INTEGRATION OF DECISION PROCEDURES INTO HIGH-ORDER INTERACTIVE PROVERS

by

Yegor Bryukhov

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

2006

UMI Number: 3204956

UMI Microform 3204956

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

	(Sergei Artemov)
Date	Chair of Examining Committee
	(Theodore Brown)
Date	Executive Officer
	Melvin Fitting
	Jason Hickey
	Rohit Parikh
	Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

INTEGRATION OF DECISION PROCEDURES INTO HIGH-ORDER INTERACTIVE PROVERS

by

Yegor Bryukhov

Adviser: Professor Sergei Artemov

An efficient proof assistant uses a wide range of decision procedures, including automatic verification of validity of arithmetical formulas with linear terms. Since the final product of a proof assistant is a formalized and verified proof, it prompts an additional task of building proofs of formulas, which validity is established by such a decision procedure.

We present an implementation of several decision procedures for arithmetical formulas with linear terms in the MetaPRL proof assistant in a way that provides formal proofs of formulas found valid by those procedures.

We also present an implementation of a theorem prover for the logic of justified common knowledge $\mathsf{S4}^{\mathsf{J}}_n$ introduced in [Artemov, 2004]. This system captures the notion of *justified common knowledge*, which is free of some of the deficiencies of the usual common knowledge operator, and is yet sufficient for the analysis of epistemic problems where common knowledge has been traditionally applied. In particular, $\mathsf{S4}^{\mathsf{J}}_n$ enjoys cut-elimination, which introduces the possibility of automatic proof search in the logic of common knowledge. Our implementation automatically builds cut-free sequent proofs

in $\mathsf{S4}_n^{\mathsf{J}}$. This prover is based on the existing matrix-based prover for intuitionistic logic [Schmitt *et al.*, 2001].

Contents

Chapter 1

Introduction

There are few mathematical results that utilized computers to check a multitude of decidable facts. The number and complexity of these facts make them intractable for human checking; even if these results were converted to a humanly readable form, they could hardly be checked "by hand" reliably. And this is probably indicative of a certain tendency in mathematics: it is likely that there will be more and more such results. Today, some mathematicians do not consider such computer-generated proofs to be quite acceptable, but we are likely to see their widespread acceptance sooner or later. Perhaps it will be the norm someday to submit papers with proofs checked by one of the publicly recognized automatic proof checkers (just as TEX a de facto standard today, met with initial skepticism).

Similarly, program verification becomes more and more important in the computer industry because of the growing need for formal verification in fi-

nancial systems, life support systems, power grid control systems, traffic control systems (aboveground, underground, air, or space), expensive remotely controlled equipment (space programs), etc.

For a tool to be useful and usable in pursuit of computer-checked proofs, it should be capable of filling in "simple" gaps in its users' reasonings. Such tools are called proof assistants. In Chapter 2, proof assistants are reviewed briefly with an emphasis on the MetaPRL logical framework, in which all the work described in this thesis is implemented.

To be truly interactive, a proof assistant needs most often to have response time in fractions of a second. It has to fill certain classes of reasoning gaps automatically so that a user can concentrate on other issues.

One can distinguish two kinds of gaps in informal proofs:

(A) trivial issues that would not even be addressed in an informal proof; they should be resolved by a proof assistant automatically and instantaneously;

(B) other issues that can be solved by a computer algorithm in a reasonable amount of time.

Of course, the description of (A) and (B) above is quite informal.

1.1 Automatic Proof Search

There is one more concern about automatic solving of certain kinds of problems. Typically, there are decidability results that suggest algorithms (decision procedures) solving problems of a certain kind. These algorithms are supported by some informal correctness theorems, but they do not construct formal and checkable proofs.

Earlier implementations of proof assistants used such algorithms as is; this approach means that one has to trust the implementation of the algorithm. One also has to believe in the correctness of the algorithm because decidability results are usually proved outside of the proof assistant at hand. It means that every decision procedure extends the so-called *trusted core* of the proof assistant at hand. Now, imagine that we change something in the background logic of the proof assistant. We cannot be sure that the decision procedures it uses are still compatible with this revised logic, so one has to revise *all* (!) decision procedures used.

We can, as an alternative, use algorithms that actually construct proofs for their conclusions within the logic upon which that proof assistant is based. In this way, even if an algorithm or its implementation is flawed, the worst thing that could happen is that it will fail to prove something. This approach does not affect the trusted core of the proof assistant and, as a result, revisions of the proof assistant's base logic do not demand revisions of these algorithms. This approach is potentially less efficient (in terms of time of execution);

it has been extensively explored since the 1990s, and though results are not always stunningly fast, we can at least say that converting a decision procedure to a proof-constructing procedure is usually doable [Boulton, 1994; Harrison, 1994; Harrison, 1996; Schmitt and Kreitz, 1995; Schmitt, 1999; Schmitt *et al.*, 2001].

1.1.1 Reasoning about Integers

One particular type of problem that can and should be solved automatically is whether or not a certain arithmetical formula logically follows from a given set of assumptions. Of course, it is not decidable for Peano Arithmetic of addition and multiplication, but it is decidable for Presburger Arithmetic of addition (arithmetic of linear terms, [Presburger, 1929]). Many mathematicians and computer scientists do not realize how important numbers are in their everyday reasoning. One uses numbers whenever (s)he needs to measure something, index something, or use the principle of induction, hence effective automation of integer reasoning is a very important feature of any proof assistant.

MetaPRL is a relatively new proof assistant that had no theory of integer numbers at all, when the author got involved in its development. Chapter 3 of this thesis describes the author's efforts to bring MetaPRL up to speed in the area of integer reasoning automation. All implemented algorithms not only decide but also construct proofs of their findings.

1.1.2 Deduction in Multi-Agent Systems of Knowledge

An ability to reason about the knowledge of intellectual agents who can reason about the world and each other's knowledge has uses in a range of applications, from philosophy to economics to security analysis (cf. [Aumann, 1976; Fagin *et al.*, 1995; Meyer and van der Hoek, 2004]). One of the founders of the formal logic of knowledge, R.J. Aumann, was awarded the 2005 Nobel Prize in Economics.

Common knowledge is a key attribute of multi-agent systems of knowledge [Fagin *et al.*, 1995]. Until recently, all known axiomatic descriptions of common knowledge used fixed-point characterizations. Such systems do not enjoy cut-elimination [Alberucci and Jäger, 2005] hence use of traditional proof-theoretic techniques is problematic, and no automatic proof search seems possible[1]. There is an alternative axiomatization [Jäger *et al.*, 2005] that avoids the cut rule entirely, but it contains a rule where the number of premises is exponential on the length of a "main" formula. Though interesting theoretically, the latter result is dubious practicality because of this exponential number of premises.

An alternative approach to common knowledge was developed in [McCarthy *et al.*, 1979; Artemov, 2005]. The paper [McCarthy *et al.*, 1979] introduced the 'any fool knows' modality axiomatically and described their epistemic Kripke-style models. Artemov's approach was based the idea of on

[1]There are, of course, tableau-based provers, but what they do is more of a semantical proof

introducing justification into formal epistemology and suggesting a new epistemic notion of 'justified knowledge,' i.e., the knowledge of F which is based on efficient access to explicit evidence for 'F.' As was shown in [Artemov, 2005], justified knowledge systems axiomatically coincide with the common knowledge systems from [McCarthy *et al.*, 1979]. Justified common knowledge leads to stronger modalities and leaner axiom systems than traditional common knowledge [Antonakos, 2006].

Systems of justified common knowledge from [McCarthy *et al.*, 1979] and [Artemov, 2005], based on the modal logics T and S4, enjoy cut-elimination (cf. [Artemov, 2005]), which makes possible the use of automated proof search in the area of common knowledge.

In Chapter 4, we present an implementation of a proof search procedure for a multi-agent system with justified common knowledge which produces sequent-style proofs as its output. Basically, we again face the problem of implementing a decision procedure that produces not only a yes/no answer to the question of whether a given formula is valid, but which also has to produce a full proof in case of a formula recognized as valid. The work is based on the existing implementation for intuitionistic first-order logic in MetaPRL [Schmitt *et al.*, 2001].

While integer prover is an applied system tool, this epistemic prover in its current form is more of a research tool. However, taking into account the present and potential uses of common knowledge systems, this epistemic prover may be regarded as a working prototype of a general-purpose computerized epistemic tool.

Chapter 2

Proof Assistants

There is multitude of proof automating systems ranging from fully automatic provers to proof assistants. The former are capable of treating only very limited (propositional and first-order) languages. The latter are general-purpose systems that can be perceived as "proof processors with spell-checkers." Like an ordinary word-processor with a spell-checker helps you to write text while catching trivial errors, a proof assistant helps you to write proofs while checking their correctness.

All modern proof assistants are more or less influenced by LCF (Logic for Computable Functions), a seminal paper [Gordon *et al.*, 1979]. It introduced a functional metalanguage that can be used to combine primitive reasoning/justification steps into more advanced patterns of reasoning called tactics. Its another pivotal contribution is isolating a core proof-checker from the rest of the system (using abstract data types). The latter guarantees that

the correctness of proofs depends only on the correctness of the proof-checker part of the system.

Some of the better-known proof assistants are:

- HOL [Gordon and Melham, 1993]
- Coq [INR, 2003]
- Mizar [Rudnicki, 1992]
- Isabelle [Paulson and Nipkow, 1990]
- PVS [Owre and Shankar, 1997; Shankar *et al.*, 1999]
- Twelf [Pfenning, 1994; Pfenning and Schuermann, 2002]
- NuPRL [Constable *et al.*, 1986]
- MetaPRL [Hickey *et al.*, 2003]

The references cite the most recent canonical descriptions of which we are aware.

The majority of proof assistants use a goal-driven derivation: instead of deriving a theorem from axioms and already-established facts (forward or bottom-up derivation), the user starts from the goal and "decomposes" (refines) it down to axioms and/or established facts (top-down derivation). This, of course, is more natural because one normally does not know in advance the exact set of ground facts needed en route to the goal and the

exact path from those ground facts to the goal. With the top-down approach, at every moment a partial derivation is a tree with possible ungrounded leaves. It becomes complete when all leaves are ground. With the bottom-up approach, a partial derivation is a forest of ground trees; the problem here is that it is possible to be left with some trees that are irrelevant to the goal (resulting in wasted time).

2.1 HOL

HOL stands for (classical) higher-order logic. It uses predicate calculus with terms from the typed lambda calculus. Historically, the main application of the HOL system is the specification and verification of hardware. HOL is a direct descendant of Edinburgh LCF. The first release of HOL was in 1988 and there are currently several versions of HOL (HOL88, HOL90, HOL98, HOL Light, HOL 4) [Gordon and Melham, 1993].

2.1.1 Large mathematical formalizations in HOL

One can find the following areas of mathematics formalized in HOL: construction of real numbers, real analysis up to the fundamental theorem of calculus, complex numbers up to the fundamental theorem of algebra, a weak form of the Prime Number Theorem, floating-point arithmetic, etc.

2.2 Coq

Coq is based on the Calculus of Inductive Constructions, extension of Girard's polymorphic λ-calculus F_ω. It is targeted for specification and verification of programs [INR, 2003].

2.2.1 Large mathematical formalizations in Coq

Coq has the following formalized theories: real analysis, constructive category theory, elements of constructive geometry, group theory, domain theory, fundamental group theory.

2.3 Mizar

In development since 1974 and aimed at reconstructing general mathematical knowledge, Mizar is based on the Tarski-Grothendieck axiomatization of set theory. Unlike other proof assistants mentioned here, Mizar is actually a non-interactive proof-checker [Rudnicki, 1992].

2.3.1 Large mathematical formalizations in Mizar

There is a journal `http://mizar.uwb.edu.pl/JFM/` devoted solely to the formalizations of mathematics in Mizar. The Mizar community formalized the Jordan Curve Theorem for special polygons, and are in the process of formalizing the general Jordan Curve Theorem and "A Compendium of Con-

tinuous Lattices."

2.4 Isabelle

Isabelle is a logical framework (though dubbed 'generic proof assistant'), meaning that it is not tightly bound to one specific logic. Different logics can be defined using Isabelle's meta logic Isabelle/Pure. The first release was in 1986. It now supports among others HOL, FOL, ZF, HOL with Scott's Logic for Computable Functions (domain theory) added, short examples of an extensional version of Martin-Löf's Type Theory (ITT), Barendregt's Lambda Cube, and others [Paulson and Nipkow, 1990].

2.4.1 Large mathematical formalizations in Isabelle

Developments in Isabelle include Java formalization (type system, operational semantics, type-safety, axiomatic semantics, etc.), and the Hahn-Banach theorem for linear spaces and for normed spaces.

2.5 PVS

Prototype Verification System (PVS) commenced in 1990 as an attempt to fill the gap between theorem provers and proof checkers. This shaped the system substantially; it has a very elaborate library of decision procedures and is used for hardware and software verification. The PVS language is simply typed

higher-order logic (as HOL) extended with subtyping, dependent typing, and parametric theories, which makes it somewhat closer to Coq and NuPRL. *It is also the only non-ITT system that generates type-checking subgoals* [Owre and Shankar, 1997; Shankar *et al.*, 1999].

2.5.1 Large mathematical formalizations in PVS

PVS has the following formalized mathematical theories analysis, domain theory, program semantics and graph theory.

2.6 Twelf

Twelf is a logical framework, a successor of Elf [Pfenning, 1994]. It supports the LF logical framework [Harper *et al.*, 1993] and the Elf constraint logic programming. Twelf seems to be rather more a research tool than a proof assistant with a well-developed library of definitions, theorems, and decision procedures.

2.7 NuPRL

The project [Constable *et al.*, 1986] started in 1979 and built around Martin-Löf's Type Theory (ITT). It is also aimed to program specification and verification. NuPRL is also a direct descendant of Edinburgh LCF.

2.7.1 Large mathematical formalizations in NuPRL

NuPRL has a number of formalized mathematical theories including but not limited to constructive real analysis, computational abstract algebra (multivariate polynomial arithmetic, unique factorization domains), Girard's paradox, extracting constructive content from classical proofs (Higman's Lemma), automata theory, and Turing machines.

2.8 MetaPRL

MetaPRL is the most recent development among the aforementioned systems; the first paper on it was published in 1997. MetaPRL is a logical framework (again, more than one logic) with the main effort invested in Constructive Type Theory; it also supports first-order logic, Aczel's constructive set theory, LF [Harper *et al.*, 1993], and others.

The MetaPRL project was initiated by Jason Hickey at Cornell University to address scalability and modularity limitations of NuPRL (at that time MetaPRL was called Nuprl-light).

MetaPRL developed as a part of PRL project, and can be considered as a direct descendant of NuPRL. The main design decisions that shaped MetaPRL were modularity and efficiency. As a result, MetaPRL has tested to be over 100 times faster than NuPRL in major domains [Hickey *et al.*, 2003]. The MetaPRL engine can also be used in a distributed mode, which gives super-linear speedup on a small number of CPUs.

2.8.1 Large mathematical formalizations in MetaPRL

Because MetaPRL is a relatively young system, it has only a few noticeable formalizations. It has constructive abstract algebra both for CTT and CZF. There is currently work in progress on a framework for verifiable compilers.

2.8.2 Derived Rules

Derived rules provide a mechanism that serves modularity. One can define a data structure and specify several inference rules for it, thus providing a logical interface to the data structure. These rules may be considered *primitive*. Alternatively, a data structure might be defined be means of other entities, and the mentioned rules should be proved admissible (in which case we say that these are *derived* rules). Users of such data type would not be affected by switching from primitive rules to derived ones or vice versa.

Derived rules are known to be more powerful than propositional statements, e.g., certain induction principles in NuPRL are not expressible in propositional form.

2.8.3 Computational Rewrites

MetaPRL supports not only inference rules but rewrites (reductions) as well. Rewrites state an equivalence of two terms in any context. MetaPRL allows derived rewrites and conditional rewrites.

Because of context independence, rewrite application is an order of mag-

nitude faster than rule application.

2.8.4 Resources

Because of multiple (often incompatible) logics and modular implementation of each logic in MetaPRL, it is almost impossible to make a general-purpose tactic with hard-coded behavior that would behave correctly in every context. The resource mechanism is the treatment for this problem. A resource is a collection of data with inheritance over modules and per-theorem granularity of scope control. For a given theorem, a resource exposes only those elements defined in included modules or in a current module up to a current theorem. Hence, a tactic that makes use of a resource always retrieve only data relevant to the current location.

Most currently used resources are coupled with term storage designed for fast term-pattern retrieval (discrimination trees). Applied to a given term, such a resource retrieves data associated with the most specific pattern matching that term within the current scope.

2.8.5 Resource Annotation

Rules and rewrites can be easily annotated for addition to particular resources, e.g., annotation of rule `gt2ge`:

```
interactive gt2ge {| ge_elim |} 'H :
   sequent { <H>; 'a > 'b; <J>; 'a >= 'b + 1 >- 'C } -->
```

```
sequent { <H>; 'a > 'b; <J> >- 'C }
```

where `ge_elim` instructs the arithmetical tactics to use this rule to convert a $>$-relation to a $\geq$-relation (these tactics use $\geq$ to represent all constraints uniformly). This feature renders resources very declarative and easy to use, though author of a resource has to provide some code in order for resource annotations to work.

2.8.6 Generic Tactics

The resource mechanism permits the writing of tactics that do not depend on a particular logic. Such tactics include term decomposition, term simplification rewriting, type-inferencing, and decision procedures.

Generic tactics simplify inclusion of new logics into MetaPRL because its high-level infrastructure can be reused and an author of a new logic theory can concentrate on specifics of the logic itself.

2.9 General-purpose logics used in proof assistants

Logics used in proof assistants fall into two categories — set theories and type theories. A very interesting overview of the latter can be found in [Barendregt and Geuvers, 2001].

2.9.1 Set Theory

Only Mizar uses set theory as its "main" logic, it is based on Tarski-Grothendieck axiomatization which is equivalent to ZFC extended with the axiom of existence of arbitrary large cardinals. Isabelle supports ZF, and MetaPRL has constructive set theory as one of its supported logics (though ITT is mainstream here) [Aczel and Rathjen, 2001; Aczel, 1986].

It is characteristic that the only proof assistant that is targeted for doing pure mathematics is based on set theory; all other proof assistants that are targeted (more or less) for applications lean towards more expressive languages. The reason is that set-theoretic coding of objects is unfeasible in practice, i.e., set theory cannot capture computations (and proofs as objects) in a feasible way.

2.9.2 Higher-order logic with simply typed lambda calculus

HOL is based on Church's simple type theory with classical logic; it is a higher-order logic. PVS uses a similar system with several extensions, e.g. subtyping.

2.9.3 Calculus of Inductive Constructions

The calculus of constructions (CIC) is introduced in [Coquand and Huet, 1988] and in Barendregt's cube [Barendregt, 1992]; it is a $\lambda P\omega$ (or λC), i.e.,

polymorphic or second-order lambda calculus with types depending on types and terms. There was an earlier version without hierarchy of sorts (universes) that is known to be too strong – it becomes inconsistent with certain desired extensions, e.g., with the strong dependent sum (Σ-type).

(Impredicative) inductive definitions [Paulin-Mohring, 1993; INR, 2003] were introduced later as a realization that coding of certain things in $\lambda P\omega$ is not satisfactory, e.g., the predecessor function for natural numbers would not evaluate in constant time.

The main disadvantage of the CIC is that it is not fully compatible with the extensionality semantics. This is partly due to the separation of propositions and sets.

2.9.4 Intuitionisitc Type Theory

Intuitionisitc Type Theory (ITT) is a work by Per Martin-Löf [Martin-Löf, 1982; Constable, 2002] and belongs to the constructive/intuitionistic tradition in mathematics, logic, and computer science.

Constructive Type Theory (CTT), the most developed "mainstream" logic in MetaPRL, is a variation of ITT.

ITT is a lambda calculus and enjoys the following features:

Types vs. sets of set theory: a type is not only a collection of objects but also an equality over those objects. Two immediate consequences — the same collection of objects may form more than one type (with different

equality relations), and any object defined as parameterized by objects of a certain type should respect equality of that type (functionality).

Propositions as types: propositions are considered to be types inhabited with their witnesses.

Open endedness 1: the philosophy of ITT accepts not only definitional extensions but axiomatic ones as well.

Open endedness 2: as a result of the latter, each type in ITT can potentially be extended.

Type-checking

As was previously mentioned, most ITT rules contain type-checking subgoals (assumptions). Type-checking assumptions are needed because, in general, the typing of terms is not decidable in ITT. Though almost all such subgoals are proved automatically, the system still has to spend time on it.

This has a noticeable impact on the performance of tactics that carry out lots of "in-theory" reasoning. For example, every time two commuting terms are swapped, it remains to be proved that these terms have the appropriate type. In consequence, the conversion of a polynomial to its canonical form, even a relatively simple one, results in dozens to hundreds of type-checking subgoals. We found that tactics that are potentially exponential in time often consume much less time than the subsequent automatic proving of produced type-checking subgoals.

This means that one has to be careful while implementing a complex tactic. One recipe is to do as much reasoning as possible "in a scrapbook" and then expose only the very straight course of reasoning to the system. Another possible solution could be some sort of reflection.

2.9.5 ITT vs. Classical Set Theory

The most important borderline is, of course, that ITT follows intuitionistic/ constructive tradition while Classical Set Theory is the classical logic.

As to the question of how the notion of set relates to the notion of type, we know that Aczel's Constructive Set Theory (CZF) [Aczel and Rathjen, 2001] is interpreted in ITT [Aczel, 1986], and CZF+EM(excluded middle) proves the same theorems as ZF, more on this can be found in [Aczel, 1999].

Chapter 3

Automating Integer Reasoning

3.1 Axiomatization of Integer and Rational Numbers in CTT

We began by defining numbers and their properties. After a number of discussions with Alexei Kopylov, Vladimir Krupski, Aleksey Nogin, and other NuPRL/MetaPRL experts, we reached a consensus about the set of axioms [Bryukhov *et al.*, 2003].

Before looking for a set of axioms, one has to choose either the integers or natural numbers as a basic type (and later define the remaining type via the chosen type). Defining natural numbers on top of integers is more straightforward than the opposite approach. Moreover, we normally do not think about integers as being constructed from natural numbers; on the contrary, we think about natural numbers as a subtype of integers. For these

reasons, we chose to use integers as a primitive type. We used a list of axioms from [Chan, 1982] as a prototype.

3.1.1 Integers

Before showing any specific rule, it is necessary to mention that term equality in CTT is always considered modulo a certain type. For example, we can think about (2,4) and (1,2) as pairs, in which case they are not equal (in type $\mathbb{Z} \times \mathbb{Z}$); on theother hand, we can think about them as representing fractions, in which case they are equal (in type $\mathbb{Q}$). Membership $a \in \mathbb{Z}$ is considered a shortcut for $a = a \ \in \mathbb{Z}$.

There are a number of decisions one has to make when choosing an axiom system for integers.

The first choice we needed to make was whether or not to define the arithmetical operators and types as primitive, postulating all the relevant axioms, or to define everything through the existing constructors. For example, we could have attempted to implement integers as Boolean lists (using a binary representation of numbers). The choice we made is to try to choose a set of axioms that would be universally true across all reasonable ways of defining the arithmetical primitives. These axioms are added to the system as basic postulates of type theory; however, at a later point we could derive them from other type constructors (using MetaPRL's derived rules mechanism [Nogin and Hickey, 2002]).

Another important choice that we had to make concerns the style of

equality reasoning. There are two types of equalities in PRL type theory:

(A) Two terms can be equal as elements of a certain type. For example, two terms $\lambda x.t_1[x]$ and $\lambda x.t_2[x]$ would be equal as elements of a type $A \to B$ (written as '$\lambda x.t_1[x] = \lambda x.t_2[x] \in A \to B$') if for equal inputs of type A, t_1 and t_2 produce equal outputs of type B. Note that two terms could be distinct elements of one type and at the same time be equal in another type — for example, $\lambda x.t_1[x] = \lambda x.t_2[x] \in \texttt{Void} \to T$ is true for arbitrary t_1, t_2 and T, where `Void` is the empty type.

(B) Finer-grained computational/definitional equality [Howe, 1989] specifies that two terms refer to objects that are not just equal but are actually *identical*. For example, the terms $\lambda x.a[x]\, b$ and $a[b]$ are computationally equal (written as '$\lambda x.a[x]\, b \equiv a[b]$'), as well as terms $1 + 2$ and 3.

The equalities of the second kind are easier and more efficient to use. In PRL type theory, we are always allowed to replace a term with a computationally equal one; however we can only replace a term with an equal one when we can prove that the context will tolerate the equalities of the given type. In other words, to be allowed to replace $C[t_1]$ with $C[t_2]$, it is insufficient to be able to prove that $t_1 = t_2 \in T$; we are also required to prove a *well-formedness assumption* stating that C respects the equalities of type T.

Not surprisingly, in our axiomatization we pick computational equality over the equality in a type whenever possible. At the same time, we chose to

still include the typing assumptions in most of the computational equivalence rules. For example, the commutativity of addition rule is as follows:

$$\frac{\Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash (a+b) \equiv (b+a)} \qquad (add\_Commut)$$

The assumptions look redundant in this rule (at least as long as the + operator is not overloaded). However, as we explained above, we want our axioms to stay valid for different formalizations of integers. In particular, in the list implementation of natural numbers (with the type of integers being defined as a disjoint union of two list types and addition defined using recursion over lists), the above rule will be provable only in the presence of the typing assumptions, as we would need to know that a and b are lists in order to be able to use their inductive properties.

We certainly want all arithmetic relations ($=, <, >$, etc.), to be decidable. In PRL type theory, there are two different ways of defining a decidable relation. The straightforward approach is to define a predicate (e.g., a function returning a *proposition*) on numbers with an additional axiom stating that this predicate happens to be decidable. The alternative is to postulate an existence of a function on numbers that returns a *Boolean* result. To better understand the difference between two choices, it is useful to keep in mind that PRL type theory is *constructive*. A PRL proposition P is identified with a type of all *constructive witnesses* for P; there can be many different propositions, and the type of all propositions is a fairly complicated one. In

contrast, PRL booleans is a 1-bit type containing just two Boolean constants. While equivalent propositions could be very different, the booleans are completely transparent — if two booleans happen to be equivalent, they must be identical. Because of the latter feature of the Boolean type, the rule

$$\frac{\Gamma \vdash a \in \mathbb{Z} \quad \Gamma \vdash b \in \mathbb{Z} \quad \Gamma \vdash c \in \mathbb{Z}}{\Gamma \vdash (a < b) \equiv ((a+c) < (b+c))} \qquad (lt\_addMono)$$

will be valid in different implementations of $\mathbb{Z}$, as long as the $<$ relation is defined via a Boolean comparison function. But if $<$ is defined directly as a proposition, then the best that can be guaranteed is $(a < b) \Leftrightarrow ((a+c) < (b+c))$, which is significantly weaker.

When implementing support for numerals, we could either take the traditional route of building all the numerals using 0 and the successor function, or we could simply *expose* MetaPRL's built-in numbers implementation. The first approach looks more reliable (and trustworthy) since one needs only to trust the proof checker; however, this approach is unbearably slow when one wants to do actual computation using these numerals. The second approach looks less reliable as built-in arithmetic now has to be trusted; however, one might argue that it is not adding any new code to the trusted code base, but is instead just exposing what is already a part of the prover. In the end, we decided to implement the second approach.

Here is an outline of our axiomatization, with an overview of the classes of axioms and some examples. The full list of axioms may be found in the

listing of the MetaPRL theories [Hickey *et al.*, , modules Itt_int_base and Itt_int_ext].

(A) Typing properties:

$$\frac{}{\Gamma \vdash \mathbb{Z} \in \mathbb{U}_i} \quad (type\_of\_Int)$$

$$\frac{}{\Gamma \vdash number\{n\} \in \mathbb{Z}} \quad (type\_of\_number)$$

where `number` is the arithmetical *numeral* operator (e.g., `number`$\{n\}$ stands for an arbitrary numeral constant).

(B) Numbers are computationally transparent — two equal integers will necessarily be identical:

$$\frac{\Gamma \vdash a = b \ \in \mathbb{Z}}{\Gamma \vdash a \equiv b} \quad (intCongruence)$$

in other words, equal integer terms always evaluate to some identical canonical form. It also means that $\mathbb{Z}$ has the most fine-grained equality possible.

(C) Reduction of operations (and relations) on integer constants to meta-level operations on integers, e.g:

$$\frac{}{\Gamma \vdash (number\{i\} + number\{j\}) \ \equiv \ number\{i +_m j\}} (reduce\_add\_meta)$$

where $+_m$ performs addition of numeral constants using the underlying internal arithmetic. This rewrite makes possible an evaluation from

$1 + 2$ to 3.

(D) Well-formedness of operations and relations. As with any new operation, we have to state a term of what type it constructs, e.g:

$$\frac{\Gamma \vdash a = a' \in \mathbb{Z} \qquad \Gamma \vdash b = b' \in \mathbb{Z}}{\Gamma \vdash (a + a') = (b + b') \in \mathbb{Z}} \qquad (add\_wf)$$

(E) Equivalence of propositional and Boolean relations. These two rules actually define $=_b$ for integers via equality in $\mathbb{Z}$:

$$\frac{\Gamma \vdash \uparrow (a =_b b) \qquad \Gamma \vdash a \in \mathbb{Z} \qquad \Gamma \vdash b \in \mathbb{Z}}{\Gamma \vdash a = b \in \mathbb{Z}} \qquad (beq\_int2prop)$$

where $\uparrow (t) \ ::= \ (t = true \ \in \ \texttt{Bool})$.

$$\frac{\Gamma \vdash a = b \in \mathbb{Z}}{\Gamma \vdash (a =_b b) \equiv true} \qquad (beq\_int\_is\_true)$$

As previously noted, we decided to define Boolean versions of $=$, $<$, etc., as primitives, and to express propositional inequalities using Boolean ones. However, equality is so fundamental in PRL type theory that we decided to have both Boolean and propositional equality as primitives and have rules that constitute their equivalence.

(F) Ring axioms — commutativity, associativity of $+$ and $*$[1], distributivity,

[1] As we write, the axiomatic definition of multiplication is being replaced with recursive definition by means of addition. Thanks to MetaPRL's derived rule mechanism, this process does not really affect anything.

properties of 0 and 1, e.g:

$$\frac{\Gamma \vdash\ a \in \mathbb{Z}}{\Gamma \vdash\ (a + 0) \equiv a} \qquad (add\_Id)$$

Here, type condition on a is necessary because the rule establishes a bidirectional equivalence relation and we, of course, do not want to allow the replacement of an arbitrary term a with $a + 0$.

(G) Axioms of $<$-order ($<$ is irreflexive, transitive, asymmetric, discrete), connection between $<$ and arithmetic operations, e.g:

$$\frac{\Gamma \vdash\ a \in \mathbb{Z} \qquad \Gamma \vdash\ b \in \mathbb{Z}}{\Gamma \vdash\ ((a\ <_b\ b)\ \wedge_b\ (b\ <_b\ a)) \equiv false} \qquad (lt\_Reflex)$$

$$\frac{\Gamma \vdash\ a \in \mathbb{Z} \qquad \Gamma \vdash\ b \in \mathbb{Z}}{\Gamma \vdash\ ((a\ <_b\ b)\ \vee_b\ (b\ <_b\ a)\ \vee_b\ (a\ =_b\ b)) \equiv true} \qquad (lt\_Trichot)$$

$$\frac{\Gamma \vdash\ a \in \mathbb{Z} \qquad \Gamma \vdash\ b \in \mathbb{Z}}{\Gamma \vdash\ (a\ <_b\ b) \equiv (((a\ +\ 1)\ =_b\ b) \vee_b ((a\ +\ 1)\ <_b\ b))} \qquad (lt\_Discret)$$

The first rule postulates irreflexivity of $<$ relation, the second rule states that $<$ is a linear order, and the third postulates discreteness of integers. These rules define properties of $<_b$ — the Boolean version of the $<$-relation; the propositional version of $<$ and other inequalities are defined via it below; their properties are derivable from properties of $<_b$.

(H) Induction and definition of primitive recursion over $\mathbb{Z}$:

$$\frac{\begin{array}{lll}\Gamma;\ n:\ \mathbb{Z};\ \Delta[n];\ m:\ \mathbb{Z};\ v:\ m\ <\ 0;\ z:\ C[m\ +\ 1] & \vdash & C[m] \\ \Gamma;\ n:\ \mathbb{Z};\ \Delta[n] & \vdash & C[0] \\ \Gamma;\ n:\ \mathbb{Z};\ \Delta[n];\ m:\ \mathbb{Z};\ v:\ 0\ <\ m;\ z:\ C[m\ -\ 1] & \vdash & C[m]\end{array}}{\Gamma;\ n:\ \mathbb{Z};\ \Delta[n]\ \vdash\ C[n]}\ (\mathit{intElimination})$$

This rule is our formulation of the induction principle. We cover both negative and positive numbers in a single rule, so we have two separate induction steps.

The next rewrite is a part of the definition of `ind` — the primitive recursion operation. We need two more (for zero and positive cases) rewrites to define `ind`.

$$\frac{\Gamma \vdash\ x < 0}{\Gamma \vdash\ \begin{array}{l}\texttt{ind}\{x; i, j.down[i; j]; base; k, l.up[k; l]\} \equiv \\ \qquad down[x; \texttt{ind}\{(x+1); i, j.down[i; j]; base; k, l.up[k; l]\}]\end{array}}\ (\mathit{reduce\_ind\_down})$$

(I) Expression of subtraction via negation; >, <=, >= via <; propositional relations via Boolean relations (except equality), e.g:

$$(a\ -\ b)\ ::=\ (a\ +\ (\ -\ b))$$

$$(a\ <\ b)\ ::=\ (\uparrow (a\ <_b\ b))$$

$$(a \ \leq_b \ b) \ ::= \ (\neg_b(b \ <_b \ a))$$

(J) Inductive definition of integer division and remainder operations, e.g:

$$\frac{\Gamma \ \vdash \ 0 \leq a \quad \Gamma \ \vdash \ a \ < \ b \quad \Gamma \ \vdash \ a \ \in \ \mathbb{Z} \quad \Gamma \ \vdash \ b \ \in \ \mathbb{Z}}{\Gamma \ \vdash \ (a \ \% \ b) \ \equiv \ a} \ (\mathit{rem\_baseReduce})$$

3.1.2 Rationals

Sup-Inf is one of the decision procedures we will be discussing in this text; it operates on the rational numbers domain, hence we had to define rationals as well.

We define rational numbers as pairs of integers and positive integers

$$\mathbb{Q} = \{(a, b) \in \mathbb{Z}^2 | (a = 0 \wedge b = 1) \vee (b > 0 \wedge \gcd(a, b) = 1)\}.$$

All other properties of rational numbers (operations, relations, etc.) are expressible via properties of integers.

For an arbitrary pair of integers (a, b) (with second integer non-zero), we define $\mathrm{rat}\{a; b\}$ in such a way that it evaluates to a normalized representation of $\frac{a}{b}$.

3.2 Normalization of Arithmetical Terms

In traditional decision procedures, a normalization step is usually performed based on some representation of arithmetic terms that is internal to the procedure. By *normalization* we mean reduction of polynomials to a certain canonical form. In our case, we need to perform an in-theory normalization as well. It is necessary because it is probably the easiest way to establish in theory that two terms are equal. Reasoning in theory, we convert both to their canonical forms; if the terms were equal, then their normal forms should be identical. In-theory normalization has a higher complexity since commutativity and associativity rules normally only allow swapping neighboring subterms. In-theory normalization also means having to work within a fairly restrictive set of allowed transformations; this makes it noticeably trickier than the "unsupervised" normalization with dedicated representation for arithmetic terms.

The canonical form of a polynomial is achieved by the following steps:

(A) Get rid of subtraction.

(B) Open parentheses using distributivity, move parentheses to the right using associativity of addition and multiplication, perform the basic simplifications (such as $0 \cdot a \to 0$, $1 \cdot a \to a$).

(C) In every monomial, sort (using the commutativity axiom) multipliers in increasing order, with numerical constants pushed to the left (we

put coefficients first because we later have to reduce similar monomials). Multiply the constants if there is more than one numeral in one monomial; but if monomial does not have a constant multiplier at all, put 1 in front of it for uniformity.

(D) Sort monomials in increasing order, reducing similar monomials on the fly. As in the previous step, numerals are pulled to the left (i.e., considered to be the least in the sort order).

(E) Get rid of redundant zeros and ones in the resulting term.

3.3 Decision Procedures

All procedures that we discuss here operate within Presburger arithmetic, i.e., arithmetic of linear forms (or in other words, arithmetic without multiplication, $2x$ is a shortcut for $x + x$). Some of the procedures can establish truth of any closed formula in Presburger arithmetic, some work only with a universal fragment. For the latter procedures we will discuss systems of inequalities without mentioning quantifiers at all. Some procedures are semi-decision procedures, i.e., they can establish truth of some formulas but cannot say anything certain about other formulas.

Whenever we want to establish the validity of a universally quantified formula, we will reason by contradiction. We take the negation of original formula and check whether it is satisfiable: if it is, then we say that the

original formula is invalid (there is a counterexample), otherwise the original formula is valid. The majority of methods convert a negated formula to DNF. We can think about each conjunction as a set of linear constraints: if consistent, then the negated formula is satisfiable and the original is invalid. If none of the constraint sets (each representing one disjunct of the DNF) is consistent, than the negated formula is not satisfiable and the original formula is valid.

There are integer programming algorithms solving the same class of problems, but usually these methods have high overheads related to converting an original problem from symbolic form to a matrix. In practice, matrices are very sparse and many integer programming algorithms do not utilize this fact.

For the rest of this section, we are going to discuss several algorithms in detail. All these methods are implemented in MetaPRL by the author, excepting Cooper's algorithm; the Omega Test implementation is still only partial.

3.3.1 Arith/Nodesat

Arith [Chan, 1982; Hirschkoff, 1996] is the simplest and least-powerful procedure we will discuss. It considers a system of inequalities to be a graph where inequality $x \geq y + n$ is interpreted as an edge from x to y with weight n. Then, if there exists a cycle of positive weight m in the graph obtained this way from the original system of inequalities, we can conclude by transitivity

that $z \geq z + m$ for a vertex z from this cycle – a contradiction.

A positive cycle can be detected by modified Bellman-Ford's shortest-path algorithm.

The Arith procedure does not employ the monotonicity axiom

$$\frac{a \geq b \qquad c \geq d}{a + c \geq b + d}$$

in its general form and cannot prove it, hence the procedure is incomplete.

In Addition, it depends on particular forms of inequalities. Inequalities $x \geq y + n$ and $(x - t) \geq (y - t) + n$ are equivalent, but from Arith's point of view they are different as they produce different edges between different pairs of vertices.

The complexity of the algorithm is $O(n^3)$.

3.3.2 Sup-Inf

Sup-Inf was first introduced by Bledsoe [Bledsoe, 1975] and was soon improved upon by Shostak [Shostak, 1977]. Here is the core idea of the algorithm:

Bledsoe: Pick a variable v that occurs in a given set of constraints, find conservative estimates of lower and upper bounds L, U of v. If $U < L$ (termination condition), we conclude that the set of constraints at hand is incompatible.

This approach works naturally for linear constraints over a field, but we want to decide arbitrary integer constraints. We can intensify the termination condition to be $\lfloor L \rfloor \geq \lfloor U \rfloor$.

This method can be further improved:

Shostak: If the termination condition does not hold, replace all occurrences of v in constraints with any (preferably integer) value from $[L; U]$. Repeat previous steps.

Unfortunately, if at a given moment v was replaced with a non-integer value, it is possible that although the termination condition was never met, the original constraints are nevertheless incompatible.

The algorithm is exponential.

3.3.3 Cooper's Algorithm

Cooper's algorithm [Cooper, 1972] is a classic work in the field; it was the first algorithm for deciding Presburger Arithmetic that designed for efficiency. Although we are not going to implement it, we describe it here because it is a baseline that was implemented by Michael Norrish [Norrish, 2003] in HOL. It is interesting to observe its performance compared to other methods.

The method is complete for Presburger Arithmetic, i.e., it can establish the truth of any closed formula. It is based on variable elimination. The core of the method replaces formulas of the form $\exists x.F(x)$ with an equivalent one that neither quantifies over nor contains x. Universal quantifiers are

expressed via existential quantifiers. By applying the core method repeatedly, we can eliminate all variables and simply evaluate the truth of the resulting variable-free formula.

Unlike many other methods, Cooper's algorithm neither requires conversion of the formula at hand to disjunctive normal form nor requires conversion of all arithmetical relations $<, >, \leq, \geq, =, \neq$ to one "standard" relation.

First, consider a limited language with only three relations: $<$, divisibility relation $n|y$ (n divides y evenly), and negated divisibility $n \nmid y$ (n does not divide y evenly), where n is always some (specific) integer number. For this simplified language, the core algorithm being applied to $\exists x.F(x)$ works like this:

(A) Eliminate all negations in F:

1. push negations all the way down to atomic formulas
2. $\neg(a < b)$ becomes $b < a + 1$
3. $\neg(a|b)$ becomes $a \nmid b$
4. $\neg(a \nmid b)$ becomes $a|b$

Let us call this new formula F_1.

(B) Find the least common multiplier μ of all coefficients of x in F_1.

(C) Multiply all relations in F_1 so that x has coefficient μ everywhere (formula F_2).

(D) Replace μx with x (formula F_3).

(E) We now have $\exists x.F(x) \equiv \exists x.F_3(x) \wedge \mu|x$.

(F) F_4 is $F_3(x) \wedge \mu|x$.

(G) Define $F_{-\infty}(x)$ to be $F_4(x)$ with all relations of the form $x < a$ replaced with 'true,' relations of the form $b < x$ replaced with 'false.'

(H) Define δ to be the least common multiplier of all a and c from $a|x + b$ and $c \nmid x + d$.

(I) We now have

$$\exists x.F_4(x) \equiv \bigvee_{i=1}^{\delta} F_{-\infty}(i) \vee \bigvee_{j=1}^{\delta} \bigvee_{(b<x) \text{ from } F_4} F_4(b + j).$$

Proof:

Suppose the right-hand side is true. If the second part of it is true, then we already found x that satisfies the left-hand side. If the first part is true for some i, then

$$F_{-\infty}(i) = F_{-\infty}(i - \delta k)$$

for any k by definition of $F_{-\infty}$. But for a sufficiently large k,

$$F_{-\infty}(i - \delta k) = F_4(i - \delta k),$$

hence there is an x that satisfies F_4.

Now, suppose that the left-hand side is true. And let x_0 to be such that $F_4(x_0)$ is true. If x_0 is of the form $b + j$ for some $(b < x)$ from F_4 and $1 \leq j \leq \delta$, then x_0 also makes true the second part of the right-hand side. Otherwise, consider $x_0 - \delta$; it can only change the truth of some $b < x$ in F_4. It means that $b < x$ and

$$\neg(b < x - \delta) \equiv x - \delta \leq b \equiv x \leq b + \delta,$$

i.e., x is of the form $b + j$ for some $(b < x)$ from F_4 and $1 \leq j \leq \delta$. This contradicts the assumption of our current case (that x is not of such form).

Now, knowing that $F_4(x) \equiv F_4(x - \delta)$, we can repeat our analysis for $x - \delta$ (instead of x) and either find that it satisfies the right-hand side or conclude that $F_4(x) \equiv F_4(x - 2\delta)$, etc.

Eventually, we will reach such $x - \delta k$ that

$$F_4(x - \delta k) \equiv F_{-\infty}(x - \delta k),$$

hence the first part of the right-hand side is true.

QED

We can easily update the algorithm to support other arithmetic relations. For this, we need to understand which relations impose lower bounds and

which do not. Then we update the definition of $F_{-\infty}$ accordingly. For example, $x = a$ should be replaced with $\perp$ and $x \neq a$ should be replaced with $\top$.

If the number of upper bounds is less than the number of lower bounds, then we can use a dual translation that uses F_{∞} and iterates over upper bounds:

$$\exists x.F_4(x) \equiv \bigvee_{i=1}^{\delta} F_{\infty}(-i) \vee \bigvee_{j=1}^{\delta} \bigvee_{(x<a) \text{ from } F_4} F_4(a-j).$$

Cooper also suggests some technique of reducing the number of disjuncts produced by quantifier elimination for the case when the original formula is in a prenex form with all quantifiers, either existential or universal. We will not describe it here (see [Cooper, 1972]).

The algorithm is superexponential (as is any complete algorithm) because of the lower-bound complexity of Presburger Arithmetic. Every nondeterministic Turing machine that decides Presburger Arithmetic requires at least $2^{2^{cn}}$ time to decide the truth of a PA formula of length n for almost all n [Reddy and Loveland, 1978; Fischer and Rabin, 1974]. The known upper bound for Copper's algorithm is $2^{2^{2^{cn \lg n}}}$ in deterministic time for a formula of length n and some constant $c > 1$ [Oppen, 1973].

3.3.4 Fourier-Motzkin's Variable Elimination

This is a "natural" method of solving systems of linear equations and inequalities, that have been familiar since high school. First we transform all constraints to the $\geq$-relation. Then we pick a variable v and transform all inequalities to either $v \geq L_i$ or $U_j \geq v$. Each pair of upper and lower bounds of v is replaced with $U_j \geq L_i$ (here we might face a quadratic explosion of the number of inequalities). All "old" inequalities containing v are removed from our set of constraints. At this moment, v is eliminated and we pick another variable and repeat the elimination process. Eventually, no variables are remain and all inequalities contain only numbers (therefore their truth value can be easily evaluated).

This algorithm, as Sup-Inf, performs naturally on fields. For rings (integer numbers), there is a fix which makes the method applicable. Rings do not have division, hence we cannot transform all inequalities to forms $v \geq L_i$ or $U_j \geq v$. But we can get $c_i v \geq L_i$ or $U_j \geq d_j v$. For each such pair:

$$\begin{array}{cc} U_j \geq d_j v & c_i v \geq L_i \\ c_i U_j \geq c_i d_j v & c_i d_j v \geq d_j L_i \end{array}$$

$$c_i U_j \geq d_j L_i$$

The resulting inequality is free of v. As before, we can apply this transformation to all pairs of lower and upper bounds of v and eliminate v from the system of inequalities completely.

Unfortunately, this transformation could expand the set of integer solutions, resulting in integer solutions of $c_i U_j \geq d_j L_i$ that are not divisible by $c_i d_j$, thus such solutions do not belong to the original system.

3.3.5 Omega Test

Omega Test [Pugh, 1991] is an improvement of Fourier-Motzkin's variable elimination process that makes the algorithm complete. It is at least exponential in the worst case. There are several special subclasses of problems for which polynomial algorithms exist; for those cases Omega Test is known to perform polynomially as well.

We can think about variable elimination as a process of projecting an n-dimensional polyhedron to the $(n-1)$-plane. On this projection, not every integer point has an integer counterpart in the original n-dimensional polyhedron. We can define a subset of projection called the *dark shadow* such that every integer point in it is guaranteed to have an integer prototype.

Suppose there is a solution for

$$c_i U_j \geq d_j L_i$$

but no solution for

$$c_i U_j \geq c_i d_j v \geq d_j L_i$$

because there is no multiple of $c_i d_j$ between $c_i U_j$ and $d_j L_i$ (c_i and d_j are

positive integers). Let $k = \lfloor U_j/d_j \rfloor$. Then

$$c_i d_j (k+1) > c_i U_j \geq d_j L_i > c_i d_j k.$$

From this, $c_i d_j (k+1) - c_i U_j \geq c_i$ and $d_j L_i - c_i d_j k \geq d_j$; their sum is:

$$\begin{aligned} c_i d_j (k+1) - c_i U_j + d_j L_i - c_i d_j k &\geq c_i + d_j & (1) \\ (c_i d_j (k+1) - c_i d_j k) + (d_j L_i - c_i U_j) &\geq c_i + d_j & (2) \\ c_i d_j + (d_j L_i - c_i U_j) &\geq c_i + d_j & (3) \\ c_i d_j - c_i - d_j &\geq c_i U_j - d_j L_i & (4) \end{aligned}$$

This is what we have if there is no integer solution for v; if we have the opposite:

$$c_i U_j - d_j L_i \geq c_i d_j - c_i - d_j + 1$$

, we know that there must be solution for v. Keeping in mind that

$$c_i d_j - c_i - d_j + 1 = (c_i - 1)(d_j - 1)$$

, we say that the dark shadow is:

$$c_i U_j - d_j L_i \geq (c_i - 1)(d_j - 1),$$

and it is equivalent to the real shadow if $c_i = 1$ or $d_j = 1$.

If we did not find integer solutions in the dark shadow but have some in the real shadow, we need to perform a more accurate analysis of the situation. The real shadow minus the dark shadow falls into several pieces; if there is an integer point in one of them, it must be not far from one of the lower bounds of v because all integer points that are too remote will be included in the dark shadow. Such an integer point would satisfy $c_i v = L_i + k$ for some i and some small, non-negative k. Let us see what the bounds are for k.

From 4, we have

$$c_i d_j - c_i - d_j + d_j L_i \geq c_i U_j$$

; together with $U_j \geq d_j v$, we have:

$$\begin{aligned}
c_i d_j - c_i - d_j + d_j L_i &\geq c_i U_j \\
c_i U_j &\geq c_i d_j v \\
c_i d_j - c_i - d_j + d_j L_i &\geq c_i d_j v \\
c_i d_j - c_i - d_j + d_j L_i &\geq d_j (L_i + k) \\
c_i d_j - c_i - d_j + d_j L_i &\geq d_j L_i + d_j k \\
c_i d_j - c_i - d_j &\geq d_j k \\
\left\lfloor \frac{c_i d_j - c_i - d_j}{d_j} \right\rfloor &\geq k.
\end{aligned}$$

So we pick some upper bound $U_j \geq d_j v$, and then for each lower bound

$c_i v \geq L_i$ and k in range

$$0 \leq k \leq \left\lfloor \frac{c_i d_j - c_i - d_j}{d_j} \right\rfloor,$$

add $c_i v = L_i + k$ to the constraint set and check satisfiability for this extended constraint set. If none of the extended constraint sets is satisfiable (and we did not find solutions in the dark shadow), then our constraint set is not satisfiable.

We call these extended constraints 'splinters.' Because splinters have zero thickness, they have empty dark shadows. This means that if the real shadow of a splinter contains integer solutions, we have no choice but to use splinters again. And far as we know, the only way to fight this case explosion is to eliminate equality constraints by solving them (rather than producing more splinters).

Our equalities are Diophantine (integer) equalities; we can solve them using general methods for solving Diophantine equations (e.g. [Banerjee, 1988]), or use the technique suggested by Pugh [Pugh, 1991], which we will describe here.

First we define an operation

$$a \mathbin{\widehat{\bmod}} b = a - b \left\lceil \frac{a}{b} \right\rceil$$

which can be thought of as a signed reminder operation — it is either $a \bmod b$ or $(a \bmod b) - b$, whichever absolute value is smaller (so $|a \mathbin{\widehat{\bmod}} b| \leq \left\lfloor \frac{b}{2} \right\rfloor$).

Note that

$$(a \mathbin{\widehat{\bmod}} b) \bmod b = a \bmod b.$$

For an equation

$$\sum_{i=0}^{n} a_i x_i = 0$$

, we define $m = \min_{i=0\ldots n} |a_i| + 1$ and a new integer variable σ such as:

$$m\sigma = \sum_{i=0}^{n} (a_i \mathbin{\widehat{\bmod}} m) x_i.$$

Such an integer σ exists if the original equation is satisfiable, because $a_i - (a_i \mathbin{\widehat{\bmod}} m)$ is divisible by m for each i.

According to the way we defined m, there is k such that $m = |a_k| + 1$ and $a_k \mathbin{\widehat{\bmod}} m = -\operatorname{sign} a_k$. We can express x_k from 3.3.5:

$$x_k = -(\operatorname{sign} a_k) m\sigma + \sum_{i \neq k} (\operatorname{sign} a_k)(a_i \mathbin{\widehat{\bmod}} m) x_i$$

and eliminate x_k from all constraints.

The constraint we started with will evolve into:

$$\begin{aligned} \sum_{i \neq k} a_i x_i + a_k(-(\operatorname{sign} a_k) m\sigma + \sum_{i \neq k} (\operatorname{sign} a_k)(a_i \mathbin{\widehat{\bmod}} m) x_i) &= 0 \\ |a_k| m\sigma + \sum_{i \neq k} (a_i + |a_k|(a_i \mathbin{\widehat{\bmod}} m)) x_i &= 0; \end{aligned}$$

remember that $|a_k| = m - 1$:

$$|a_k| m\sigma + \sum_{i \neq k} ((a_i - (a_i \widehat{\bmod} m)) + m(a_i \widehat{\bmod} m))x_i = 0$$

and $a_i - (a_i \widehat{\bmod} m)$ is divisible by m, so we can divide this equation by m:

$$|a_k|\sigma + \sum_{i \neq k} \left(\frac{a_i - (a_i \widehat{\bmod} m)}{m} + (a_i \widehat{\bmod} m) \right) x_i = 0,$$

which actually reduces coefficients of the remaining x_i-s by two-thirds.

By doing so several times, we will reduce some coefficient to 1, and we eliminate the appropriate variable, thus reducing the number of variables occurring in the constraint set.

As an heuristic, we can start from an equation that contains a coefficient of the lowest absolute value. By doing this, we hope to get coefficient 1 as soon as possible, and we can also stick with that one particular equation because after elimination of some x_i and injecting a new σ, we will have a coefficient with the smallest absolute value in the same (modified) equation.

More Optimizations and Heuristics

The original description of Omega Test [Pugh, 1991] suggests a number of heuristics to improve overall performance of the method. We found some of them useful, some of them — not:

- The use of vectors (arrays) to represent linear forms (including equa-

tions and inequalities). As an alternative, we attempted to store linear forms in splay trees. Our tests splay trees performed a little better, which we attribute it to the fact that constraints were sparse and some operations on linear forms require the full scan of a container (array or splay tree).

- Some variables might have only upper (or only lower) bounds. We can remove these variables and all constraints that contain them. It makes sense to perform this check on each iteration of (Fourier-Motzkin) variable elimination.

- All constraints are stored in a hash table where keys are normalized vectors/lists of coefficients excluding a constant term. By doing so, we can easily detect redundant constraints like $F \geq 5$ and $F \geq 7$ and eliminate the weaker ones. They also use a custom hashing algorithm to guarantee that opposing constraints $F \geq n$ and $F \leq m$ receive opposing hash keys. (We have not tried such custom hashing yet.)

  Even without custom hashing, a hash table allows for efficient detection of opposing constraints. We use it to detect incompatible constraints $F \geq n$ and $F \leq m$ when $m < n$, and equalities when $m = n$.

- Before even computing hash keys, we normalize the constraint. To normalize a constraint is to divide it by $d = \gcd\{c_i | i = 1 \ldots n\}$. For equality, if d does not divide c_0 evenly, we can immediately conclude that constraints are incompatible. For inequality, if d does not divide

c_0 evenly, we replace c_0 with $\lfloor c_0/d \rfloor$ (along with dividing all other coefficients of the constraint). This tightens the constraint in a real domain but leaves the set of integer solutions unchanged.

We introduced one more major optimization. As previously mentioned, before running Omega Test *per se*, one has to convert the original problem to its DNF with $\geq$ and $=$ relations only. Certain constraints invest in a number of disjuncts, say $a \neq b \equiv (a \geq b+1) \vee (b \geq a+1)$, so $\neq$ doubles the potential number of disjuncts. Suppose that we used Omega Test for the disjunct D_1 where $a \geq b+1$ occurred and it did not use the particular constraint $a \geq b+1$. In this case, its dual disjunct D_2 with $b \geq a+1$ need not to be solved because the solution we have for D_1 will work for D_2 as well.

Our tests indicate that such pruning of the problem space reduced the total running time by several times.

3.4 From Decision Procedures to Tactics

Traditionally, decision procedures provide an answer as to whether or not a certain closed formula in a certain language is true or false. 'Tactic' is an historical name for a piece of code that evaluates to a composition of inference rules in a certain logical system, i.e., each application of a tactic to a complete set of arguments produces (or fais to produce) a derived rule. It is clear that the result of a tactic merits much more confidence than the result of a decision procedure (given that axioms and primitive rules are decidable

sets).

Fortunately, there are some methodologies for converting decision procedures to tactics. What is probably the first work on this topic can be found in Boulton's Ph.D. thesis [Boulton, 1994]; more on this topic can be found in [Harrison, 1994; Harrison, 1996; Norrish, 2003]. We can easily identify three approaches (described below), but in some cases, specialized techniques are used that do not fall under any of these general approaches [Schmitt and Kreitz, 1995; Schmitt, 1999].

3.4.1 Proof by Evidence

In certain cases, a decision procedure produces some evidence of the correctness of the result. Such an evidence can often be used to construct an actual proof of the correctness of the result.

Arith/Nodesat

Arith finds a positive cycle, i.e., a set of inequalities; summed, they produce a contradictory inequality. Hence it is enough to perform this last step inside the logical system at hand — sum up the contradictory inequalities and prove that the result is a contradiction.

3.4.2 Proof of Proforma Theorem

One can prove within a logical system (in which we want to generate formal proofs) that a given decision procedure is indeed correct. Although this approach always works in theory (given that the logical system is strong enough), there are number of disadvantages:

(A) Proving such a theorem formally can be a quite complex and/or tedious task.

(B) Decision procedures are usually used to automate reasoning about certain classes of problems in a proof assistant. But when proving such a proforma theorem, it is necessary to reason about this same class of problems without the automation one seeks to implement (which is simply not yet available).

(C) Later on, it will be necessary to use (execute) this formalized version of the decision procedure; it is often substantially less efficient than its non-formalized version.

Omega Test

As was previously mentioned, the splinter part of this method was implemented in HOL using a proforma theorem by [Norrish, 2003].

3.4.3 Proof Script of Proforma Theorem

Sometime it is too difficult to prove a proforma theorem, but relatively easy to prove its instances for each particular problem (for example, a proforma theorem might use very complicated inductive reasoning, but for each particular problem, one can avoid induction by using iteration of similar reasoning steps).

This approach seems easier than the previous one, but it could be less efficient in certain cases.

Sup-Inf

Each decision step of Sup-Inf can be substantiated by a certain derived rule (and the set of rules needed is finite). We do not need anything else to prove its result correct: memorize the whole trace of Sup-Inf's reasoning and convert it to a proof tree. It does not hurt to prune dead-end branches as you construct the proof tree.

Fourier-Motzkin's Variable Elimination

Each step of this process has an obvious supporting rule.

Omega Test

Fourier-Motzkin's Variable Elimination is a substantial part of Omega Test. The remaining parts — solving equalities, dark shadow, and splinters can

also be treated on a case-by-case basis with proof scripts (i.e., without the need to prove a generic proforma theorem)

Cooper's Algorithm

In [Norrish, 2003], Cooper's Algorithm was implemented via re-proving proforma theorem for each individual problem.

3.5 Discussion

We currently have working implementations of Arith, Sup-Inf, and partial Omega Test. Here is the comparison of our implementations with similar work in Coq (no publications known) and HOL [Norrish, 2003] :

System	Method	Time, secs	Proved/30
MetaPRL	Arith	2.5	8
MetaPRL	SupInf	> 240	???
MetaPRL	Omega	9+16	29
Coq	Omega	14	29
HOL	ARITH_TAC	145	29
HOL	Omega	156	29
HOL	Cooper	> 1200	???

Tests were run on Pentium III 866 MHz, 133MHz FSB, 512 MB RAM, Windows XP Pro Service Pack 2.

HOL-Omega is a full implementation of Omega Test; Coq-Omega is an incomplete implementation of approximately same proof power as MetaPRL-Omega.

MetaPRL-SupInf consumed all available physical memory (512 MB) after four minutes, at which point we stopped it. We must say that while implementing Omega Test, we introduced several optimizations that substantially improved Omega Test's performance; they were not backported to Sup-Inf implementation.

HOL-Cooper ran for 20 minutes and did not terminate, so we stopped it.

We also tried to combine Arith with Omega (try Arith first; if it fails, use Omega) but it did not show any improvement, which means that in detecting "easy" cases, Omega is as effective as Arith.

Our implementation of Omega Test, depending on the actual problem, spent 5 to 95 per cent of the time proving type-checking subgoals, the task unique to ITT. In our test cases, nine seconds were spent on deciding and proving validity and 16 seconds on type-checking, which MetaPRL's main competitor, Coq, does not do at all.

Omega Test and Sup-Inf perform somewhat similar reasoning, but Omega Test has a much simpler iterative structure (in contrast to the highly recursive structure of Sup-Inf) that allows for early detection of incompatible constraints. So even in the worst case, they have similar complexity; in practice, Omega Test will probably always outperform Sup-Inf.

In HOL, the implementation of Cooper's algorithm is inferior to the implementation of Omega Test. This probably happens for the same reason — the lack of earlier detection of contradictions. Note that in HOL they are implemented by the same person, probably within the same time period.

One can argue that since the implementation of Omega Test in HOL is complete, it excuses its lower-level performance in comparison with incomplete versions in Coq and MetaPRL. Final measurements do not include the only test case that is not a valid (true) formula, so it could not add to HOL-Omega running time. All other test cases do not need dark shadows and splinters in order to be solved, so all implementations had the same amount of work to be done. Although the complete implementation could have a higher footprint, it does not explain an order-of-magnitude difference in performance.

Chapter 4

Prover for Logic with Justified Common Knowledge

In this chapter, we present an implementation of a decision procedure for a multi-agent logic with justified common knowledge. As is true of all other implementations presented in this thesis, it produces not only a yes/no answer but also a sequent-style proof if the formula is found valid.

With a certain amount of good will, we can say that in the proof reconstruction we use the 'proof-by-evidence' technique. We say this because the search space, which has to be explored in order to establish the validity of a formula, is huge compared to the actual proof of the formula. The procedure sorts through different systems of equations trying to solve them. A solved system is an evidence of the validity of a formula and has some relevance to the final proof, whereas the whole previous search is irrelevant to the proof

of a formula's validity.

4.1 Evidence-Based Knowledge

Philosophers have pondered over the nature of knowledge for ages. Plato defines knowledge as *justified true belief*. The modal logic approach to knowledge, developed in [Hintikka, 1961; Hintikka, 1962], captures the *true belief* components of Plato's tripartite definition. The *justification* component was introduced into formal epistemology in [Artemov and Nogina, 2005b; Artemov and Nogina, 2005a; Artemov, 2004].

4.1.1 Common Knowledge

Common knowledge is a standard notion in formal epistemology. If K_i $(i = 1, \ldots, n)$ are knowledge operators of individual agents and

$$\mathsf{E}\varphi = \mathsf{K}_1\varphi \wedge \ldots \wedge \mathsf{K}_n\varphi,$$

we can informally define common knowledge as:

$$\mathsf{C}\varphi \leftrightarrow \varphi \wedge \mathsf{E}\varphi \wedge \ldots \wedge \mathsf{E}^m\varphi \wedge \ldots \ .$$

The standard way of defining common knowledge axiomatically is via the *Fixed-Point Axiom*:

$$\mathsf{C}\varphi \leftrightarrow \mathsf{E}(\varphi \wedge \mathsf{C}\varphi)$$

together with the *Induction Rule*:

$$\frac{\varphi \to \mathsf{E}(\psi \wedge \varphi)}{\varphi \to \mathsf{C}\psi}.$$

This formalization does not behave well proof-theoretically; in particular, the above axiomatization does not admit cut-elimination [Alberucci and Jäger, 2005], hence an automatic proof search becomes problematic. [Jäger *et al.*, 2005] gives an alternative axiomatization that does not use cut at all, but contains a rule with a number of premises exponential on the length of a "main" formula. Though interesting theoretically, the latter result is of dubious practicality because of this exponential number of premises.

4.1.2 Evidence-Based Axiomatization

Instead of the modal operator of common knowledge, we consider an explicit evidence operator $t : \varphi$ to stand for

$$t \textit{ is an evidence for } \varphi$$

[Artemov, 1994; Artemov, 1995; Artemov, 2001; Artemov, 2005]. Here t is an evidence term (evidence) constructed from possibly subscripted constants $a, b, c, \ldots$, possibly subscripted variables $x, y, z, \ldots$, binary operations $\cdot$ (application, composition of evidence) and $+$ (union, alternative evidence), and unary operation ! (inspection, verification of evidence).

The following is a grammar for formulas:

$$\varphi = \bot \mid S \mid \varphi \to \varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \mathsf{K}_i\varphi \mid t : \varphi$$

with S standing for a sentence variable and t for an evidence term.

Axioms and rules of $\mathsf{T}_n\mathsf{LP}$:

(A) Classical propositional logic

1. Ten axioms from Kleene [Kleene, 1952] or similar system
2. Modus Ponens

(B) Agents' knowledge principles (usual for logic T)

1. $\mathsf{K}_i(\varphi \to \psi) \to (\mathsf{K}_i\varphi \to \mathsf{K}_i\psi)$
2. $\mathsf{K}_i\varphi \to \varphi$
3. $\dfrac{\vdash \varphi}{\vdash \mathsf{K}_i\varphi}$ internalization of knowledge

(C) Evidence principles (usual for LP)

1. $t : (\varphi \to \psi) \to (s : \varphi \to (t \cdot s) : \psi)$ application
2. $t : \varphi \to !t : t : \varphi$ inspection
3. $t : \varphi \to (t + s) : \varphi$, $t : \varphi \to (s + t) : \varphi$ union
4. $t : \varphi \to \varphi$ reflexivity
5. $\dfrac{}{\vdash c : A}$, where A is any axiom from this list and c is any evidence constant

(D) Connection between knowledge and evidence

1. $t : \varphi \rightarrow \mathsf{K}_i\varphi$, for any agent i

$\mathsf{S4}_n\mathsf{LP}$ is $\mathsf{T}_n\mathsf{LP}$ extended with the axioms $\mathsf{K}_i\varphi \rightarrow \mathsf{K}_i\mathsf{K}_i\varphi$.

$\mathsf{S5}_n\mathsf{LP}$ is $\mathsf{S4}_n\mathsf{LP}$ extended with the axioms $\neg\mathsf{K}_i\varphi \rightarrow \mathsf{K}_i\neg\mathsf{K}_i\varphi$.

All three systems are closed under substitution of evidence terms for evidence variables and formulas for propositional variables.

The following theorems hold for $\mathsf{T}_n\mathsf{LP}$, $\mathsf{S4}_n\mathsf{LP}$ and $\mathsf{S5}_n\mathsf{LP}$ [Artemov, 2005]:

Internalization property. Given $\vdash \varphi$, there is an evidence term t such that $\vdash t : \varphi$.

Lifting lemma. If $\psi_1, \ldots, \psi_p, y_1 : \chi_1, \ldots, y_q : \chi_q \vdash \varphi$, then there is an evidence term $p(x_1, \ldots, x_p, y_1, \ldots, y_q)$ such that

$$x_1 : \psi_1, \ldots, x_p : \psi_p, y_1 : \chi_1, \ldots, y_q : \chi_q \vdash p(x_1, \ldots, x_p, y_1, \ldots, y_q) : \varphi.$$

Common Knowledge relevance. For any evidence term t the Fixed-Point Axiom of common knowledge:

$$t : \varphi \leftrightarrow \mathsf{E}(\varphi \wedge t : \varphi)$$

is derivable in $\mathsf{T}_n\mathsf{LP}$, $\mathsf{S4}_n\mathsf{LP}$ and $\mathsf{S5}_n\mathsf{LP}$.

4.1.3 Justified knowledge modality

Forgetful projection $()^\circ$ of evidence operators:

$$t : \varphi \mapsto \mathsf{J}\varphi$$

replaces $t : \varphi$ by $\mathsf{J}\varphi$. As was shown in [Artemov, 2005], J is a new modality which behaves like an $S4$-modality. The resulting systems $\mathsf{T}_n^{\mathsf{J}}$, $\mathsf{S4}_n^{\mathsf{J}}$, and $\mathsf{S5}_n^{\mathsf{J}}$ are basically T_n, $\mathsf{S4}_n$, and $\mathsf{S5}_n$ augmented by a copy of $\mathsf{S4}$ with the modality J and the *undeniability of evidence* principle:

$$\mathsf{J}\varphi \to \mathsf{K}_i\varphi.$$

Forgetful projections enjoy the following properties [Artemov, 2005]:

Projection. $(\mathsf{T}_n\mathsf{LP})^\circ \subseteq \mathsf{T}_n^J$, $(\mathsf{S4}_n\mathsf{LP})^\circ \subseteq \mathsf{S4}_n^J$, and $(\mathsf{S5}_n\mathsf{LP})^\circ \subseteq \mathsf{S5}_n^J$.

Cut-Elimination $\mathsf{T}_n^{\mathsf{J}}$ and $\mathsf{S4}_n^{\mathsf{J}}$ enjoy cut-elimination.

Realization theorem. There is an algorithm that given a T_n^J-derivation ($\mathsf{S4}_n^J/\mathsf{S5}_n^J$-derivation) of φ , retrieves a $\mathsf{T}_n\mathsf{LP}$-derivation ($\mathsf{S4}_n\mathsf{LP}/\mathsf{S5}_n\mathsf{LP}$-derivation) of some ψ such that $(\psi)^\circ = \varphi$.

Fixed-Point Axiom The justified knowledge modality J satisfies the Fixed-Point Axiom of common knowledge for each of $\mathsf{T}_n^{\mathsf{J}}$, $\mathsf{S4}_n^{\mathsf{J}}$, and $\mathsf{S5}_n^{\mathsf{J}}$:

$$\mathsf{J}\varphi \leftrightarrow \mathsf{E}(\varphi \wedge \mathsf{J}\varphi).$$

Connection with Common Knowledge. Each evidence-based principle is a common knowledge principle but not vice versa:

$$(\mathsf{S4}_n^J)^* \subset (\mathsf{S4}_n^C)$$

$$(\mathsf{S4}_n^J)^* \neq (\mathsf{S4}_n^C)$$

where $()^*$ replaces all occurrences of J with C.

Hence Justified Knowledge is a well-behaved, constructive version of Common Knowledge, which can be used as the latter in solving specific problems (cf. [Artemov, 2004]).

Epistemic systems with common knowledge $\mathsf{T}_n^{\mathsf{J}}$, $\mathsf{S4}_n^{\mathsf{J}}$, and $\mathsf{S5}_n^{\mathsf{J}}$ were first introduced by [McCarthy *et al.*, 1979] axiomatically as the knowledge of a dummy sceptical 'fool,' without any connection to justified knowledge. Artemov's studies of evidence-based knowledge in [Artemov, 2004; Artemov, 2005] led to the notion of the justified knowledge of φ as knowledge when an agent has an access to an evidence of φ. It turned out that the axiom systems for McCarthy's 'any fool knows' modality and Artemov's justified knowledge modality coincide.

In this chapter, we describe an implemented automatic proof-search procedure for $\mathsf{S4}_n^{\mathsf{J}}$ which produces cut-free proofs. Given a cut-free proof in $\mathsf{S4}_n^{\mathsf{J}}$, one can use the Realization Algorithm to recover evidence terms in this proof.

We formalized the *Wise Men* and *Muddy Children* puzzles [Fagin *et al.*,

1995], which our $\mathsf{S4}_n^{\mathsf{J}}$-prover automatically proved. The Muddy Children Puzzle was formulated in several flavors for four children, and depending on its formulation, the proof search and verification times varied from under one second to 90 seconds.

Our $\mathsf{S4}_n^{\mathsf{J}}$-prover is based on existing implementation for intuitionistic logics. The cases of T and $\mathsf{S5}$ as the base knowledge systems, rather than, $\mathsf{S4}$ are left for future work. The Realization Algorithm is currently implemented for $\mathsf{S4}$, but we leave it outside the scope of this thesis since we still need to extend it to the full $\mathsf{S4}_n^{\mathsf{J}}$ and connect it to our $\mathsf{S4}_n^{\mathsf{J}}$-prover.

4.2 Pre-existing Code for First-Order Intuitionistic Logic

We based our work on the existing code of the J-prover in the MetaPRL logical framework [Schmitt *et al.*, 2001]. The implementation is based on a uniform algorithm and matrix characterization for the first-order intuitionistic logic J, the modal logics K, $\mathsf{K4}$, T, $\mathsf{S4}$, $\mathsf{S5}$, and fragments of linear logic [Otten and Kreitz, 1996b; Kreitz and Otten, 1999]. It also contains an algorithm for conversion of an existing matrix proof into a sequent proof [Schmitt and Kreitz, 1995; Kreitz and Schmitt, 2000; Schmitt, 1999].

Note that J here stands for first-order intuitionistic logic and has nothing to do with 'J' in $\mathsf{S4}_n^{\mathsf{J}}$ where it stands for "justified" S4-modality.

The code we started from supported only the first-order intuitionistic logic J, whereas we were interested in classical modal logic. We generalize the matrix characterization of $\mathsf{S4}$ to $\mathsf{S4}^{\mathsf{J}}_n$, and extend that code to support matrix proof search and sequent proof reconstruction for $\mathsf{S4}^{\mathsf{J}}_n$.

We also improve the prefix unification algorithm [Otten and Kreitz, 1996a], which is an important part of the matrix proof search algorithm: on some problems it consumes as much as 99% of the total proof search and construction time. The unification is used iteratively with incremental extension of the equation system. The original algorithm solved new systems from scratch each time. We modified it to reuse the information obtained from the previous iteration in the next iteration when one more equation was added. For systems with a unique most-general unifier, one has only to retain the unifier which itself represents the solved system. Unfortunately, prefix unification problems do not have unique solutions but rather minimal sets of unifiers. Naturally, not all members of a unifier set U for system S survive as members of a unifier set for system $S \cup p = q$.

The algorithm terminates when any unifier is found, but its full state contains the partially explored decision tree which we are required to retain for the next iteration. Passing the unexplored part of the decision tree from iteration to iteration reduced the running time by at least an order of magnitude on difficult problems.

The rest of the chapter is organized in the following way. In Section 4.3, we introduce the matrix characterization of logical validity for propositional classical logic [Bibel, 1981; Bibel, 1987; Andrews, 1981]. In Section 4.4, we extend this notion to S4 [Wallen, 1990]. Section 4.5 contains our extension of the matrix method to $\mathsf{S4}^{\mathsf{J}}_n$, and Section 4.6 describes modifications to the prefix unification algorithm, which is a part of matrix method. Section 4.7 briefly describes the main ideas of matrix proof to sequent proof conversion [Kreitz and Schmitt, 2000; Schmitt, 1999] and our modifications to enable support of $\mathsf{S4}^{\mathsf{J}}_n$. And finally, Section 4.8 describes the Muddy Children Puzzle and its computer-generated solution.

4.3 Matrix Characterization of Logical Validity

The connection method was introduced in [Bibel, 1981; Bibel, 1987] for classical logic. In [Wallen, 1990], it was extended to a number of non-classical logics. Kreitz *et al* suggested a uniform algorithm for a variety of modal logics, intuitionistic logic, and fragments of linear logic [Otten and Kreitz, 1996b; Kreitz and Otten, 1999]; described a uniform method of reconstruction of Gentzen sequent proofs from matrix proofs [Schmitt and Kreitz, 1995; Kreitz and Schmitt, 2000; Schmitt, 1999]; and implemented it for intuitionistic logic in the MetaPRL proof assistant [Schmitt *et al.*, 2001].

In this section, we will introduce the basic concepts of matrix method

and describe logical validity in these terms. We will limit our attention to the propositional case because our target logic $\mathsf{S4}_n^{\mathsf{J}}$ is propositional.

A *formula tree* of a formula F is basically its syntax tree. Each node s corresponds to one particular subformula F_s. We give unique names $a_1, a_2, \ldots$ to each node and call them *positions*. A *label* of a position s is the major (outermost) connective of F_s or is itself F_s if it is atomic. In the latter case, s is called *atomic position* or simply *atom*. The *tree ordering* $<$ on positions is induced by the tree structure where the root is the smallest element with respect to this tree ordering $<$.

Each position is associated with *polarity* in a standard way with the root having polarity 0.

The *principal type* $Ptype(u)$ of a position u is defined by its label and polarity according to the following table:

principal type α	$(A \wedge B)^1$	$(A \vee B)^0$	$(A \to B)^0$	$(\neg A)^1$	$(\neg A)^0$
subformulae polarities	A^1, B^1	A^0, B^0	A^1, B^0	A^0	A^1
principal type β	$(A \wedge B)^0$	$(A \vee B)^1$	$(A \to B)^1$		
subformulae polarities	A^0, B^0	A^1, B^1	A^0, B^1		

Atomic positions have no principal type.

Definition 1 *For a principal type t, we say that two positions a and b of F are* **t-related** *($a \sim_t b$) if the $<$-biggest position c of F that is less than a and b has principal type t ($c = \max\{x \mid x < a \ \wedge \ x < b\}$). a is t-related to a set S iff $a \sim_t s$ for all $s \in S$.*

Definition 2 *For each principal type t, there are* **secondary types** $t_0, \ldots, t_k$ *where k is the arity of connectives that produce type t. For position u of principal type t, its $<$-successors have secondary types t_i according to their relative order as arguments of u. We denote the secondary type of a position v as $Stype(v)$. The root has no secondary type.*

The *matrix(-representation)* of a formula F is a two-dimensional representation of its atomic subformulae without connectives (but with parentheses); it is mainly used for illustrative purposes. In this representation, α-related positions are arranged side by side; β-related positions are arranged one on top of another.

Example 1 *The matrix representation of the formula $\neg((\neg A \vee \neg C) \wedge ((A \wedge B) \vee (B \wedge C)))$ is:*

$$\begin{pmatrix} A^0 \\ C^0 \end{pmatrix} \quad \begin{pmatrix} (A^1 & B^1) \\ (B^1 & C^1) \end{pmatrix}$$

Definition 3 *A* **path** *through a formula F is a maximal set of α-related positions. It can be visualized as a maximal "horizontal" line through the matrix of F (or more precisely, a maximal line that never runs vertically).*

Definition 4 *A* **connection** *is a pair of positions with the same labels but opposite polarities.*

Theorem 1 *A propositional formula F is (classically) valid iff every path through F has a connection.*

A proof of this theorem (first-order case) can be found in [Bibel, 1987].

4.4 Matrix characterization for S4

The matrix characterization for S4 was given in [Wallen, 1990]. For this, we need to add two extra principal types ν, π for modal positions; define prefixes and unification over them; define multiplicity and what is an indexed formula; and define complementary connections and admissible substitutions.

We will use only one modality $\Box$ because $\Diamond$ has no explicit counterpart in the logic of knowledge with justification in which we want to realize $\mathsf{S4}^{\mathsf{J}}_n$-theorems.

Two new principal types are defined according to the table below:

principal type ν	$(\Box A)^1$
subformula polarity	A^1
principal type π	$(\Box A)^0$
subformula polarity	A^0

We denote the set of all positions of primary type ν as $\mathcal{V}$, of primary type π as Π, all positions of secondary type ν_0 as $\mathcal{V}_0$, and of secondary type π_0 as Π_0.

Definition 5 *For a position u, the* **modal prefix** $\mathrm{pre}_M(u)$ *is a string $u_1 \ldots u_n$ of all positions $u_1 < u_2 < \ldots < u_n \leq u$ such that $u_i \in \mathcal{V}_0 \cup \Pi_0$.*

Definition 6 *The modal substitution $\sigma : \mathcal{V}_0 \to (\mathcal{V}_0 \cup \Pi_0)^*$ induces a relation $\sqsubset_M \subseteq (\mathcal{V}_0 \cup \Pi_0) \times \mathcal{V}_0$: if $\sigma_M(u) = p$, then $v \sqsubset u$ for all $v \in \mathcal{V}_0 \cup \Pi_0$ occurring in p.*

In other words, ν_0-positions are considered to be variables and π_0-positions are considered to be constants. All positions in the result of $\sigma(u)$ are less than u with respect to the ordering induced by σ.

Definition 7 *A modal substitution σ is* **S4-admissible** *iff the induced reduction ordering $\lhd = (< \cup \sqsubset)^+$ is irreflexive.*

In S4-tableaux, ν-formulae can be used several times to populate several prefixes (worlds); in order to reflect it in the matrix characterization, we introduce the modal multiplicity $\mu : \mathcal{V}_0 \to \mathbb{N}$ and define the *indexed formula* F^μ which is just a pair – a formula with a multiplicity function over it. A position $u \in \mathcal{V}$ has $\mu(u)$ instances of its successor trees with root v, $u < v$. Whenever we refer to an indexed formula, we imply that positions are duplicated according to the multiplicity function.

Definition 8 *A connection $\{u, v\}$ is* **complementary** *under a substitution σ iff $\sigma(\mathrm{pre}(u)) = \sigma(\mathrm{pre}(v))$. A path is complementary iff it contains a complementary connection.*

Definition 9 *A set of connections C* **spans** *the formula A iff any path through A contains some connection from C.*

Theorem 2 *A formula is S4-valid iff there is a multiplicity μ, an S4-admissible substitution σ, and a set of σ-complementary connections C that spans A [Wallen, 1990].*

4.5 Matrix Characterization of $\mathsf{S4}^{\mathsf{J}}_n$

$\mathsf{S4}^{\mathsf{J}}_n$ can be described as n modalities K_i with $\mathsf{S4}$ behavior, modality J with $\mathsf{S4}$ behavior, and the connection axiom schema

$$\mathsf{J}p \rightarrow \mathsf{K}_i p, \text{ for all } i. \tag{1}$$

To extend matrix characterization from singular $\mathsf{S4}$ to several such modalities, one has to label tree positions with the appropriate modalities and make such positions incompatible for prefix unification purposes.

In terms of tableau rules, connection axiom (1) can be expressed as

$$\frac{T\ \sigma \quad \mathsf{J}p}{T\ \sigma.a_i\ \mathsf{J}p} \text{ for an existing prefix } \sigma.a_i.$$

This means that J-modality is compatible with worlds generated by any modality $\mathsf{K}_1, \ldots, \mathsf{K}_n, \mathsf{J}$. For unification purposes, we allow variable positions related to J-modalities to unify with any string.

We will mark position a with a superscript i if it is related to modality K_i. For J, we will use superscript 0.

Definition 10 *For a position a^i, we define* $\mathrm{sort}\{a^i\} = i$

Definition 11 *To describe prefix unification for $\mathsf{S4}^{\mathsf{J}}_n$, we define a relation $\preceq$ on positions. For two positions a and b, we say that $a \preceq b$ iff a is ν_0^0 or a is ν_0^i, and b is ν_0^i or π_0^i. Basically, this indicates when b can be substituted for a.*

Corollary 1 *$\preceq$ is reflexive and transitive.*

Definition 12 *A substitution $\{V_1 \setminus s_1, \ldots, V_n \setminus s_n\}$ is* **$\mathsf{S4}^{\mathsf{J}}_n$-admissible** *iff it is $\mathsf{S4}$-admissible, and for all i and all a from s_i, holds $V_i \preceq a$.*

Theorem 3 *A formula is $\mathsf{S4}^{\mathsf{J}}_n$-valid iff there is a multiplicity μ, an $\mathsf{S4}^{\mathsf{J}}_n$-admissible substitution σ, and a set of σ-complementary connections C that spans A.*

Proof The only difference between $\mathsf{S4}$ and $\mathsf{S4}^{\mathsf{J}}_n$ tableaux are the rules of propagation of $\Box$ from prefix to prefix. Our adjustments in the definition of admissible substitution reflects those differences exactly.

Suppose there is a matrix proof, i.e., a multiplicity assignment, an admissible substitution, and a spanning set of connections. Let us show that there is a tableau proof as well.

We can modify our definition of a path through a formula A^{μ} as follows. We say that some sets of α-related positions are paths (and our original paths become atomic paths). Instead of $P \cup \{v\}$, we will write P, v.

(A) Root is a path.

(B) If P, u is a path; and u is an α-position with subformulae u_0 and u_1; then P, u_0 and P, u_1 are paths as well.

(C) If P, u is a path; and u is a β-position with subformulae u_0 and u_1; then P, u_0, u_1 is a path.

(D) If P, u is a path; and u is a π-position with subformula u_0; then P, u_0 is a path.

(E) If P, u is a path; and u is a ν-position with subformula u_0; then P, $u_{0,1}, \ldots, u_{0,\mu(u)}$ is a path where $u_{0,i}$ are instances of u_0.

If we augment each position in paths with its prefixes, paths will become $\mathsf{S4}^{\mathsf{J}}_n$-tableau branches. Atomic paths will be closed branches (if there is a spanning set of connections). Our path generation rules become valid tableau rules as long as prefixes are introduced by π-rules before they are used by ν-rules. But we can always apply the ν-rule after the appropriate π-rule – they cannot block each other because their active positions are from different branches of the syntax tree (we need only respect the reduction ordering when we choose the order in which to process the positions). Hence if there is a matrix proof, then there is a tableau proof as well.

Suppose our formula is valid and we have a tableau proof of it. Then we can find the appropriate multiplicity and an $\mathsf{S4}^{\mathsf{J}}_n$-admissible substitution.

Let us fix the tableau rules (propositional rules are standard, closures are allowed over atomic formulas only):

$$\frac{F\ \sigma \quad \Box_i A}{F\ \sigma.\alpha_i \quad A}\ (\pi^i) \qquad \frac{T\ \sigma \quad \Box_i A}{T\ \sigma.\tau \quad A}\ (\nu^i)$$

with $\Box_i = \mathsf{K}_i$ for $i > 0$, $\Box_0 = \mathsf{J}$; α is always a fresh symbol, and τ is a string of "old" symbols subscripted by i's only if $i > 0$, or subscripted arbitrarily when $i = 0$.

The number of times a ν^i-rule was applied to a particular subformula in the tableau proof is exactly the multiplicity of this subformula in the formula tree.

Every π^i-rule associates α_i with the π^i_0 position of the formula tree relevant to $F\ \Box_i A$. Every ν^i-rule produces a substitution $V \setminus \tau^*$, where V is the ν^i position of the formula tree relevant to $T\ \Box_i A$; τ^* is a string of π_0-positions associated by π-rules with the elements of τ.

The resulting substitution is $\mathsf{S4}^{\mathsf{J}}_n$-admissible:

- sort compatibility requirements are fulfilled by tableau rules' prefix requirements
- the substitution turns all the prefix equations into identities by construction
- irreflexivity of the reduction ordering is also by construction

Each tableau branch represents one atomic path; each closed branch has a complementary connection, and all branches of a closed tableau cover all atomic paths. QED.

Note Our implementation does not use tableaux in any way, we refer to them only for the sake of the proof(-sketch).

[Cerrito and Mayer, 1997] give an upper bound for multiplicity function for a number of propositional modal logics, including S4. We tried to implement it, but it proved to be impractically high.

4.6 Prefix Unification for $\mathsf{S4}^{\mathsf{J}}_n$

[Otten and Kreitz, 1996a] present a prefix unification algorithm for a number of logics, including S4 (the algorithm is the same, but it uses different sets of rules for different logics).

We consider strings over $\mathcal{V} \cup \mathcal{C}$ where $\mathcal{V}$ serves as a set of variables and $\mathcal{C}$ is a set of constants. Symbols are actually positions, so for the $\mathsf{S4}^{\mathsf{J}}_n$ case, we assume that each symbol has a sort : $\mathcal{V} \cup \mathcal{C} \rightarrow \mathbb{N}$, which we denote as a superscript, and we also have a sort compatibility relation $\prec$ over symbols.

Definition 13 (T-string property) *Two strings, a and b, over an alphabet $\mathcal{A}$ have the* **T-string property** *iff*

(A) $\forall_{1 \leq i,j \leq |s|}.i \neq j \rightarrow s_i \neq s_j$ *and* $\forall_{1 \leq i,j \leq |t|}.i \neq j \rightarrow t_i \neq t_j$

(B) $\exists_{0 \leq k \leq \min(|s|,|t|)}.(\forall_{1 \leq i \leq k}.s_i = t_i \& \forall_{k < i \leq |s|} \forall_{k < j \leq |t|}.s_i \neq t_j)$

In other words, these strings form a tree.

Definition 14 *A set of strings S has the* **T-string property** *iff all pairs of strings $s, t \in S$ have the T-string property.*

Definition 15 *A system of equations* $\{s_1 = t_1, \ldots, s_n = t_n\}$ *has the T-string property iff the set* $\{s_1, \ldots, s_n, t_1, \ldots, t_n\}$ *has the T-string property.*

Definition 16 *A substitution* σ *is a* **T-unifier** *of a system of equations* $\{s_i = t_i | 1 \leq i \leq n\}$ *iff* $\sigma s_i = \sigma t_i$ *for all* $1 \leq i \leq n$.

Definition 17 *A substitution* σ *is an* **instance** *of* τ *if there is* δ *such that* $\sigma = \delta \circ \tau$ *(*δ *applied after* τ*).*

Definition 18 *A set of substitutions* Σ *is a* **(minimal) set of most general unifiers** *for a system of equations* Γ *iff:*

Correctness *Every* $\sigma \in \Sigma$ *is a T-unifier for* Γ.

Completeness *Every T-unifier* τ *for* Γ *is an instance of some* $\sigma \in \Sigma$.

Minimality *No* $\sigma \in \Sigma$ *is an instance of another* $\sigma' \in \Sigma$.

A nondeterministic version of the unification algorithm is the following. It assumes that each equation in Γ has the form $s = r|t$ where $r = \varepsilon$ at the beginning:

while $\Gamma \neq \emptyset$ **do**

select the left-most system $s = t \in \Gamma$

set Γ to $\Gamma \setminus \{s = t\}$

select a transformation rule $\Pi \rightarrow \Delta, \Theta$ from $\mathcal{T}$ which is applicable to $s = t$; if no rule is applicable, stop with failure

apply rule $\Pi \rightarrow \Delta, \Theta$ to $s = t$, let Δ', Θ' be the results

$\Gamma := \Theta'(\Gamma)$, $\sigma := \Theta(\sigma)$

$\Gamma := \Delta' \cup \Gamma$, $\sigma = \sigma \cup \Theta'$, trivial equation is not added back to the system

stop with success and return σ.

The set of most general unifiers is the result of all successful runs of the algorithm.

Definition 19 *For two variable positions a^i and b^j, we say that $a^i \simeq b^j$ iff $i = j$ or one of them is 0.*

Corollary 2 *$\simeq$ is an equivalence relation.*

If $a^i \simeq b^j$, then the least constraining sort of variables that can be substituted for *both* a^i and b^j is $\max(i, j)$. Indeed, if $\max(i, j)$ is positive, then it is the only possible sort; if it is 0, then it can be replaced with any other sort, but if we choose any positive sort, we will then be bound to it.

The original list of rules for T-unification for S4 described in [Otten and Kreitz, 1996a] changes to the one described in Table 4.1. To that we added Rules 11 through 15, and sort compatibility conditions to Rules 7 through 10.

Corollary 3 (T-String Preserving) *Let Γ be a system of equations with the T-string property, and σ be a most general unifier for one of its equations. Then a system of equations $\sigma\Gamma$ still has the T-string property.*

R1	$\{\varepsilon = \varepsilon \mid \varepsilon\}$	$\rightarrow \{\}, \{\}$
R2	$\{\varepsilon = \varepsilon \mid t^+\}$	$\rightarrow \{t^+ = \varepsilon \mid \varepsilon\}, \{\}$
R3	$\{Xs = \varepsilon \mid Xt\}$	$\rightarrow \{s = \varepsilon \mid t\}, \{\}$
R4	$\{Cs = \varepsilon \mid Vt\}$	$\rightarrow \{Vt = \varepsilon \mid Cs\}, \{\}$
R5	$\{Vs = z \mid \varepsilon\}$	$\rightarrow \{s = \varepsilon \mid \varepsilon\}, \{V \setminus z\}$
R6	$\{Vs = \varepsilon \mid C_1 t\}$	$\rightarrow \{s = \varepsilon \mid C_1 t\}, \{V \setminus \varepsilon\}$
R7	$\{Vs = z \mid C_1 C_2 t\}$	$\rightarrow \{s = \varepsilon \mid C_2 t\}, \{V \setminus zC_1\}$, where $V \preceq C_1$
R8	$\{Vs^+ = \varepsilon \mid V_1 t\}$	$\rightarrow \{V_1 t = V \mid s^+\}, \{\}$, where $V_1 \preceq V$
R9	$\{Vs^+ = z^+ \mid V_1 t\}$	$\rightarrow \{V_1 t = V' \mid s^+\}, \{V \setminus z^+ V'\}$, where $V \simeq V_1$, and $\mathrm{sort}\{V'\} = \max(\mathrm{sort}\{V\}, \mathrm{sort}\{V_1\})$
R10	$\{Vs = z \mid Xt\}$	$\rightarrow \{Vs = zX \mid t\}, \{\}$, where $V \preceq X$, $V \neq X$, and $s = \varepsilon$ or $t \neq \varepsilon$ or $X \in \mathcal{C}$
R11	$\{Vs = z \mid V_1 t\}$	$\rightarrow \{s = \varepsilon \mid V_1 t\}, \{V \setminus z\}$, where ($z \neq \varepsilon$ and $V \not\simeq V_1$) or ($z = \varepsilon$ and not $V \simeq V_1$)
R12	$\{V = \varepsilon \mid V_1 t\}$	$\rightarrow \{V = \varepsilon \mid t\}, \{V_1 \setminus \varepsilon\}$, where $V_1 \preceq V$ but $V \not\preceq V_1$
R13	$\{V = \varepsilon \mid V_1 t\}$	$\rightarrow \{V_1 t = V' \mid \varepsilon\}, \{V \setminus V'\}$, where $V_1 \preceq V$ but $V \not\preceq V_1$, and $\mathrm{sort}\{V'\} = \mathrm{sort}\{V\}$
R14	$\{Vs = zV_1 \mid Ct\}$	$\rightarrow \{s = \varepsilon \mid Ct\}, \{V \setminus zV_1\}$, where $V \not\preceq C$
R15	$\{Vs = z^+ \mid V_1 t\}$	$\rightarrow \{Vs = z^+ \mid t\}, \{V_1 \setminus \varepsilon\}$, where $V \not\preceq V_1$

Table 4.1: s, t, and z denote (arbitrary) strings, and s^+, t^+, z^+ non-empty strings. X, V, V_1, C, C_1, and C_2 denote single characters with $X \in \mathcal{V} \cup C \cup \mathcal{V}'$, $V, V_1 \in \mathcal{V} \cup \mathcal{V}'$ (with $V \neq V_1$), and $C, C_1, C_2 \in \mathcal{C}$. $V' \in \mathcal{V}'$ is a new variable which does not occur in the substitution σ computed so far.

Definition 20 *Let* mgu(Γ) *be the set of all substitutions produced by the given algorithm with the given set of rules.*

Corollary 4 (Correctness) *Let* $\Gamma = \{s = \varepsilon | t\}$ *be an equation of T-strings. Then all* $\sigma \in$ mgu(Γ) *are T-unifiers for* Γ.

Proof The proof is by induction on the number of rule applications needed for the algorithm to terminate, as in [Otten and Kreitz, 1996a].

Suppose that σ was obtained by a certain sequence of rule applications $r_1, \ldots, r_n$ from system Γ. When $n = 0$ we have a base case, a trivial system of equations, and any substitution is a unifier.

By the induction hypothesis, $r_2, \ldots, r_n$ produces a unifier σ' for $\Gamma' = r_1(\Gamma)$. r_1 induces a substitution τ and $\sigma = \sigma' \cup \tau$. By simple investigation of rules R1-R15, we notice that if σ' is a unifier for Γ', then $\sigma = \sigma' \cup \tau$ is a unifier for Γ. QED.

Theorem 4 (Termination) *Let* $\Gamma = \{s = \varepsilon | t\}$ *be an equation of T-strings. Then the given algorithm with the given set of rules always terminates.*

Proof [Otten and Kreitz, 1996a] prove it by defining an ordering on equations and showing that we need at most two rule applications to obtain a smaller equation with respect to that ordering. This proof works for our set of rules as well.

For a pair of equations $e_1 = \{s_1 = z_1 | t_1\}$ and $e_2 = \{s_2 = z_2 | t_2\}$ we

define $<_e$:

$$e_1 <_e e_2 = \begin{cases} |s_1 z_1 t_1| < |s_2 z_2 t_2|, \text{ or} \\ |s_1 z_1 t_1| = |s_2 z_2 t_2| \text{ and } |z_1| > |z_2|, \text{ or} \\ e_1 = \{\} \text{ and } e_2 = \{\varepsilon = \varepsilon | \varepsilon\}. \end{cases}$$

For rules R1-R10, [Otten and Kreitz, 1996a] show that we need at most two rule applications to obtain a smaller equation with respect to $<_e$.

It is easy to see that rules R11-R15 produce $<_e$-smaller equations.

The equation $\{\}$ is smaller than any solvable equation, hence there is no infinite sequence of applicable rules (any sequence would either produce $\{\}$ or fail). Thus, the algorithm eliminates all equations one by one or stop with failure. QED.

Unlike us, [Otten and Kreitz, 1996a] claim minimality of the resulting unifier sets for their S4-set of rules. See our practical considerations at the end of this section.

Theorem 5 (Completeness) *Let* $\Gamma = \{s = \varepsilon | t\}$ *be an equation of T-strings.* mgu(Γ) *is complete, i.e., any unifier of* Γ *is an instance of some* $\sigma \in$ mgu(Γ).

Proof [Otten and Kreitz, 1996a] claim completeness of the original S4-set of rules. Completeness of our extended set can be obtained by lengthy

analysis of all cases when S4-rules are not applicable due to modal sort compatibility requirements.

Suppose we have an equation

$$\{Vs = z|Xt\}$$

and S4-rules are not applicable due to sort incompatibilities.

For the case when X is a constant, and V cannot absorb it, we have to use substitution $\{V \setminus z\}$ and start to accumulate new substitution. For this case, we have two subcases:

- $z = xC$; rule R7 was applicable one step earlier, which covers this case
- $z = xV_1$; rule R14 covers this case

This reasoning covers the cases when rule R7 is not applicable and, partly, when rule R10 is not applicable.

Now, suppose X is a variable V_1 and V cannot absorb V_1 ($V \not\trianglelefteq V_1$). We have the following options:

- We already accumulated some substitution for V ($z \neq \varepsilon$). We can then stop accumulating the substitution for V, set $\{V \setminus z\}$ and proceed with a new equation $\{s = \varepsilon|t\}$.

  As an alternative, we can set $\{V_1 \setminus \varepsilon\}$ and continue to accumulate the substitution for V, this is covered by rule R15

- V has not accumulated anything ($z = \varepsilon$). If V_1 can absorb V ($V_1 \preceq V$), then the cases when V_1 actually absorbs V are covered by rules R8 and R13. If V_1 can absorb V but actually does not, then we have to set $\{V_1 \setminus \varepsilon\}$ by rule R12.

  Alternatively, V and V_1 are incompatible in both directions and this is covered by the second case of the rule R11.

QED.

4.6.1 Practical considerations

Informally, T-unification problems are not exactly isomorphic to desired solutions of prefix equations. For example, fresh variables introduced in Rules 9 and 13 are set to ε after a unifier is found. Substitution $\{V_1 \setminus \varepsilon, V_2 \setminus \varepsilon\}$ is an instance of $\{V_1 \setminus V_2\}$; for proof search purposes they are equally good, but if we still have some equations to solve, the former substitution being applied to them produces a smaller problem. These considerations suggest that minimality and completeness of the resulting unifier sets do not guarantee optimality.

We must also pay close attention to the order in which rules are attempted – swapping Rules 12 and 13 changes running time on certain problems by more than an order of magnitude, and changes (practical) feasibility of some problems. We plan to measure the "popularity" of the rules on a large base of problems and reorder them by decreasing order of popularity.

4.7 From Matrix Proofs to Sequent Proofs

The algorithm for the conversion of matrix proofs to sequent proofs was given by Kreitz and Schmitt [Kreitz and Schmitt, 2000; Schmitt, 1999] for a number of modal logics (K,K4,T,S4,D,D4), first-order intuitionistic logic, and fragments of linear logic.

The full description of the algorithm is lengthy and very technical, so we will highlight only the main idea and details that change with our shift from S4 to $\mathsf{S4}^{\mathsf{J}}_n$.

The algorithm traverses over the formula tree starting from its root and converts each position into an inference rule (or several inference rules); the result is a derivation tree. Suppose the original formula that we had to prove was $A \vee B$ (or $\Rightarrow A \vee B$ as a sequent). The primary type of the root is α and the main connective is $\vee$; the algorithm chooses this rule:

$$\frac{\Gamma \Rightarrow A, \Delta \qquad \Gamma \Rightarrow B, \Delta}{\Gamma \Rightarrow A \vee B, \Delta}$$

Positions for A and B become *open* and the algorithm chooses the next position among open ones.

In general, inference rules are not permutable because certain rules can destroy formulae of a sequent that have not yet been used, hence the order in which the algorithm has to explore the formula tree is important. The primary factor is the reduction ordering induced by unifying substitution.

Definition 21 *We say that the processed positions are* **solved**, *and that* P_a *is the set of solved atomic positions.*

Definition 22 *We call* **open** *a not-solved position whose immediate predecessor in the tree ordering is solved.* $P_o = \{x \mid x \text{ not solved } \& \ y \prec x \text{ solved}\}$ *is the set of open positions.*

Basically, open positions represent current formulas in the sequent.

Definition 23 *If* $x \sqsubset y$ *(substitution induced relation), we say that* x *is blocking* y. *And* $W_y = \{x \mid x \sqsubset y\}$ *is the set of positions blocking* y.

Open positions that are blocked will not be processed until there is no blocking. There are more requirements for first-order S4 in [Kreitz and Schmitt, 2000; Schmitt, 1999]; unfortunately, they do not fully agree.

Note Since generated proofs are always checked by the MetaPRL engine, it is not critical whether the algorithm has bugs and/or is incomplete. There are two worst-case scenarios: the algorithm runs into a deadlock because of these blocking conditions, or it produces an incorrect proof that will be rejected by MetaPRL.

Consider the S4-rule for $\Box$-introduction to the right:

$$\frac{\Gamma^* \Rightarrow \Delta, A}{\Gamma \Rightarrow \Delta, \Box A} \qquad (\Rightarrow \Box),$$

where $\Gamma^* = \{\Box X | \Box X \in \Gamma\}$. It is convenient to formulate with the $(^*)$ operation when we build a top-down proof.

If we apply this rule too early, the $(^*)$ operation might delete some formulae that we will need later in our proof. The conversion algorithm has a complicated condition describing when it is not too early to use this rule.

For $\mathsf{S4}^{\mathsf{J}}_n$, the similar rule is:

$$\frac{\Gamma^* \Rightarrow \Delta, A}{\Gamma \Rightarrow \Delta, \Box A} \qquad (\Rightarrow \Box)$$

If $\Box$ is J, then $\Gamma^* = \{\mathsf{J}A | \mathsf{J}A \in \Gamma\}$.

If $\Box$ is K_i, then $\Gamma^* = \{\mathsf{J}A | \mathsf{J}A \in \Gamma\} \cup \{\mathsf{K}_i A | \mathsf{K}_i A \in \Gamma\}$.

So in the case of $\mathsf{S4}^{\mathsf{J}}_n$, not all boxes survive after this rule — only boxes of the same sort or stronger (i.e. J). In order to reflect this in the proof conversion algorithm for $\mathsf{S4}^{\mathsf{J}}_n$, we impose one more condition when a π-position cannot be processed — *if there is an open ν-position of a different non-zero sort*. Otherwise, if we process that π-position, we will apply the $\Rightarrow \Box$-rule to it and it will delete the incompatible ν-positions.

This modification is the only one needed to add $\mathsf{S4}^{\mathsf{J}}_n$ support to the existing proof conversion algorithm for $\mathsf{S4}$.

4.8 Muddy Children Puzzle

The paradigmatic example of $\mathsf{S4}^{\mathsf{J}}_n$ application is the renowned:

Muddy Children Puzzle There are n children playing in a yard; as they play m of them become muddy. Their father enters the yard and says: "At least one of you is muddy, do you know whether or not you are muddy?" Nobody answers. The father then asks the following question, over and over: "Do any of you know if you have mud on your forehead?"

Assume it is common knowledge that all the children are perceptive, intelligent, truthful, and that they answer simultaneously. What will happen?

Let us formalize this puzzle in $\mathsf{S4}^{\mathsf{J}}_n$ for four children, two of whom are muddy. The propositional letter c_i will stand for "the i^{th} child is muddy." We will use the formalization that we heard from Melvin Fitting.

Definition 24 *The i^{th} child knows whether or not A is true:*

$$KW_iA = \mathsf{K}_iA \vee \mathsf{K}_i\neg A.$$

Definition 25 *It is common knowledge that every child knows whether or not the other children are muddy:*

$$\begin{aligned} KAO = \mathsf{J}(&KW_1c_2 \wedge KW_1c_3 \wedge KW_1c_4 \wedge KW_2c_1 \wedge KW_2c_3 \wedge KW_2c_4 \wedge \\ &KW_3c_1 \wedge KW_3c_2 \wedge KW_3c_4 \wedge KW_4c_1 \wedge KW_4c_2 \wedge KW_4c_3.) \end{aligned}$$

Definition 26 *The initial state before the father spoke was*

$$s_0 = c_1 \wedge c_2 \wedge \neg c_3 \wedge \neg c_4.$$

Definition 27 *The father's statement "At least one of you is muddy" is formalized as*

$$s_1 = \mathsf{J}(c_1 \vee c_2 \vee c_3 \vee c_4).$$

Definition 28 *Some child knows whether or not (s)he is muddy:*

$$SK = KW_1c_1 \vee KW_2c_2 \vee KW_3c_3 \vee KW_4c_4.$$

Theorem 6 *After the father asked his question the first time, nobody answered. Muddy children 1 and 2 know that they are muddy by the father's second question:*

$$s_0, \mathsf{J}(\neg(s_1 \rightarrow SK)), KAO \Rightarrow \mathsf{K}_1c_1$$

and

$$s_0, \mathsf{J}(\neg(s_1 \rightarrow SK)), KAO \Rightarrow \mathsf{K}_2c_2$$

- $\mathsf{J}s_1 \rightarrow SK$ *states that after the first question some child knows if (s)he is muddy.*
- $\mathsf{J}(\neg(s_1 \rightarrow SK))$ *states that it is common knowledge that no child actually answered after the first question.*

Proof The proof was found automatically by our matrix-based prover and verified by MetaPRL. On a P3-700MHz computer, it took approximately 150 seconds in total (but separately for c_1 and c_2). The proof can be found in Appendix A.

4.9 Final Remarks

In this chapter, we presented a matrix-based prover for $\mathsf{S4}_n^{\mathsf{J}}$. It builds cut-free proofs that can be fed to a Realization Algorithm that would recover evidence terms (justifications) for justified common knowledge modality J, and build an $\mathsf{S4}_n\mathsf{LP}$-proof for a realization of the original formula [Artemov, 2004]. It is our immediate goal to finish this Realization Algorithm and connect it to our $\mathsf{S4}_n^{\mathsf{J}}$-prover.

We are exploring ways to improve the performance of our implementation. One way to address it is to implement a non-constant multiplicity function, i.e., to increase multiplicity of generative positions individually, on demand (although it is not obvious how to do it). Perhaps we should move to tableau-based algorithms, though we need sequent proofs as the output for the Realization Algorithm.

We also want to extend our work to other modal logics ($\mathsf{T}_n^{\mathsf{J}}$ and $\mathsf{S5}_n^{\mathsf{J}}$) [Artemov, 2004] and investigate possible connections between the logic of proofs LP (and $\mathsf{T}_n\mathsf{LP}$, $\mathsf{S4}_n\mathsf{LP}$, $\mathsf{S5}_n\mathsf{LP}$) and description logics.

Appendix A

Formal Proof of Muddy Children Puzzle

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_2, c_3, \mathsf{K}_1 c_3, \Gamma_3 \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}\,\mathrm{Ax}}{\Gamma_2, \mathsf{K}_1 c_3, \Gamma_3 \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}\,\Box \Rightarrow \qquad \alpha : \Gamma_2, \mathsf{K}_1 \neg c_3, \Gamma_3 \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}{\Gamma_1, KW_1 c_2, \mathsf{K}_1 c_3 \vee \mathsf{K}_1 \neg c_3, KW_1 c_4, \bigwedge\limits_{i \neq 1, i \neq j} KW_i c_j, KAO \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}\,\vee \Rightarrow}{\Gamma_1, \bigwedge\limits_{i \neq j} KW_i c_j, \mathsf{J}\left(\bigwedge\limits_{i \neq j} KW_i c_j\right) \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}\,\wedge^* \Rightarrow$$

$$\dfrac{c_1, c_2, \mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{J}\left(\bigwedge\limits_{i \neq j} KW_i c_j\right) \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}{c_1, c_2, \neg c_3, \neg c_4, \mathsf{J}(\neg(s_1 \rightarrow SK)), KAO \Rightarrow \mathsf{K}_1 c_1}\,\neg^* \Rightarrow$$

$$\dfrac{c_1, c_2, \neg c_3, \neg c_4, \mathsf{J}(\neg(s_1 \rightarrow SK)), KAO \Rightarrow \mathsf{K}_1 c_1}{c_1 \wedge c_2 \wedge \neg c_3 \wedge \neg c_4, \mathsf{J}(\neg(s_1 \rightarrow SK)), KAO \Rightarrow \mathsf{K}_1 c_1}\,\wedge^* \Rightarrow$$

then

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2, \mathsf{K}_1\neg c_3, c_4, \mathsf{K}_1 c_4, \Gamma_4 \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}\,\mathrm{Ax}}{\Gamma_2, \mathsf{K}_1\neg c_3, \mathsf{K}_1 c_4, \Gamma_4 \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}\,\Box\Rightarrow \qquad \beta : \Gamma_2, \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, \Gamma_4 \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}{\alpha : \Gamma_2, \mathsf{K}_1\neg c_3, \mathsf{K}_1 c_4 \vee \mathsf{K}_1\neg c_4, \bigwedge\limits_{i\neq 1, i\neq j} KW_i c_j, KAO \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}\,\vee\Rightarrow$$

then

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_7, c_3, \mathsf{K}_1 c_3, \Gamma_6 \Rightarrow \Delta_1, c_3}\,\mathrm{Ax}}{\Gamma_7, \mathsf{K}_1 c_3, \Gamma_6 \Rightarrow \Delta_1, c_3}\,\Box\Rightarrow \qquad \gamma : \mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1\neg c_3, \Gamma_8, \mathsf{K}_1\neg c_3, \Gamma_6 \Rightarrow \Delta_1, c_3}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, KW_1 c_1, KW_1 c_2, KW_1 c_3, \Gamma_6 \Rightarrow \Delta_1, c_3}\,\vee\Rightarrow}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \neg c_3, \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, KW_1 c_1, KW_1 c_2, KW_1 c_3, \Gamma_6 \Rightarrow \Delta_1}\,\neg\Rightarrow}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, KW_{i\in\{1,2\}} c_{j\neq i}, \bigwedge\limits_{i=3,4\ i\neq j} KW_i c_j, KAO, s_0 \Rightarrow \Delta_1}\,\Box\Rightarrow}{\Gamma_5, \bigwedge\limits_{i\neq j} KW_i c_j, KAO, s_0 \Rightarrow \Delta_1}\,\wedge\Rightarrow}{\Gamma_5, KAO, s_0 \Rightarrow c_1, c_4, KW_1 c_1, \mathsf{K}_2 c_2, \mathsf{K}_2\neg c_2, KW_3 c_3 \vee KW_4 c_4}\,\Box\Rightarrow}{\Gamma_5, KAO, s_0 \Rightarrow c_1, c_4, \bigvee\limits_i KW_i c_i}\,\Rightarrow\vee^*}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, KAO \Rightarrow c_1, c_4, s_1 \rightarrow SK}\,\Rightarrow\rightarrow}{\neg(s_1 \rightarrow SK), \mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, KAO \Rightarrow c_1, c_4}\,\neg\Rightarrow}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, KAO \Rightarrow c_1, c_4}\,\Box\Rightarrow}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1\neg c_3, \neg c_4, \mathsf{K}_1\neg c_4, KAO \Rightarrow c_1}\,\neg\Rightarrow}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, KAO \Rightarrow c_1}\,\Box\Rightarrow}{\beta : c_1, c_2, \mathsf{J}(\neg(s_1 \rightarrow SK)), KW_1 c_2, \mathsf{K}_1\neg c_3, \mathsf{K}_1\neg c_4, \bigwedge KW_i c_j, KAO \Rightarrow \mathsf{K}_1 c_1, c_3, c_4}\,\Rightarrow\Box$$

then

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_{11}, c_4, \mathsf{K}_2 c_4, \Gamma_{10} \Rightarrow c_1, c_4, \Delta_2, c_3, c_3}\,\mathrm{Ax}}{\Gamma_{11}, \mathsf{K}_2 c_4, \Gamma_{10} \Rightarrow c_1, c_4, \Delta_2, c_3, c_3}\,\Box \Rightarrow \qquad \delta : \Gamma_{11}, \mathsf{K}_2 \neg c_4, \Gamma_{10} \Rightarrow \Delta_3}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1 \neg c_3, \Gamma_8, \mathsf{K}_1 \neg c_3, \Gamma_9, KW_2 c_4, \Gamma_{10} \Rightarrow \Delta_1, c_3, c_3}\,\vee \Rightarrow}{\mathsf{J}(\neg(s_1 \rightarrow SK)), \neg c_3, \mathsf{K}_1 \neg c_3, \Gamma_8, \mathsf{K}_1 \neg c_3, \Gamma_6 \Rightarrow \Delta_1, c_3}\,\neg \Rightarrow}{\gamma : \mathsf{J}(\neg(s_1 \rightarrow SK)), \mathsf{K}_1 \neg c_3, \Gamma_8, \mathsf{K}_1 \neg c_3, \Gamma_6 \Rightarrow \Delta_1, c_3}\,\Box \Rightarrow$$

then

$$\dfrac{\dfrac{\dfrac{}{\Gamma_{12}, c_3, \mathsf{K}_2 c_3, \mathsf{K}_2 \neg c_4, \Gamma_{10} \Rightarrow \Delta_3}\,\mathrm{Ax}}{\Gamma_{12}, \mathsf{K}_2 c_3, \mathsf{K}_2 \neg c_4, \Gamma_{10} \Rightarrow \Delta_3}\,\Box \Rightarrow \qquad \varepsilon : \Gamma_{12}, \mathsf{K}_2 \neg c_3, \mathsf{K}_2 \neg c_4, \Gamma_{10} \Rightarrow \Delta_3}{\delta : \Gamma_{12}, KW_2 c_3, \mathsf{K}_2 \neg c_4, \Gamma_{10} \Rightarrow \Delta_3}\,\vee \Rightarrow$$

then

$$\dfrac{\dfrac{\dfrac{}{\Gamma_{13}, c_1, \mathsf{K}_2 c_1, \Gamma_{14} \Rightarrow c_1, \Delta_4}\,\mathrm{Ax}}{\Gamma_{13}, \mathsf{K}_2 c_1, \Gamma_{14} \Rightarrow \Delta_3}\,\Box \Rightarrow \qquad \zeta : \Gamma_{13}, \mathsf{K}_2 \neg c_1, \Gamma_{14} \Rightarrow \Delta_3}{\varepsilon : \Gamma_{13}, KW_2 c_1, \Gamma_{14} \Rightarrow \Delta_3}\,\vee \Rightarrow$$

then

$$
\begin{array}{cl}
\text{propositional reasoning} & \\
\hline
\Gamma_{18}, c_1 \vee c_2 \vee c_3 \vee c_4, \mathsf{J}(c_1 \vee c_2 \vee c_3 \vee c_4), s_1 \Rightarrow c_2, c_1, SK, c_3, c_4 & \\
\hline
 & \Box\Rightarrow \\
\Gamma_{18}, \mathsf{J}(c_1 \vee c_2 \vee c_3 \vee c_4), s_1 \Rightarrow c_2, c_1, SK, c_3, c_4 & \\
\hline
 & \text{(renaming)} \\
\Gamma_{17}, \mathsf{K}_2\neg c_4, KAO, s_1, s_1 \Rightarrow c_2, c_1, SK, c_3, c_4 & \\
\hline
 & \neg\Rightarrow \\
\Gamma_{17}, \neg c_4, \mathsf{K}_2\neg c_4, KAO, s_1, s_1 \Rightarrow c_2, c_1, SK, c_3 & \\
\hline
 & \Box\Rightarrow \\
\mathsf{J}(\neg(s_1 \to SK)), \mathsf{K}_2\neg c_1, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, KAO, s_1, s_1 \Rightarrow c_2, c_1, SK, c_3 & \\
\hline
 & \neg\Rightarrow \\
\mathsf{J}(\neg(s_1 \to SK)), \mathsf{K}_2\neg c_1, \neg c_3, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, KAO, s_1, s_1 \Rightarrow c_2, c_1, SK & \\
\hline
 & \Box\Rightarrow \\
\mathsf{J}(\neg(s_1 \to SK)), \mathsf{K}_2\neg c_1, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, KAO, s_1, s_1 \Rightarrow c_2, c_1, SK & \\
\hline
 & \Rightarrow\to \\
\mathsf{J}(\neg(s_1 \to SK)), \mathsf{K}_2\neg c_1, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, KAO, s_1 \Rightarrow c_2, c_1, s_1 \to SK & \\
\hline
 & \neg\Rightarrow \\
\neg(s_1 \to SK), \mathsf{J}(\neg(s_1 \to SK)), \mathsf{K}_2\neg c_1, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, KAO, s_1 \Rightarrow c_2, c_1 & \\
\hline
 & \Box\Rightarrow \\
\mathsf{J}(\neg(s_1 \to SK)), \mathsf{K}_2\neg c_1, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, KAO, s_1 \Rightarrow c_2, c_1 & \\
\hline
\mathsf{J}(\neg(s_1 \to SK)), \neg c_1, \mathsf{K}_2\neg c_1, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, KAO, s_1 \Rightarrow c_2 \neg\Rightarrow & \\
\hline
 & \Box\Rightarrow \\
\mathsf{J}(\neg(s_1 \to SK)), \mathsf{K}_2\neg c_1, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, KAO, s_1 \Rightarrow c_2 & \\
\hline
 & \Rightarrow\Box \\
\zeta : \mathsf{J}(\neg(s_1 \to SK)), \mathsf{K}_1\Gamma_{15}, \mathsf{K}_2\neg c_1, \mathsf{K}_2\neg c_3, \mathsf{K}_2\neg c_4, \Gamma_{16}, KAO, s_1 \Rightarrow \Delta_4, \mathsf{K}_2 c_2, \Delta_5 &
\end{array}
$$

Bibliography

[Aczel and Rathjen, 2001] Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical Report 40, Mittag-Leffler, 2001.

[Aczel, 1986] Peter Aczel. The type theoretic interpretation of constructive set theory: Inductive definition. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers, 1986.

[Aczel, 1999] Peter Aczel. On relating type theories and set theories. In Naraschewski Altenkirch and Reus, editors, *Proceedings of Types for Proofs and Programs 98*, volume 1657 of *Lecture Notes in Computer Science*, 1999.

[Alberucci and Jäger, 2005] Luca Alberucci and Gerhard Jäger. About cut elimination for logics of common knowledge. *Annals of Pure and Applied Logic*, 133(1-3):73–99, 2005.

[Andrews, 1981] Peter B. Andrews. Theorem proving via general matings. *JACM*, 28(2):193–214, 1981.

[Antonakos, 2006] E. Antonakos. Comparing justified and common knowledge. *The Bulletin of Symbolic Logic*, 12, 2006. in: *2005 Summer Meeting of the ASL.*

[Artemov and Nogina, 2004] Sergei Artemov and Elena Nogina. Logic of knowledge with justifications from the provability perspective. Technical Report TR-2004011, CUNY Ph.D. Program in Computer Science, 2004.

[Artemov and Nogina, 2005a] Sergei Artemov and Elena Nogina. Introducing justification into epistemic logic. *Journal of Logic and Computation*, 15(6):1059–1073, 2005.

[Artemov and Nogina, 2005b] Sergei Artemov and Elena Nogina. On epistemic logic with justification. In R. van der Meyden, editor, *Theoretical Aspects of Rationality and Knowledge. Proceedings of the Tenth Conference (TARK 2005), June 10-12, 2005, Singapore.*, pages 279–294. National University of Singapore, 2005.

[Artemov, 1994] Sergei Artemov. Logic of proofs. *Annals of Pure and Applied Logic*, 67(1):29–59, 1994.

[Artemov, 1995] Sergei Artemov. Operational modal logic. Technical Report MSI 95-29, Cornell University, 1995.

[Artemov, 2001] Sergei Artemov. Explicit provability and constructive semantics. *The Bulletin for Symbolic Logic*, 6(1):1–36, 2001.

[Artemov, 2004] Sergei Artemov. Evidence-based common knowledge. Technical Report TR-2004018, CUNY Ph.D. Program in Computer Science Technical Reports, November 2004.

[Artemov, 2005] Sergei Artemov. Evidence-based common knowledge. Technical Report TR-2004018, revised version, CUNY Ph.D. Program in Computer Science, 2005.

[Aumann, 1976] Robert J. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6):1236–1239, 1976.

[Banerjee, 1988] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Press, Boston, (Mass.), 1988.

[Barendregt and Geuvers, 2001] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1149–1238. Elsevier, 2001.

[Barendregt, 1992] Henk P. Barendregt. *Handbook of Logic in Computer Science*, volume 2, chapter Lambda Calculi with Types, pages 118–310. Oxford University Press, 1992.

[Basin and Wolff, 2003] David Basin and Burkhart Wolff, editors. *Proceedings of the 16^{th} International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Bibel, 1981] Wolfgang Bibel. On matrices with connections. *JACM*, 28(4):633–645, 1981.

[Bibel, 1987] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg Verlag, Braunschweig, 2nd edition, 1987.

[Bledsoe, 1975] W. W. Bledsoe. A new method for proving certain Presburger formulas. *Fourth Intl. Joint Conference on AI,* Tblisi, USSR, September 1975.

[Boulton, 1994] Richard John Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge Computer Laboratory, May 1994. Technical Report 337.

[Bryukhov *et al.*, 2003] Yegor Bryukhov, Alexei Kopylov, Vladimir Krupski, and Aleksey Nogin. Implementing and automating basic number theory in MetaPRL proof assistant. In David Basin and Burkhart Wolff, editors, *16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003). Emerging Trends Proceedings*, pages 29–39. Universität Freiburg, 2003.

[Cerrito and Mayer, 1997] Serenella Cerrito and Marta Cialdea Mayer. Hintikka multiplicities in matrix decision methods for some propositional modal logics. In *Analytic Tableaux and Related Methods (TABLEAUX 97)*, pages 138–152, 1997.

[Chan, 1982] T. Chan. An algorithm for checking PL/CV arithmetic inferences. In G. Goos and J. Hartmanis, editors, *An Introduction to the PL/CV Programming Logic*, volume 135 of *Lecture Notes in Computer Science*, appendix D, pages 227–264. Springer-Verlag, 1982.

[Constable *et al.*, 1986] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.

[Constable, 2002] Robert L. Constable. Naïve computational type theory. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability, Proceedings of International Summer School Marktoberdorf, July 24 to August 5, 2001*, volume 62 of *NATO Science Series III*, pages 213–260, Amsterdam, 2002. Kluwer Academic Publishers.

[Cooper, 1972] D.C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.

[Coquand and Huet, 1988] Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Fagin *et al.*, 1995] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. Massachusetts Institute of Technology, 1995.

[Fischer and Rabin, 1974] M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. In R. M. Karp, editor, *Complexity of Computation*, pages 27–41, Amer. Math. Soc., Providence, RI, 1974.

[Fitting, 2003] Melvin Fitting. A semantics for the logic of proofs. Technical Report TR-2003012, CUNY Ph.D. Program in Computer Science, 2003.

[Fitting, 2004] Melvin Fitting. Semantics and tableaus for lps4. Technical Report TR-2004016, CUNY Ph.D. Program in Computer Science, 2004.

[Fitting, 2005] Melvin Fitting. The logic of proofs, semantically. *Annals of Pure and Applied Logic*, 132(1):1–25, 2005.

[Gettier, 1963] Edmund L. Gettier. Is justified true belief knowledge? *Analysis*, 23:121–123, 1963.

[Gordon and Melham, 1993] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[Gordon *et al.*, 1979] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, NY, 1979.

[Harper *et al.*, 1993] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993. A revised and expanded verion of '87 paper.

[Harrison, 1994] J. Harrison. Binary decision diagrams as a hol derived rule. In T. F. Melham and J. Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 254–268. Springer, Berlin, Heidelberg, 1994.

[Harrison, 1996] J. Harrison. Stalmarck's algorithm as a HOL derived rule. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234, Turku, Finland, August 1996. Springer Verlag.

[Hickey *et al.*,] Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.

[Hickey *et al.*, 2003] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In Basin and Wolff [2003], pages 287–303.

[Hintikka, 1961] Jaakko Hintikka. Modalities and quantification. *Theoria*, 27:119–128, 1961.

[Hintikka, 1962] Jaakko Hintikka. *Knowledge and Belief*. Cornell University Press, Ithaca, NY, 1962.

[Hirschkoff, 1996] Daniel Hirschkoff. *Nodesat*, an arithmetical tactic for the Coq proof assistant. Technical Report 96-61, CERMICS, Noisy-le-Grand, April 1996.

[Howe, 1989] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, Asilomar Conference Center, Pacific Grove, California, June 1989. IEEE, IEEE Computer Society Press.

[INR, 2003] INRIA. *The Coq Proof Assistant Reference Manual*, 2003.

[Jäger *et al.*, 2005] G. Jäger, M. Kretz, and Th. Studer. Cut-free common knowledge. 2005. Submitted for publication.

[Kleene, 1952] S. C. Kleene. *Introduction to Metamathematics*. D. Van Nostrand, Princeton, 1952.

[Kreitz and Otten, 1999] Christoph Kreitz and Jens Otten. Connection-based theorem proving in classical and non-classical logics. *Journal for Universal Computer Science, Special Issue on Integration of Deductive Systems*, 5(3):88–112, 1999.

[Kreitz and Schmitt, 2000] Christoph Kreitz and Stephan Schmitt. A uniform procedure for converting matrix proofs into sequent-style systems. *Journal of Information and Computation*, 162(1–2):226–254, 2000.

[Martin-Löf, 1982] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic,*

Methodology, and Philosophy of Science, pages 153–175, Amsterdam, 1982. North Holland.

[McCarthy *et al.*, 1979] John McCarthy, M. Sato, T. Hayashi, and S. Igarishi. On the model theory of knowledge. Technical Report STAN-CS-79-725, Stanford University, 1979.

[Meyer and van der Hoek, 2004] John-Jules Ch. Meyer and Wiebe van der Hoek. *Epistemic Logic for AI and Computer Science*. Cambridge University Press, 2004.

[Nogin and Hickey, 2002] Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *Lecture Notes in Computer Science*, pages 281–297. Springer-Verlag, 2002.

[Norrish, 2003] Michael Norrish. Complete integer decision procedures as derived rules in HOL. In Basin and Wolff [2003], pages 71–86.

[Oppen, 1973] Derek C. Oppen. Elementary bounds for Presburger arithmetic. In ACM, editor, *Conference record of Fifth Annual ACM Symposium on Theory of Computing: papers presented at the Symposium, Austin, Texas, April 30–May 2, 1973*, pages 34–37, New York, NY, USA, 1973. ACM Press.

[Otten and Kreitz, 1996a] Jens Otten and Christoph Kreitz. T-string-unification: Unifying prefixes in non-classical proof methods. In U. Moscato, editor, *5^{th} Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, volume 1071 of *Lecture Notes in Artificial Intelligence*, pages 244–260. Springer Verlag, 1996.

[Otten and Kreitz, 1996b] Jens Otten and Christoph Kreitz. A uniform proof procedure for classical and non-classical logics. In G. Görz and S. Hölldobler, editors, *KI-96: Advances in Artificial Intelligence*, volume 1137 of *Lecture Notes in Artificial Intelligence*, pages 307–319. Springer Verlag, 1996.

[Owre and Shankar, 1997] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[Parikh and Ramanujam, 1985] Rohit Parikh and R. Ramanujam. Distributed processes and the logic of knowledge. In *Proceedings of the Conference on Logic of Programs*, pages 256–268, London, UK, 1985. Springer-Verlag.

[Parikh, 1987] Rohit Parikh. Knowledge and the problem of logical omniscience. In *Methodologies for Intelligent Systems*, pages 432–439, 1987.

[Parikh, 1995] R. Parikh. Logical omniscience. In *Logic and Computational Complexity. International Workshop LCC '94. Selected Papers. Springer-Verlag. 1995*, pages 22–9, 1995.

[Paulin-Mohring, 1993] Christine Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In J. F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[Paulson and Nipkow, 1990] L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge Computing Laboratory, 1990.

[Pfenning and Schuermann, 2002] Frank Pfenning and Carsten Schuermann. *Twelf User's Guide, Version 1.4*, December 2002.

[Pfenning, 1994] Frank Pfenning. Elf: A meta-language for deductive systems. In *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 811– 815, Nancy, France, June 1994. Springer-Verlag.

[Presburger, 1929] M. Presburger. Über de vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchen, die addition als einzige operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématicienes des Pays Slaves*, pages 92–101. Warsaw, 1929.

[Pugh, 1991] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[Reddy and Loveland, 1978] C. R. Reddy and D. W. Loveland. Presburger arithmetic with bounded quantifier alternation. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 320–325. ACM Press, 1978.

[Rudnicki, 1992] Piotr Rudnicki. An overview of the Mizar project. Notes to a talk at the workshop on Types for Proofs and Programs, June 1992.

[Schmitt and Kreitz, 1995] Stephan Schmitt and Christoph Kreitz. On transforming intuitionistic matrix proofs into standard-sequent proofs. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods: Proc. of the 4th International Workshop TABLEAUX'95*, pages 106–121. Springer, Berlin, Heidelberg, 1995.

[Schmitt *et al.*, 2001] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 421–426. Springer-Verlag, 2001.

[Schmitt, 1999] Stephan Schmitt. *Proof Reconstruction in Classical and Non-classical Logics*. PhD thesis, Technical University of Darmstadt, 1999.

[Shankar *et al.*, 1999] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[Shostak, 1977] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the Association for Computing Machinery*, 24(4):529–543, October 1977.

[van Benthem, 1991] Johan van Benthem. Reflections on epistemic logic. *Logique et Analyse*, 34(133–134):5–14, 1991.

[Wallen, 1990] Lincoln A. Wallen. *Automated deduction in nonclassical logics*. MIT Press, Cambridge, MA, USA, 1990.