdot \bar{y} \leq b$ for all $(\bar{x}, b) \in X$ and such that $\bar{c} \cdot \bar{y} \geq B$ (where the dot-product $\bar{u} \cdot \bar{v}$ of two m-tuples $\bar{u} = (u_1, u_2, \ldots, u_m)$ and $\bar{v} = (v_1, v_2, \ldots, v_m)$ is given by $\sum_{i=1}^{m} u_i v_i$)?

Definition A.3 (CLIQUE)**.** Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, and a positive integer K.

QUESTION: Does G have a clique with K vertices?

Definition A.4 (SET PACKING)**.** Given a collection C of finite sets and a positive integer $K \leq |C|$.

QUESTION: Does C contain at least K mutually disjoint sets?

Definition A.5 (VERTEX COVER)**.** Given a graph G and a positive integer $K \leq |V|$.

QUESTION: Is there a *vertex cover* of size K or less for G, that is, a subset $V' \subseteq V$ such that $|V'| \leq K$ and, for each edge $\{u, v\} \in E$, at least one of u and v belongs to V'?

Definition A.6 (SET COVERING)**.** Given a collection C of subsets of a finite set S, and a positive integer $K \leq |C|$.

QUESTION: Does C contain a cover for S of size K or less, i.e. a subset $C' \subset C$ with $|C'| \leq K$ such that every element of S belongs to at least one member of C?

Definition A.7 (FEEDBACK NODE SET)**.** Given a directed graph $G = (V, A)$, and a positive integer $K \leq |V|$.

QUESTION: Is there a subset $V' \subseteq V$ with $|V'| \leq K$ such that V' contains at least one vertex from every directed cycle in G?

Definition A.8 (FEEDBACK ARC SET)**.** Given a directed graph $G = (V, A)$, and a positive integer $K \leq |A|$.

QUESTION: Is there a subset $A' \subseteq A$ with $|A'| \leq K$ such that A' contains at least one arc from every directed cycle in G?

Definition A.9 (DIRECTED HAMILTON CIRCUIT)**.** Given a directed graph G.

QUESTION: Does G have a directed cycle which includes every vertex exactly once?

Definition A.10 (UNDIRECTED HAMILTON CIRCUIT)**.** Given a graph G.

QUESTION: Does G have a cycle which includes every vertex exactly once?

Definition A.11 (3-SAT)**.** Given a set $U = \{u_1, u_2, \ldots, u_n\}$ of Boolean variables and a conjunctive normal form formula $f = c_1 \wedge c_2 \wedge \ldots c_m$ on the Boolean variables in U such that $c_i = l_{i1} \vee l_{i2} \vee l_{i3}, \forall i = 1, \ldots, m$ and $l_{i1}, l_{i2}, l_{i3} \in U \cup \overline{U}$ where $\overline{U} = \{\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n\}$.

QUESTION: Is there a truth assignment to the Boolean variables such that every clause in f is satisfied?

Definition A.12 (CHROMATIC)**.** Given a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, and a positive integer C.

QUESTION: Is G C-colorable, i.e., does there exist a function

$$\chi : V \rightarrow \{1, 2, \ldots, C\}$$

such that $\chi(u) \neq \chi(v)$ whenever $\{u, v\} \in E$?

Definition A.13 (CLIQUE COVER)**.** Given a graph $G = (V, E)$, and a positive integer $K \leq |V|$.

QUESTION: Can the vertices of G be partitioned into $k \leq K$ disjoint sets $V_1, V_2, \ldots, V_k$ such that, for $1 \leq i \leq k$, the subgraph induced by V_i is a complete graph (clique)?

Definition A.14 (EXACT COVER)**.** Given a set $U = \{u_1, \ldots, u_n\}$ and a collection C of subsets of U.

QUESTION: Does C contain an exact cover of U, i.e. a subcollection $C' \subseteq C$ of sets such that every element of U occurs in exactly one member of C'?

Definition A.15 (HITTING SET). Given a collection C of subsets of a finite set S, and a positive integer $K \leq |S|$.

QUESTION: Is there a subset $S' \subseteq S$ with $|S'| \leq K$ such that S' contains at least one element from each subset in C?

Definition A.16 (STEINER TREE). Given a graph $G = (V, E)$, a weight $w(e) \in \mathbb{Z}^+ \cup \{0\}$ for each $e \in E$, a subset $R \subseteq V$, and a positive integer bound B.

QUESTION: Is there a subtree of G that includes all the vertices of R and such that the sum of the weights of the edges in the subtree is no more than B?

Definition A.17 (3-DIMENSIONAL MATCHING). Given a set $M \subseteq W \times X \times Y$, where W, X, and Y are disjoint sets having the same number q of elements.

QUESTION: Does M contain a matching, i.e. a subset $M' \subset M$ such that $|M'| = q$ and no two elements of M' agree in any coordinate?

Definition A.18 (SUBSET SUM – called KNAPSACK by Karp). Given a finite set U, for each $u \in U$ a size $s(u) \in \mathbb{Z}^+$, and a positive integer B.

QUESTION: Is there a subset $U' \subset U$ such that $\sum_{u \in U'} s(u) = B$?

Definition A.19 (JOB SEQUENCING). Given a set T of tasks, for each task $t \in T$ a length $l(t) \in \mathbb{Z}^+$, a weight $w(t) \in \mathbb{Z}^+$, and a deadline $d(t) \in Z^+$, and a positive integer K.

QUESTION: Is there a one-processor schedule σ for T such that the sum of $w(t)$ taken over all $t \in T$ for which $\sigma(t) + l(t) \geq d(t)$, does not exceed K?

Definition A.20 (PARTITION). Given a finite set A and a size $s(a) \in \mathbb{Z}^+$ for each $a \in A$.

QUESTION: Is there a subset $A' \subseteq A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$?

Definition A.21 (MAX-CUT)**.** Given a graph $G = (V, E)$, weight $w(e) \in \mathbb{Z}^+$ for each $e \in E$, and a positive integer K.

QUESTION: Is there a partition of V into disjoint sets V_1 and V_2 such that sum of the weights of the edges from E that have one endpoint in V_1 and one endpoint in V_2 is at least K?

B 3-SAT to CHROMATIC Reduction in Linear LF

The 40 cases mentioned in Figure 2.11 are given below.

$$\frac{\Gamma, (u_1, v_1, v_1', x_1); \Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma, (u_1, v_1, v_1', x_1); \Delta, u_1 \vdash (\textsf{pos}\ u_1) \vee (\textsf{pos}\ u_1) \vee (\textsf{pos}\ u_1) \\ \Rightarrow \textsf{newv}\ c.\textsf{newe}\ e_1 : (c, v_1').G\end{array}}\ c''1.1$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1); \Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma, (u_1, v_1, v_1', x_1); \Delta, u_1 \vdash (\textsf{pos}\ u_1) \vee (\textsf{pos}\ u_1) \vee (\textsf{neg}\ u_1) \\ \Rightarrow \textsf{newv}\ c.G\end{array}}\ c''1.2$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1); \Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma, (u_1, v_1, v_1', x_1); \Delta, u_1 \vdash (\textsf{pos}\ u_1) \vee (\textsf{neg}\ u_1) \vee (\textsf{pos}\ u_1) \\ \Rightarrow \textsf{newv}\ c.G\end{array}}\ c''1.3$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1); \Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma, (u_1, v_1, v_1', x_1); \Delta, u_1 \vdash (\textsf{pos}\ u_1) \vee (\textsf{neg}\ u_1) \vee (\textsf{neg}\ u_1) \\ \Rightarrow \textsf{newv}\ c.G\end{array}}\ c''1.4$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1); \Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma, (u_1, v_1, v_1', x_1); \Delta, u_1 \vdash (\textsf{neg}\ u_1) \vee (\textsf{pos}\ u_1) \vee (\textsf{pos}\ u_1) \\ \Rightarrow \textsf{newv}\ c.G\end{array}}\ c''1.5$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1);\Delta \vdash c \downarrow G}{\Gamma,(u_1,v_1,v_1',x_1);\Delta,u_1 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{pos}\ u_1) \vee (\mathsf{neg}\ u_1)} \ c''1.6$$

$$\Rightarrow \mathsf{newv}\ c.G$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1);\Delta \vdash c \downarrow G}{\Gamma,(u_1,v_1,v_1',x_1);\Delta,u_1 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{neg}\ u_1) \vee (\mathsf{pos}\ u_1)} \ c''1.7$$

$$\Rightarrow \mathsf{newv}\ c.G$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1);\Delta \vdash c \downarrow G}{\Gamma,(u_1,v_1,v_1',x_1);\Delta,u_1 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{neg}\ u_1) \vee (\mathsf{neg}\ u_1)} \ c''1.8$$

$$\Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1 : (c,v_1).G$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta \vdash c \downarrow G}{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2 \vdash (\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_2)} \ c''2.1$$

$$\Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1 : (c,v_1')\ e_2 : (c,v_2').G$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta \vdash c \downarrow G}{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2 \vdash (\mathsf{pos}\ u_1) \vee (\mathsf{pos}\ u_1) \vee (\mathsf{neg}\ u_2)} \ c''2.2$$

$$\Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1 : (c,v_1')\ e_2 : (c,v_2).G$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta \vdash c \downarrow G}{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2 \vdash (\mathsf{pos}\ u_1) \vee (\mathsf{neg}\ u_1) \vee (\mathsf{pos}\ u_2)} \ c''2.3$$

$$\Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_2 : (c,v_2').G$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta \vdash c \downarrow G}{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2 \vdash (\mathsf{pos}\ u_1) \vee (\mathsf{neg}\ u_1) \vee (\mathsf{neg}\ u_2)} \ c''2.4$$

$$\Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_2 : (c,v_2).G$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{neg}\ u_1)\vee(\mathsf{pos}\ u_1)\vee(\mathsf{pos}\ u_2)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_2:(c,v_2').G\end{array}}\ c''2.5$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{neg}\ u_1)\vee(\mathsf{pos}\ u_1)\vee(\mathsf{neg}\ u_2)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_2:(c,v_2).G\end{array}}\ c''2.6$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{neg}\ u_1)\vee(\mathsf{neg}\ u_1)\vee(\mathsf{pos}\ u_2)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1)\ e_2:(c,v_2').G\end{array}}\ c''2.7$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{neg}\ u_1)\vee(\mathsf{neg}\ u_1)\vee(\mathsf{neg}\ u_2)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1)\ e_2:(c,v_2).G\end{array}}\ c''2.8$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{pos}\ u_1)\vee(\mathsf{pos}\ u_2)\vee(\mathsf{pos}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1')\ e_2:(c,v_2').G\end{array}}\ c''3.1$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{pos}\ u_1)\vee(\mathsf{pos}\ u_2)\vee(\mathsf{neg}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_2:(c,v_2').G\end{array}}\ c''3.2$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta \vdash c \downarrow G}{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta, u_1, u_2 \vdash (\mathsf{pos}\ u_1) \vee (\mathsf{neg}\ u_2) \vee (\mathsf{pos}\ u_1) \atop \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1 : (c, v_1')\ e_2 : (c, v_2).G}\ c''3.3$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta \vdash c \downarrow G}{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta, u_1, u_2 \vdash (\mathsf{pos}\ u_1) \vee (\mathsf{neg}\ u_2) \vee (\mathsf{neg}\ u_1) \atop \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_2 : (c, v_2).G}\ c''3.4$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta \vdash c \downarrow G}{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta, u_1, u_2 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{pos}\ u_2) \vee (\mathsf{pos}\ u_1) \atop \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_2 : (c, v_2').G}\ c''3.5$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta \vdash c \downarrow G}{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta, u_1, u_2 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{pos}\ u_2) \vee (\mathsf{neg}\ u_1) \atop \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1 : (c, v_1)\ e_2 : (c, v_2').G}\ c''3.6$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta \vdash c \downarrow G}{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta, u_1, u_2 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{neg}\ u_2) \vee (\mathsf{pos}\ u_1) \atop \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_2 : (c, v_2).G}\ c''3.7$$

$$\frac{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta \vdash c \downarrow G}{\Gamma, (u_1, v_1, v_1', x_1), (u_2, v_2, v_2', x_2); \Delta, u_1, u_2 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{neg}\ u_2) \vee (\mathsf{neg}\ u_1) \atop \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1 : (c, v_1)\ e_2 : (c, v_2).G}\ c''3.8$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{pos}\ u_2)\vee(\mathsf{pos}\ u_1)\vee(\mathsf{pos}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1')\ e_2:(c,v_2').G\end{array}}\ c''4.1$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{pos}\ u_2)\vee(\mathsf{pos}\ u_1)\vee(\mathsf{neg}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_2:(c,v_2').G\end{array}}\ c''4.2$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{pos}\ u_2)\vee(\mathsf{neg}\ u_1)\vee(\mathsf{pos}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_2:(c,v_2').G\end{array}}\ c''4.3$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{pos}\ u_2)\vee(\mathsf{neg}\ u_1)\vee(\mathsf{neg}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1)\ e_2:(c,v_2').G\end{array}}\ c''4.4$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{neg}\ u_2)\vee(\mathsf{pos}\ u_1)\vee(\mathsf{pos}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1')\ e_2:(c,v_2).G\end{array}}\ c''4.5$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{neg}\ u_2)\vee(\mathsf{pos}\ u_1)\vee(\mathsf{neg}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_2:(c,v_2).G\end{array}}\ c''4.6$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{neg}\ u_2)\vee(\mathsf{neg}\ u_1)\vee(\mathsf{pos}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_2:(c,v_2).G\end{array}}\ c''4.7$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2);\Delta,u_1,u_2\vdash(\mathsf{neg}\ u_2)\vee(\mathsf{neg}\ u_1)\vee(\mathsf{neg}\ u_1)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1)\ e_2:(c,v_2).G\end{array}}\ c''4.8$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2),(u_3,v_3,v_3',x_3);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),\ldots,(u_3,v_3,v_3',x_3);\Delta,u_1,u_2,u_3\vdash(\mathsf{pos}\ u_1)\vee(\mathsf{pos}\ u_2)\vee(\mathsf{pos}\ u_3)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1')\ e_2:(c,v_2')\ e_3:(c,v_3').G\end{array}}\ c''5.1$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2),(u_3,v_3,v_3',x_3);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),\ldots,(u_3,v_3,v_3',x_3);\Delta,u_1,u_2,u_3\vdash(\mathsf{pos}\ u_1)\vee(\mathsf{pos}\ u_2)\vee(\mathsf{neg}\ u_3)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1')\ e_2:(c,v_2')\ e_3:(c,v_3).G\end{array}}\ c''5.2$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2),(u_3,v_3,v_3',x_3);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),\ldots,(u_3,v_3,v_3',x_3);\Delta,u_1,u_2,u_3\vdash(\mathsf{pos}\ u_1)\vee(\mathsf{neg}\ u_2)\vee(\mathsf{pos}\ u_3)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1')\ e_2:(c,v_2)\ e_3:(c,v_3').G\end{array}}\ c''5.3$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2),(u_3,v_3,v_3',x_3);\Delta\vdash c\downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),\ldots,(u_3,v_3,v_3',x_3);\Delta,u_1,u_2,u_3\vdash(\mathsf{pos}\ u_1)\vee(\mathsf{neg}\ u_2)\vee(\mathsf{neg}\ u_3)\\ \Rightarrow\mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1')\ e_2:(c,v_2)\ e_3:(c,v_3).G\end{array}}\ c''5.4$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2),(u_3,v_3,v_3',x_3);\Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),\ldots,(u_3,v_3,v_3',x_3);\Delta,u_1,u_2,u_3 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{pos}\ u_2) \vee (\mathsf{pos}\ u_3)\\ \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1)\ e_2:(c,v_2')\ e_3:(c,v_3').G\end{array}}\ c''5.5$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2),(u_3,v_3,v_3',x_3);\Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),\ldots,(u_3,v_3,v_3',x_3);\Delta,u_1,u_2,u_3 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{pos}\ u_2) \vee (\mathsf{neg}\ u_3)\\ \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1)\ e_2:(c,v_2')\ e_3:(c,v_3).G\end{array}}\ c''5.6$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2),(u_3,v_3,v_3',x_3);\Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),\ldots,(u_3,v_3,v_3',x_3);\Delta,u_1,u_2,u_3 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{neg}\ u_2) \vee (\mathsf{pos}\ u_3)\\ \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1)\ e_2:(c,v_2)\ e_3:(c,v_3').G\end{array}}\ c''5.7$$

$$\frac{\Gamma,(u_1,v_1,v_1',x_1),(u_2,v_2,v_2',x_2),(u_3,v_3,v_3',x_3);\Delta \vdash c \downarrow G}{\begin{array}{c}\Gamma,(u_1,v_1,v_1',x_1),\ldots,(u_3,v_3,v_3',x_3);\Delta,u_1,u_2,u_3 \vdash (\mathsf{neg}\ u_1) \vee (\mathsf{neg}\ u_2) \vee (\mathsf{neg}\ u_3)\\ \Rightarrow \mathsf{newv}\ c.\mathsf{newe}\ e_1:(c,v_1)\ e_2:(c,v_2)\ e_3:(c,v_3).G\end{array}}\ c''5.8$$

Bibliography

[1] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *Fifteenth Annual IEEE Symposium on Logic in Computer Science*, 2000.

[2] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In Thomas Reps, editor, *27th Annual Symposium on Principles of Programming Languages*, pages 243–253. ACM Press, January 2000.

[3] Pablo Argón, John Mullins, and Olivier Roux. A correct compiler construction using Coq. In Didier Galmiche, editor, *Informal Proceedings of the Workshop on Proof Search in Type-Theoretic Languages*, pages 3–12, New Brunswick, New Jersey, July 1996.

[4] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions. In *Twenty-fourth Annual ACM Symposium on Theory of Computing*, 1992.

[5] Stephen Bellantoni, K.-H. Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104, 2000.

[6] Francois Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data and Knowledge Engineering*, 5:289–312, 1990.

[7] Luís Caires. *A Model for Declarative Programming and Specification with Concurrency and Mobility*. PhD thesis, University of Lisbon, 1999.

[8] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part 1). In *Fourth International Symposium on Theoretical Aspects of Computer Science (TACS)*. Springer LNCS 2215, 2001.

[9] Luca Cardelli, Philippa Gardner, and Giorgio Ghelli. A spatial logic for querying graphs. In Peter Widmayer, Francisco Triguero, Matthew Hennessy Rafael Morales, Stephan Eidenbenz, and Ricardo Conejo, editors, *29th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 597–610. Springer LNCS 2380, 2002.

[10] Vuokko-Helena Caseiro. *Equations for defining poly-time functions*. PhD thesis, University of Oslo, 1997.

[11] Iliano Cervesato. *A Linear Logical Framework*. PhD thesis, Dipartimento di Informatica, Università di Torino, February 1996.

[12] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[13] Iliano Cervesato, Frank Pfenning, David Walker, and Kevin Watkins. A Concurrent Logical Framework II: Examples and Applications. Technical Report CMU-CS-02-102, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 2002, revised May 2003.

[14] Boleslaw Ciesielski and Mitchell Wand. Using the theorem prover Isabelle-91 to verify a simple proof of compiler correctness. Technical Report NU-CCS-91-20, College of Computer Science, Northeastern University, December 1991.

[15] A. Cobham. The intrinsic computational complexity of functions. In Y. Bar-Hellel, editor, *Proceedings of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science*, pages 24–30, 1965.

[16] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[17] Stephen Cook. The complexity of theorem proving procedures. In *Third Annual ACM Symposium on Theory of Computing*, pages 151–158. Association of Computing Machinery, 1971.

[18] Haskell B. Curry and Robert Feys. *Combinatory Logic*. North-Holland, 1958.

[19] N. G. de Bruijn. A survey of the project automath. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, 1980.

[20] N.G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, pages 29–61, Versailles, France, December 1968. Springer-Verlag LNM 125.

[21] M. Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[22] Martin Erwig. Active patterns. In *IFL '96: Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, pages 21–40, London, UK, 1997. Springer-Verlag.

[23] M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. In *14th IEEE Computer Security Foundations Workshop*, pages 160–173. IEEE Computer Society, 2001.

[24] Harald Ganzinger and David McAllester. A new meta-complexity theorem for bottom-up logic programs. In *Proc. International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Computer Science*, pages 5114–5128. Springer-Verlag, 2001.

[25] Harald Ganzinger and David McAllester. Logical algorithms. In *18th International Conference in Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 209–223. Springer, 2002.

[26] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

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

[28] Robert Givan and David McAllester. Polynomial-time computation via local inference relations. *ACM Transactions on Computational Logic*, 3(4):521–541, October 2002.

[29] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh lcf. In *Lecture Notes in Computer Science*, volume 78. Springer Verlag, Berlin, Germany, 1978.

[30] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

[31] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society, June 1987.

[32] Robert Harper and Frank Pfenning. On equivalence and canonical forms in lf type theory. Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University, July 2000.

[33] R. Harrop. Concerning formulas of the types $A \rightarrow B \vee C, A \rightarrow (Ex)(Bx)$. *Journal of Symbolic Logic*, 25:27–32, 1960.

[34] Martin Hofmann. *Typed lambda calculi for polynomial-time computation.* Habilitation thesis, TU Darmstadt, 1998.

[35] Martin Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure Applied Logic*, 104(1-3):113–166, 2000.

[36] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183:57–85, 2003.

[37] W. A. Howard. The formulas-as-types notion of construction. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[38] Dean Jacobs. A pragmatic view of types for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 7, pages 217–227. MIT Press, logical frameworks edition, 1992.

[39] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of computer computations*, pages 85–103. Plenum Press, New York, NY, 1972.

[40] Richard M. Karp. On complexity of combinatorial problems. *Networks*, 5:45–68, 1975.

[41] Daniel Leivant. Subrecursion and lambda representation over free algebras. In S. Buss and P. Scott, editors, *Feasible Mathematics*, pages 281–292. Birkhauser, 1990.

[42] Daniel Leivant. A foundational delineation of computational feasibility. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 2–11. IEEE, 1991.

[43] D. Lester and S. Mintchev. Towards machine-checked compiler correctness for higher-order languages. In L. Pacholski and J. Tiuryn, editors, *Proceedings of the 8th Workshop on Computer Science Logic (CSL'94)*, Kazimierz, Poland, September 1994. Springer LNCS 933.

[44] F. D. Lewis. *Solving NP-Complete Problems.* World Wide Web. This is web version of a work in progress.

[45] Pablo López, Frank Pfenning, Jeff Polakow, and Kevin Watkins. Monadic concurrent linear logic programming. In *7th International Symposium on Principles and Practice of Declarative Programming*, Lisbon, Portugal, July 2005.

[46] W. Marrero, E. Clarke, and S. Jha. Model checking for security protocols. Technical Report CMU-CS-97-139, Carnegie Mellon University, 1997.

[47] Catherine Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1), 1992.

[48] Catherine Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

[49] Catherine Meadows. Analysis of the internet key exchange protocol using the nrl protocol analyzer. In *IEEE Symposium on Security and Privacy*, pages 216–231. IEEE Computer Society, 1999.

[50] Catherine A. Meadows. Formal verification of cryptographic protocols: A survey. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*. LNCS, Springer-Verlag, 1994.

[51] Dale Miller. Hereditary harrop formulas and logic programming. In *Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science*, Moscow, August 1987.

[52] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[53] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[54] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[55] Jeff Naughton and Raghu Ramakrishnan. Bottom-up evaluation of logic programs. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic*. MIT Press, 1991.

[56] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[57] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *Logic in Computer Science*, pages 93–104, 1998.

[58] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[59] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[60] Cornelia Pusch. Verification of compiler correctness for the WAM. In J. Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOL'96)*, Turku, Finland, August 1996. Springer-Verlag LNCS 1125.

[61] Zhenyu Qian. Unification of higher-order patterns in linear time and space. *J. Log. Comput.*, 6(3):315–341, 1996.

[62] Uday S. Reddy. A typed foundation for directional logic programming. In E. Lamma and P. Mello, editors, *Proceedings of the Third International Workshop on Extensions of Logic Programming*, pages 282–318, Bologna, Italy, February 1992. Springer-Verlag LNAI 607.

[63] Jason Reed. Redundancy elimination for LF. In *Logical Frameworks and Meta-languages (LFM)*, 2004.

[64] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag.

[65] Carsten Schürmann. A type-theoretic approach to induction with higher-order encodings. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence and Reasoning(LPAR 2001)*, pages 266–281, Havana, Cuba, 2001. Springer Verlag LNAI 2250.

[66] Carsten Schürmann and Jatin Shah. Representing reductions of NP-complete problems in logical frameworks — a case study. Technical Report TR-1254, Yale University, 2003.

[67] Carsten Schürmann and Jatin Shah. Representing reductions of NP-complete problems in logical frameworks – a case study. In *Mechanized Reasoning about Languages with variable binding (MERLIN)*, Uppsala, Sweden, August 2003.

[68] Helmut Schwichtenberg. Classifying recursive functions. In E. Griffor, editor, *Handbook of Computability Theory*, pages 533–586. North–Holland, Amsterdam, 1999.

[69] Robert F. Stark. The declarative semantics of the prolog selection rule. In S. Abrhamsky, editor, *Proceedings of the Ninth Annual Symposium on Logic in Computer Science*, Paris, France, July 1994. IEEE Computer Society Press.

[70] Rakesh M. Verma. General techniques for analyzing recursive algorithms with applications. *SIAM Journal of Computing*, 26(2):568–581, April 1997.

[71] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. A Concurrent Logical Framework I: Judgments and Properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 2002, revised May 2003.