2014

TR-2014001: Computer-Aided Reasoning about Knowledge and Justifications

Natalia Novak

Computer-Aided Reasoning about Knowledge and Justifications.

Dissertation

Natalia Novak

CUNY Graduate Center

June 16, 2009

Contents

Introduction

The study of epistemic reasoning is one of the core areas in Computer Science and Artificial Intelligence. The traditional systems of formal epistemology are based on modal logics and have been subjects of intense research activity during the past decades [Fagin *et al.*, 1995; Meyer and Hoek, 1995]. There are several computer-aided systems of modal and epistemic reasoning available (for an incomplete list, see [Schmidt, 2009]).

A recent foundational effort in this area has enriched modal epistemic logic with the internalized notion of justification, which became part of the language of epistemic logic [Antonakos, 2007; Artemov, 2001; Artemov and Nogina, 2005; Artemov, 2006; Artemov, 2008; Brezhnev, 2000; Brezhnev and Kuznets, 2006; Bryukhov, 2005; Fitting, 2005; Krupski, 2006b; Krupski, 2006a; Kuznets, 2008; Milnikel, 2007; Renne, 2008; Rubtsova, 2006; Yavorskaya (Sidon), 2006]. This development substantially broadens the scope of applications of epistemic logic. We now have the capability to not only reason about epistemic states of knowledge and belief of agents, but also to track their justifications and to sort those which are pertinent to given

facts and sufficient for epistemic conclusions. The very notion of Evidence became the subject of rigorous studies.

Knowledge, belief, and evidence are fundamental notions which appear in a wide range of areas. Among already existing and potential applications of epistemic reasoning with justifications are Artificial Intelligence, Information Security, Game Theory and Economics, Philosophy, Mathematics, Computer-Aided reasoning systems, Law, etc.

This dissertation is the first to build and study computer-aided systems of epistemic reasoning with justifications. The main focus of this dissertation is Artemov's Realization Theorem, which is the fundamental result that reveals the robust evidence system behind traditional epistemic modal logic reasoning. The Realization Theorem recovers evidence terms for each occurrence of epistemic modality in a given theorem. The first version of the realization theorem was established in [Artemov, 1995; Artemov, 2001] and produced evidence terms which could be exponential in the length of a given cut-free proof of the theorem. Further improvements, starting with [Brezhnev and Kuznets, 2006], have lowered this bound significantly and offered a realization algorithm which produces evidence terms quadratic in the size of a given cut-free proof of an epistemic modal theorem. In this dissertation work, we improve and implement Brezhnev-Kuznets' realization algorithm and test its performance on a number of paradigmatic epistemic problems. These tests show robust behavior of realization terms in which complexity stays firmly within theoretically predicted polynomial (quadratic) bounds.

Our implementation of the Realization Algorithm is performed within the framework of the MetaPRL computer-aided reasoning system.

The dissertation consists of two Chapters. In the first Chapter, we make a deep, systematic comparison of the MetaPRL system with the well-known system Coq, which is also based on the idea of using higher-order type systems and constructive reasoning. We implement in MetaPRL the Calculus of Inductive Constructions which is the theoretical base for Coq. This work has shown that the common points of MetaPRL and Coq revealed their principal methodological differences. A projected application of this work is the possibility of performnig re-validation of the existing library of Coq proofs which could help to build more trust in Coq.

Chapter Two contains the main contribution of the dissertation: the implementation of the Realization Algorithm and the results of tests runs on a wide range of well-known epistemic problems.

Chapter 1

Implementation of Calculus of Inductive Constructions in MetaPRL logical framework

MetaPRL is a logical framework that was conceived in an attempt to address scalability problems and other limitations of its predecessor, NuPRL. The most well-developed logic in MetaPRL is a variation of Per Martin-Löf's Intuitionistic Type Theory. MetaPRL is positioned as a logical framework, therefore, it is interesting to see how well it can handle other logics. This work is an attempt to implement the *Calculus of Inductive Constructions* (CIC) in MetaPRL. CIC is the underlying logic of Coq proof assistant, which has an extensive mathematical library and is actively developed in Europe.

1.1 Overview of Coq and Calculus of Inductive Constructions

The Coq proof development system allows to build proofs interactively, and provides various (Coq-)certified decision and semi-decision algorithms [Coq, a]. It includes extensive libraries, has graphical user interface and documentation tools, and can also be used in Proof General Emacs mode.

This system is the result of more than 25 years of research. In 1984, Thierry Coquand and Gérard Huet developed and implemented the *Calculus of Constructions*. In 1991, Christine Paulin-Mohring extended it to the *Calculus of Inductive Constructions* [Paulin-Mohring, 1993]. The system further developed from the efforts of Chet Murthy, Jean-Christophe Filliâtre, Bruno Barras, and Hugo Herbelin. More than thirty other people contributed to the development of specific features. (Many more people contributed and continue to contribute to the system.)

Calculus of Inductive Constructions is based on typed lambda calculus. Without a notion of inductive definition, it is basically λ_{Pw} (aka λ_C) in Barendregt's cube of typed lambda calculi [Barendregt, 1992b] or to Girard's polymorphic Fw system [Girard, 1986] extended with types dependent on terms.

1.2 Overview of MetaPRL logical framework

The MetaPRL [Met, ; Hickey, 2001; Hickey *et al.*, 2003b] project commenced in 1996 with the intention of addressing the concerns of scalability in formal systems, as it was realized that its predecessor, NuPRL, does not really scale to some verification problems.

In MetaPRL, logics are composed from modules, which are constructed incrementally by adding formal properties, e.g., definitions, rules, and theorems [Hickey, 2003; Hickey *et al.*, 2003c]. The empty module is the root of each logic. This construction allows re-use: logics can share a common core with properties that are inherited as they are extended. Relations between logics are constructed by inheritance or explicitly constructed as functions between modules. Modularity of MetaPRL provides the flexibility to support a wide range of applications and restricts the logical search complexity.

Coq also employs modular design to structure its mathematical library; modules there should be compatible with each other, because they belong to the same logic. In MetaPRL, many logics can be developed, therefore modules can be incompatible.

One can interact with MetaPRL either via console or Internet browser. In the latter instance, the user basically has the same console enhanced by menu and structural point-and-click access to proofs and terms.

1.3 Translation

MetaPRL logical framework allows definition of Gentzen-style rule schemas. Coq does not have an explicit formal description of rules; they are implicitly manifested by Coq implementation.

We are aware of only one formalization of Coq rules, which was implemented in Coq itself, [Barras, 1999]. It is not an exact implementation; its treatment of inductive types is a bit different. As you will see later, we also had to deviate because the original rules from the [Coq, b] manual are very complex in some places. Moreover, Coq authors admit that those areas are the primary source of problems, and required tweaking more than once.

We will go over the rules from the Coq manual and present our reimplementation in MetaPRL notation.

First, however, let us briefly discuss MetaPRL rule syntax and semantics. One can find more information on MetaPRL web-site `http://metaprl.org`.

1.3.1 MetaPRL Rule Syntax and Semantics

A rule has roughly the following form:

```
rule Name Arguments :
Sequent1 --> ... SequentN --> Sequent,
```

where `Sequent1`, ..., `SequentN` are assumptions and `Sequent` is a conclusion.

Each sequent has the form

```
sequent { Hyp1; ...; HypN >- Conclusion }.
```

Each hypothesis is either a variable declaration '`v : Term`' or a context '`<H>`.' The first-order variable `v` is bound for the rest of the sequent. The variable declaration can actually omit the variable's name if it is irrelevant and not used elsewhere. Contexts are placeholders for arbitrarily many variable declarations.

The '`Term`' in a variable declaration '`v : Term`' is a term with probably some occurrences of second- and first-order variables.

First-order variables are introduced by variable declarations and terms (lambda abstraction, quantifiers introducing new bound first-order variables). Names of bound first-order variables are prefixed by single quotes to distinguish them from constants and simplify parsing.

Second-order variables are also prefixed with single quotes and can be used in place of a (sub)term. They function as placeholders and, together with contexts, provide a mechanism for defining rule schemas rather than rule instances. Unlike first-order variables, second-order variables' binding scope is the entire rule.

Example:

```
rule A 'a :
sequent{ <H>; x:'T; <J['x]>; <K<|H|> > >- 'a in S<|J|>['x] } -->
sequent{ <H>; x:'T; <J['x]>; <K<|H|> > >- 'S<|J|>['x] in Set },
```

here x is a first-order variable and a, T, and S are second-order variables. Second-order variables are placeholders for arbitrary terms. H, J, and K are contexts, i.e., placeholders for an arbitrary number of first-order variable declarations.

Square bracket notation here means that free occurrences of x are allowed in J, S. No free occurrences of x are allowed in K.

The reverse is true of dependencies on contexts: when no explicit dependency is listed, dependency on all preceding contexts is assumed. J has no explicit context dependency here and so depends on H, not on K, because K is declared after J. K explicitly depends only on H and is not allowed to depend on J. S depends on J only and is not allowed to depend on H and K.

When we say that a context J depends on a variable x, we mean that hypotheses, which are matched with J in a particular instance of the rule at hand, can have free occurrences of x.

When we say that a second-order variable A depends on a context, we mean that in a particular instance of the rule at hand, i.e., where that context is instantiated as $v_1 : T_1, \ldots, v_n : T_n$, A can depend on $v_1, \ldots, v_n$.

When we say that a context Δ depends on a context Γ, we mean that in a particular instance of the rule at hand, in which context Γ is instantiated as $v_1 : T_1, \ldots, v_n : T_n$ and context Δ is instantiated as $w_1 : S_1, \ldots, w_m : S_m$, each of S_i can depend on $v_1, \ldots, v_n$.

In MetaPRL, rules are usually applied backwards, meaning that for rule A from above to be applicable to a given goal, its conclusion sequent has to match the goal. In this case, the goal is replaced with several goals, one for each assumption sequent with all second-order variables replaced by matching terms from the original goal.

If some second-order variables do not occur in the conclusion sequent of the rule, substituting terms have to be provided by the time of rule application. In the case of rule A, a is such a variable and is listed in the rule header to let MetaPRL know which term has to be used for a when the rule is applied.

Also, in situations such as this:

```
rule B 'H 'A :
   sequent { <H>; 'A; <J> >- 'C } -->
   sequent { <H>; <J> >- 'C },
```

MetaPRL has no idea where to split the list of hypotheses in order to insert A, so it has to again be provided by a parameter to the rule (note 'H in the rule header); this parameter is not really a context but a natural number designating how many hypotheses have to be allocated to H. The rest will go to J.

In the following text, we will gradually change from an explicit MetaPRL syntax to a more standard mathematical notation. This is done to give readers a taste of the syntactic capabilities and limitations of MetaPRL, and

to simultaneously conceal trivial details in complex rules that will follow at the end of this chapter. In particular, to indicate a dependency on context, we will write $\Gamma_{I,\langle\Delta\rangle}$ to indicate that Γ_I depends on Δ.

1.3.2 Non-inductive component of Calculus of Inductive Constructions

From this point, we follow closely the structure of the Coq 7.3 manual [Coq, b], and implement almost every rule in MetaPRL. Whenever we refer to a rule by name, we mean a rule from the manual. Later in the text, we switch to a couple of related papers because the manual starts skipping details.

The terms

CIC is a typed lambda calculus, therefore, all well-formed terms have types. We therefore have to declare a term

```
declare member{'t;'T},
```

displayed as $t \in T$, which states that t has type T. One has to distinguish variable declarations $t : T$ in hypotheses from judgments $t \in T$. The former is a fixed syntactic construct, the latter a user-defined term. Traditional CIC syntax uses $t : T$ for both purposes; we will follow the same practice later in the text.

Sorts

Types are terms themselves. Every type belongs to either `Set`, `Prop`, or $\texttt{Type}_i$ type; these types of types are called *sorts*.

```
declare Prop
declare Set
declare "type"[i:l]
```

double quotes are used because MetaPRL is an OCaml extension and 'type' is an OCaml keyword. `[i:l]` means that the term has a non-term parameter i – a level expression, i.e., a composition of successor and maximum operations and natural numbers.

Terms

CIC terms are built from the following rules:

(A) the sorts `Set`, `Prop`, and $\texttt{Type}_i$ are terms;

(B) global names (constants) from the environment are terms;

(C) variables are terms;

(D) $(x : T)U$ is a term if x is a variable, and T and U are terms. This is a dependent product, defined in MetaPRL as `fun{'T; x.'U['x]}` with shorter notation `x:'T -> 'U[x]`;

(E) $[x : T]U$ is a term if x is a variable, T and U are terms. This is a λ-abstraction, MetaPRL syntax is `lambda{'T; x.'U['x]}`;

(F) $(T\ U)$ is a term if T and U are terms. It is an application term, defined in MetaPRL as `apply{'T; 'U}` and short syntax `'T 'U`;

(G) $[x := t]T$ is a term if t and T are terms and x is a variable. This is a let-in construct, MetaPRL syntax is `let_in{'t; x.'T['x]}`.

Typed terms

Coq *local context* is a list of typed variable declarations. Coq has two types of declarations: assumptions $x : T$ and definitions $x := t : T$. The latter is not directly supported by MetaPRL; it also does not appear essential to CIC so we decided not to reproduce it for the present. Note that CIC context is not a fully functional standalone notion because it introduces binding of some variables and needs some destination where this binding will work. MetaPRL's analog of CIC context is called *sequent*, mainly for historical reasons. Sequent is a special kind of term with three components: a label, a binding (list of typed variable declarations), and a term where those declared variables are bound. Since sequent is a term, it can be used anywhere a regular term can be used, in particular it means that sequents can be nested.

CIC rules also make use of *environment*, which is a list of global declaration of constants. 'Global' means that constants added to the environment in one statement (theorem or declaration) survive to the following statements.

While MetaPRL supports some internal analog of Coq environment, it does not allow rules to operate on it explicitly.

The difference between the MetaPRL and Coq notions of environment is

the fact that MetaPRL allows the addition of constants that are not necessarily well-defined from a CIC point of view. On the other hand, CIC ensures that all added terms are well-typed.

Whenever we encounter an assumption of a rule of the form that something is in the environment, we can replace it with the assumption that it is well-typed.

We could, of course, emulate environment as an outer sequent (context) to every judgment, but our goal is to implement CIC in as native a manner as possible without extra layers of abstraction.

This means that some CIC rules cannot be expressed in MetaPRL explicitly and some have to be adjusted.

Typing rules

Coq rules have two types of judgments: $E[\Gamma] \vdash t : T$ and $\mathcal{WF}(\mathsf{E})[\Gamma]$. For the moment, we do not implement the latter. The well-formedness judgment is used in rules of the form:

$$\frac{\mathcal{WF}(\mathsf{E})[\Gamma] \qquad \text{other assumptions}}{E[\Gamma] \vdash \text{some conclusion}}.$$

If we omit the well-formedness assumption, we will allow non-well-formed Γ in the rule's conclusion but we can interpret such sequents as vacuously true, thus not increasing the power of the rule.

It turns out that these two types of judgments are not sufficient, at least on a user level. We certainly want to make claims such as $\Gamma \vdash (A \wedge B) \rightarrow$

$(B \wedge A)$ but it is neither a typing nor well-formedness judgment. CIC follows a propositions-as-types paradigm, so the claim can be reduced to a typing judgment, but at the moment we formulate it, we do not yet have a witness term; we will obtain it only when we prove the claim.

Hence it is more straightforward to introduce another judgment of the form '$\Gamma \vdash T$ true.' We can actually omit the 'true' part, after which the typing judgment will be a special case of $\Gamma \vdash T$ with $t : T$ being a term.

MetaPRL can be made to track witnesses (*extracts* in MetaPRL terminology) to every judgment and, returning to our example of conjunction commutativity, after we prove it we can ask MetaPRL to give us the witness term, i.e., a member of type $(A \wedge B) \rightarrow (B \wedge A)$ (in context Γ).

We omit (*W-E*), (*W-S*), and (*Def*) rules because their conclusions are well-formedness judgments.

(*Ax*) group of rules controls the sorts:

```
rule ax_prop :
   sequent { <H> >- member{Prop;"type"[i:l]}  }

rule ax_set :
   sequent { <H> >- member{Set;"type"[i:l]}  }

rule ax_type :
   sequent { <H> >- member{"type"[i:l];"type"[i':l]}  } .
```

We also declare three auxiliary predicates

```
declare of_some_sort{'P} // 'P has some sort
declare is_sort{'s} // 's is a sort
declare prop_set{'s} // 's is Prop or Set
```

used in later formalizations, with rules

```
rule prop_a_sort :
   sequent { <H> >- member{'P;Prop} } -->
   sequent { <H> >- of_some_sort{'P} }

rule set_a_sort :
   sequent { <H> >- member{'P;Set} } -->
   sequent { <H> >- of_some_sort{'P} }

rule type_a_sort :
   sequent { <H> >- member{'P;"type"[i:l]} } -->
   sequent { <H> >- of_some_sort{'P} }

rule set_is_sort :
   sequent { <H> >- is_sort{Set} }

rule prop_is_sort :
   sequent { <H> >- is_sort{Prop} }

rule type_is_sort :
```

```
   sequent { <H> >- is_sort{"type"[i:l]} }

rule prop_a_prop_set :
   sequent { <H> >- prop_set{Prop} }

rule set_a_prop_set :
   sequent { <H> >- prop_set{Set} } .
```

We introduce a rule that states how to switch from our 'T true' judgment to a typing judgment:

```
rule introduction 't :
   sequent { <H> >- 't in 'T } -->
   sequent { <H> >- 'T } = 't
```

and how to switch back:

```
rule proposition :
   ('t: sequent { <H> >- 'T }) -->
   sequent { <H> >- 't in 'T } .
```

The first rule states that in order to prove that T is true, it is sufficient to prove that $t \in T$ for some t. So we have to provide t as a parameter to the rule, which is why we have t following after the rule name. The '$= t$' part in the conclusion part of the rule says that t is a witness to T. In the case

of this rule, we need to know the witness upfront, but in more complicated cases it can depend on witnesses to assumption sequents, hence MetaPRL will have to track it over the course of the proof.

(*Var*) rule becomes:

```
rule var 'H :
   sequent { <H> >- of_some_sort{'T} } -->
   sequent { <H>; x: 'T; <J['x]> >- 'x in 'T }
```

and as previously stated, we do not support *let* expressions yet and instead of the well-formedness requirement for the entire context and environment, we only impose a check on T.

(*Const*) rule basically states that if we proved earlier that some constant is of certain type, we can use it in our proof; however, this is a standard MetaPRL feature that does not require a separate rule.

Being a lambda calculus CIC defines functional type, lambda abstraction, and application:

```
declare "fun"{'T;x.'U['x]}
declare lambda{'T;x.'t['x]}
declare apply{'t;'u}
```

displayed as $x : T \to U[x]$, $\lambda x : T.t[x]$ and, $t\ u$ respectively.

Rules (*Prod*) for dependent product become:

```
rule prod_1 's1 :
```

```
   sequent { <H> >- prop_set{'s1} } -->
   sequent { <H> >- member{ 'T; 's1 } } -->
   sequent { <H>; x:'T >- member{ 'U['x]; 's2 } } -->
   sequent { <H> >- (x:'T -> 'U['x]) in 's2 }

rule prod_2 's1 :
   sequent { <H>; x:'T >- prop_set{'s2} } -->
   sequent { <H>; x:'T >- 'U['x] in 's2 } -->
   sequent { <H> >- 'T in 's1 } -->
   sequent { <H> >- (x:'T -> 'U['x]) in 's2 }

rule prod_types "type"[i:l] "type"[j:l] :
   sequent { <H> >- 'T in "type"[i:l] } -->
   sequent { <H>; x:'T >- 'U['x] in "type"[j:l] } -->
   sequent { >- level_le[i:l,k:l] } -->
   sequent { >- level_le[j:l,k:l] } -->
   sequent { <H> >- (x:'T -> 'U['x]) in "type"[k:l] } .
```

Traditionally, when U does not depend on an element of T, it is rendered as $T \to U$. Unfortunately, to make this format acceptable by the MetaPRL parser, one has to define a separate term and axiomatize its equivalence. It is somewhat difficult to support this equivalent form uniformly and automatically because at the time of this writing, MetaPRL does not provide such capability out of the box.

(*Lam*) rule becomes:

```
rule lam 's :
   sequent { <H> >- (x:'T -> 'U['x]) in 's  } -->
   sequent { <H> >- is_sort{'s } } -->
   sequent { <H>; x:'T >- 't['x] in 'U['x] } -->
   sequent { <H> >- lambda{'T;x.'t['x]} in (x:'T -> 'U['x]) } .
```

(*App*) becomes:

```
rule app (x:'T -> 'U['x]) :
   sequent { <H> >- 'u in (x:'T -> 'U['x])  } -->
   sequent { <H> >- 't in 'T  } -->
   sequent { <H> >- apply{'u;'t} in 'U['t] }


rule appFormation (x:'T -> 'U['x]) :
   ('u : sequent { <H> >- x:'T -> 'U['x] }) -->
   ('t : sequent { <H> >- 'T })-->
   sequent { <H> >- 'U['t] } = apply{'u;'t} .
```

The second rule is a derivative of the first, and is a form of the cut rule.

For now, we do not implement `let_in`-related rules.

Also note that despite Gentzen-style formulation here of all rules, Coq's interface utilizes natural deduction style, so CIC rules do not possess the usual complementarity of Gentzen style systems, i.e., introduction rules are present but elimination rules are usually not.

1.3.3 Conversion rules

beta-conversion is defined as follows:

```
rewrite beta :
   ( apply{ lambda{'T; x.'t['x]}; 'u } ) <--> ( 't['u] )
```

ι-reduction will be discussed after introduction of induction definitions.

δ-conversion and ζ-conversion are about `let` construct in contexts and `let_in` term, both of which we currently do not support.

Convertibility rule goes into MetaPRL without change; it basically states that if type A is 'less' then type B, all members of A are also members of B:

```
declare conv_le{'t; 's}


rule conv_le_1 :
   sequent { <H> >- conv_le{ 't; 't } }


rule conv_le_2 :
   sequent { >- level_le[i:l,j:l] } -->
   sequent { <H> >- conv_le{ "type"[i:l]; "type"[j:l] } }


rule conv_le_3 :
   sequent { <H> >- conv_le{ Prop; "type"[i:l] } }


rule conv_le_4 :
```

```
   sequent { <H> >- conv_le{ Set; "type"[i:l] } }

rule conv_le_5 :
   sequent { <H>; x:'T >- conv_le{ 'T1['x]; 'U1['x] } } -->
   sequent { <H> >- conv_le{ (x:'T -> 'T1['x]); (x:'T -> 'U1['x]) }}

rule conv_rule 's 'T:
   sequent { <H> >- 'U in 's } -->
   sequent { <H> >- 't in 'T } -->
   sequent { <H> >- conv_le{ 'T; 'U } } -->
   sequent { <H> >- 't in 'U } .
```

1.3.4 Inductive definitions

The most general form of inductive definition in CIC has the form:

$$\texttt{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C)$$

where

Γ is the context in which this definition is well-formed,

Γ_P is a list of typed parameters,

Γ_I is a list of hereby inductively defined types,

Γ_C is a list of constructors for those types.

Example 1

$$\mathit{Ind}()[A : \mathit{Set}](\mathit{List} : \mathit{Set} := \mathit{nil} : \mathit{List}, \mathit{cons} : A \to \mathit{List} \to \mathit{List})$$

defines an inductive type of list, `List` T *then is a list of elements of type* T *(where* T : `Set`*),* `nil` T *is an empty list of type* `List` T*,* `cons` T h t *is a list of type* `List` T *with head* h *(of type* T*) and tail* t *(of type* `List` T*).*

Note that this definition cannot be used to refer to a list of propositions because propositions do not belong to sort `Set`.

Because inductive definitions have variable arity, we have to use sequents. Also, both Γ_I and Γ_C can refer to Γ_P and for that matter, Γ_C can refer to Γ_I; we had to make these sequents nested. Γ is a regular hypothesis list, so we do not really have to include it into the definition.

```
declare sequent [IndParams] { Term : Term >- Term } : Term
declare sequent [IndTypes] { Term : Term >- Term } : Term
declare sequent [IndConstrs] { Term : Term >- Term } : Term
```

and list definition becomes

```
IndParams{| A: Set >-
   IndTypes{| List: Set >-
      IndConstrs{| nil: 'List; cons: 'A -> 'List -> 'List >- it
      |}
   |}
|}
```

where `it` is a 'unit' term which just fills sequent's succedent placeholder.

Types of inductive objects

Every inductive type and constructor from a well-formed inductive definition (the rule will follow) has a certain type and is automatically added to the environment.

Assuming Γ_P is $[p_1 : P_1; \ldots; p_r : P_r]$, Γ_I is $[I_1 : A_1; \ldots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \ldots; c_n : C_n]$,

$$\frac{\texttt{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C) \in E}{(I_j : (p_1 : P_1) \ldots (p_r : P_r) A_j) \in E}$$

for all j and

$$\frac{\texttt{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C) \in E}{(c_i : (p_1 : P_1) \ldots (p_r : P_r) C_i\{I_j/(I_j p_1 \ldots p_r)\}_{j=1\ldots k}) \in E}$$

for all i.

As previously mentioned, MetaPRL does not allow rules to add new constants to the environment, so our rules only assert types, and declaring new constants in a global context can be done separately using `declare` instruction.

As was probably observed, CIC uses 'massive' notation extensively, i.e., we can find this syntax in the rule definitions:

- $(t\ t_1 \ldots t_n)$ which stands for $(\ldots (t\ t_1) \ldots t_n)$,

- $(p_1 : P_1) \ldots (p_n : P_n)A$ is a multiple dependent product,
- $[p_1 : P_1; \ldots; p_n : P_n]A$ is a multiple λ-abstraction.

Consequently, we had to formalize such notations.

```
declare sequent [prodH] { Term : Term >- Term } : Term


prim_rw prodH_base {| reduce |} :
   prodH{| >- 'S |} <--> 'S


prim_rw prodH_step {| reduce |} :
   prodH{| <H>; x:'T >- 'S['x] |} <-->
   prodH{| <H> >- x:'T -> 'S['x] |}
```

basically states that `prodH` with empty antecedent is equivalent to its succedent. If the antecedent is not empty, we can move the last type to succedent and use it as a domain to form a dependent product with old succedent. Therefore, we have a recursive definition which states that `prodH{| x1: T1; ...; xn: Tn >- S['x1,...,'xn] |}` is equivalent to $(x_1 : T_1) \ldots (x_n : T_n)S$. Later in the text we will write $(\Gamma)S$ for $(x_1 : T_1) \ldots (x_n : T_n)S$ if $\Gamma = x_1 : T_1, \ldots, x_n : T_n$.

We also define helper functions that can perform full transformation between `prodH` and a series of `fun` in one (composite) step.

Remember, that MetaPRL does not allow rules to add new constants to the environment, hence we need some means of referring to types and

constructors defined inductively. We do so by placing a variable representing the required type or constructor into the succedent of the innermost sequent of the inductive definition.

Example 2 `define unfold_List : List <-->`

```
IndParams{| A: Set >-
   IndTypes{| List: Set >-
      IndConstrs{| nil: 'List; cons: 'A -> 'List -> 'List >- 'List
      |}
   |}
|}
```

Here, the last occurrence of the `List` variable is where the meaning of the entire expression is defined. As a matter of fact, one can use not only `List`, `nil`, and `cons` here, but any expression.

This is controlled by the following rewrites:

$$\texttt{Ind}[\Gamma_P](\Gamma_I; I : T_{\langle \Gamma_P \rangle}; \Delta_{I,\langle \Gamma_P \rangle} := \Gamma_C[I])t[I] \leftrightarrow$$
$$\texttt{Ind}[\Gamma_P](\Gamma_I; I : T_{\langle \Gamma_P \rangle}; \Delta_{I,\langle \Gamma_P \rangle} := \Gamma_C[I])t[$$
$$\texttt{Ind}[\Gamma_P](\Gamma_I; I : T_{\langle \Gamma_P \rangle}; \Delta_{I,\langle \Gamma_P \rangle} := \Gamma_C[I])I]$$
$$(indSubstDef)$$

$$\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; c : C_{\langle \Gamma_P, \Gamma_I \rangle}; \Delta_{C,\langle \Gamma_P, \Gamma_I \rangle})t[c] \leftrightarrow$$
$$\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; c : C_{\langle \Gamma_P, \Gamma_I \rangle}; \Delta_{C,\langle \Gamma_P, \Gamma_I \rangle})t[$$
$$\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C; c : C_{\langle \Gamma_P, \Gamma_I \rangle}; \Delta_{C,\langle \Gamma_P, \Gamma_I \rangle})c]$$
$$(indSubstConstr)$$

$$\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)t_{\langle\rangle} \leftrightarrow t$$
$$(indCarryOut).$$

We also need to define the massive application:

```
declare sequent [applH] { Term : Term >- Term } : Term


rewrite applHBase :
   applH{| >- 'S |} <-->
   'S


rewrite applHStep :
```

```
   applH{| x:'T; <H> >- 'S |} <-->
   applH{| <H> >- apply{'S;'T}|} .
```

It might appear to reverse the order, but we want `applH{| H >- S |}` to mean $S\ H$ with arguments for S listed in usual order, hence the definition. Later, we will write it as $S\ \Gamma$ for $S\ t_1 \ldots t_n$ if $\Gamma = x_1 : t_1, \ldots, x_n : t_n$. Note that variables' declarations in Γ do not matter – only the types (terms).

Massive lambda abstraction:

```
declare sequent [lambdaH] { Term : Term >- Term } : Term


rewrite lambdaHBase :
   lambdaH{| >-'t|} <--> 't


rewrite lambdaHStep :
   lambdaH{|<H>; x:'s >-'t['x]|} <-->
   lambdaH{|<H> >-lambda{'s; x.'t['x]}|},
```

a shorter notation is $\lambda\Gamma.t$ (traditional) or $[\Gamma]t$ (Coq style).

Finally, we are 'almost' ready to define MetaPRL versions of typing rules for inductive types and constructors. Assuming Γ_P is $[p_1 : P_1; \ldots; p_r : P_r]$, Γ_I is $[I_1 : A_1; \ldots; I_k : A_k]$, and Γ_C is $[c_1 : C_1; \ldots; c_n : C_n]$,

$$\frac{\Gamma \vdash \texttt{IndWF}[\Gamma_P](\Gamma_I := \Gamma_C)}{\Gamma \vdash (\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)I_j) : (\Gamma_P).A_j}$$

for all j and

$$\frac{\Gamma \vdash \texttt{IndWF}[\Gamma_P](\Gamma_I := \Gamma_C)}{\Gamma \vdash ((\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)c_i) : (\Gamma_P)C_i\{I_j/(I_j\ \Gamma_P)\}_{j=1\ldots k})}$$

for all i. We have put 'almost' in quotes because the definition of massive substitution used in the last rule required 16 new types of sequents, 7 primitive rewrites, and several times more consequence-rewrites and helper functions that allow it to perform the substitution by applying those rewrites in the correct order. Its complexity is due to the fact that it had to be performed simultaneously with wrapping into a massive dependent product in order to handle variable binding correctly. Basically, it is a recursive algorithm that goes over every inductive type and parameter and adds parameters to inductive type arguments and wraps everything into a product.

Well-formed inductive definitions

The semi-formal rule from the Coq manual looks like this:

$$\frac{(E[\Gamma;\Gamma_P] \vdash A_j : s'_j)_{j=1\ldots k} \quad (E[\Gamma;\Gamma_P;\Gamma_I] \vdash C_i : s_{p_i})_{i=1\ldots n}}{\mathcal{WF}(E;\texttt{Ind}(\Gamma)[\Gamma_P](\Gamma_I := \Gamma_C))[\Gamma]}$$

with the extra conditions:

(A) $k > 0, I_j, c_i$ are different names for $j = 1 \ldots k$ and $i = 1 \ldots n$,

(B) for $j = 1 \ldots k$ we have A_j is an arity of sort s_j and $I_j \notin \Gamma \cup E$,

(C) for $i = 1 \ldots n$ we have C_i is a type of constructor of I_{p_i} which satisfies

the positivity condition for $I_1 \ldots I_k$ and $c_i \notin \Gamma \cup E$.

These conditions are certainly the most important part of the rule and we had to formalize them.

Our version of the rule itself looks like this:

$$\frac{\begin{array}{ll}\Gamma; \Gamma_P \vdash & of\_some\_sort\_m(\Gamma_I) \\ \Gamma; \Gamma_P \vdash & arity\_of\_some\_sort\_m(\Gamma_I) \\ \Gamma; \Gamma_P \vdash & req3\_m(\Gamma_I := \Gamma_C)\end{array}}{\Gamma \vdash \ \mathtt{IndWF}[\Gamma_P](\Gamma_I := \Gamma_C)}$$

where $of\_some\_sort\_m(\Gamma_I)$ checks that for every $I : A$ in Γ_I $\Gamma, \Gamma_P \vdash of\_some\_sort(A)$ holds.

Similarly, $arity\_of\_some\_sort\_m(\Gamma_I)$ checks that for every $I : A$ in Γ_I $\Gamma, \Gamma_P \vdash arity\_of\_some\_sort(A)$ holds. $arity\_of\_some\_sort(A)$ is defined by the following rules:

$$\frac{}{\Gamma \vdash arity\_of\_some\_sort(\mathtt{Set})} \qquad \frac{}{\Gamma \vdash arity\_of\_some\_sort(\mathtt{Prop})}$$

$$\frac{}{\Gamma \vdash arity\_of\_some\_sort(\mathtt{Type}_i)} \qquad \frac{\Gamma; x : T \vdash arity\_of\_some\_sort(U[x])}{\Gamma \vdash arity\_of\_some\_sort((x : T)U)}$$

$req3\_m(\Gamma_I := \Gamma_C)$ checks that for every $c : C$ from Γ_C

$$\Gamma, \Gamma_P \vdash req3(\Gamma_I; C_{\langle \Gamma, \Gamma_P, \Gamma_I \rangle}) \text{ holds.}$$

Where $req3$ is controlled by this rule:

$$\frac{\begin{array}{lcl}\Gamma, \Gamma_I, I : A_{\langle\Gamma\rangle}, \Delta_{I,\langle\Gamma\rangle} & \vdash & type\_of\_constructor(C[I]; I) \\ \Gamma & \vdash & positivity\_cond\_m(\Gamma_I, I : A_{\langle\Gamma\rangle}, \Delta_{I,\langle\Gamma\rangle \vdash I}) \\ \Gamma & \vdash & arity\_of\_sort(A, s) \\ \Gamma, \Gamma_I, I : A_{\langle\Gamma\rangle}, \Delta_{I,\langle\Gamma\rangle} & \vdash & C[I] : s\end{array}}{\Gamma \vdash req3(\Gamma_I, I : A_{\langle\Gamma\rangle}, \Delta_{I,\langle\Gamma\rangle}; C[I])}.$$

$type\_of\_constructor(C; I)$ means that C is in a form acceptable for a type of constructor for an inductive type I:

$$\frac{}{\Gamma \vdash type\_of\_constructor((I\ \Gamma_{args}); I)}$$

$$\frac{\Gamma, x : T \vdash type\_of\_constructor(C[x]; I)}{\Gamma \vdash type\_of\_constructor((x : T)C[x]; I)}.$$

$\Gamma \vdash positivity\_cond\_m(\Gamma_I, I : A_{\langle\Gamma\rangle}, \Delta_{I,\langle\Gamma\rangle} \vdash C[I])$ checks that $\Gamma, \Gamma_I, I : A_{\langle\Gamma\rangle} \vdash positivity\_cond(C[I]; I)$ holds for every $I : A$ from $positivity\_cond\_m$'s context argument. Singular $positivity\_cond$ is defined as follows:

$$\frac{}{\Gamma, x : T, \Delta[x] \vdash positivity\_cond(x\ \Gamma_{args}; x)}$$

$$\frac{\begin{array}{lcl}\Gamma, x : S, \Delta[x] & \vdash & strictly\_pos(x; T[x]) \\ \Gamma, x : S, \Delta[x], y : T[x] & \vdash & positivity\_cond(U[y, x]; x)\end{array}}{\Gamma, x : S, \Delta[x] \vdash positivity\_cond((y : T[x])U[y, x]; x)},$$

with *strictly_pos* defined as:

$$\frac{}{\Gamma, x : S, \Delta[x] \vdash strictly\_pos(x; T)}$$

$$\frac{}{\Gamma, x : S, \Delta[x] \vdash strictly\_pos(x; x\ \Gamma_{args})}$$

$$\frac{\Gamma, x : S, \Delta[x], y : U \vdash strictly\_pos(x; V[y, x])}{\Gamma, x : S, \Delta[x] \vdash strictly\_pos(x; (y : U)V[y, x])}.$$

One more condition controls the use of inductive definitions in the second argument of *strictly_pos* but it would require introduction of the notion of *imbricated positivity condition* which requires another half-dozen rules, so we decided not to implement it for the time being.

The last undefined notion is $arity\_of\_sort(A, s)$:

$$\frac{}{\Gamma \vdash arity\_of\_sort(\texttt{Set}; \texttt{Set})} \qquad \frac{}{\Gamma \vdash arity\_of\_sort(\texttt{Prop}; \texttt{Prop})}$$

$$\frac{}{\Gamma \vdash arity\_of\_sort(\texttt{Type}_i; \texttt{Type}_i)} \quad \frac{\Gamma, x : T \vdash arity\_of\_sort(U[x]; s)}{\Gamma \vdash arity\_of\_sort((x : T)U[x]; s)}.$$

Destructors

Providing an inductive definition is not enough: one also has to have a means of using it. For inductive types, we need a way to perform case analysis, i.e., which constructor was used to build a given term, as well as a way to define functions on inductive types recursively. CIC uses two separate operations for this. For the purposes of consistency and strong normalizability, CIC allows

primitive recursion only.

Unfortunately, the Coq manual provides only a simplified version of the rules with only one inductive type per definition. So we had to look into [Paulin-Mohring, 1993] for a more general case, which does not, however, use a parameters context in inductive definitions. Therefore we had to adapt it to the modern Coq style.

Our version of the non-dependent case analysis rule:

$$\frac{\begin{array}{lll}
\Gamma;\Gamma_{P,\langle\rangle} & \vdash & \mathit{good_nodep}(A_{\langle\Gamma_P\rangle}; s_{2,\langle\rangle}) \\
\Gamma;\Gamma_{P,\langle\rangle} & \vdash & \mathit{equal_length}(\Gamma_{\mathit{pred},\langle\Gamma\rangle}; \Gamma_I) \\
\Gamma & \vdash & c : (\mathtt{Ind}[\Gamma_P](\Gamma_{I,\langle\Gamma_P\rangle}, I : A_{\langle\Gamma_P\rangle}, \Delta_{I,\langle\Gamma_P\rangle} := \Gamma_{C,\langle\Gamma_P,\Gamma_I,I,\Delta_I\rangle})I\ \Gamma_{\mathit{args}}) \\
\Gamma & \vdash & \mathit{ForAll1T}(\Gamma_{\mathit{pred},\langle\Gamma\rangle}, P_{\langle\Gamma\rangle}, \Delta_{\langle\Gamma\rangle}; p.(p : (\Gamma_P \to s_{2,\langle\rangle}))) \\
\Gamma;\Gamma_{P,\langle\rangle} & \vdash & \mathit{ForAll1T1DT}(\Gamma_{F,\langle\Gamma\rangle}; \\
 & & (\Gamma_{I,\langle\Gamma_P\rangle}, I : A_{\langle\Gamma_P\rangle}, \Delta_{I,\langle\Gamma_P\rangle})\Gamma_{C,\langle\Gamma_P,\Gamma_I,I,\Delta_I\rangle}; \\
 & & f, v, C.(f : \mathit{ElimCaseType}(C; \Gamma_{\mathit{pred},\langle\Gamma\rangle}, P_{\langle\Gamma\rangle}, \Delta_{\mathit{pred},\langle\Gamma\rangle}))
\end{array}}{\Gamma \vdash \mathit{Elim}(c; \Gamma_{\mathit{pred},\langle\Gamma\rangle}, P_{\langle\Gamma\rangle}, \Delta_{\mathit{pred},\langle\Gamma\rangle}; \mathit{ElimCases}(\Gamma_{F,\langle\Gamma\rangle})) : (P_{\langle\Gamma\rangle}\ \Gamma_{\mathit{args},\langle\Gamma\rangle})}$$

and the dependent case analysis rule:

$$\frac{\begin{array}{lll}
\Gamma;\Gamma_{P,\langle\rangle} & \vdash & \mathit{good_dep}(A_{\langle\Gamma_P\rangle};s_{2,\langle\rangle}) \\
\Gamma;\Gamma_{P,\langle\rangle} & \vdash & \mathit{equal_length}(\Gamma_{\mathit{pred},\langle\Gamma\rangle};\Gamma_I) \\
\Gamma & \vdash & c : (\mathtt{Ind}[\Gamma_P](\Gamma_{I,\langle\Gamma_P\rangle}, I : A_{\langle\Gamma_P\rangle}, \Delta_{I,\langle\Gamma_P\rangle} := \Gamma_{C,\langle\Gamma_P,\Gamma_I,I,\Delta_I\rangle})I\ \Gamma_{\mathit{args}}) \\
\Gamma & \vdash & \mathit{ForAll1T}(\Gamma_{\mathit{pred},\langle\Gamma\rangle}, P_{\langle\Gamma\rangle}, \Delta_{\mathit{pred},\langle\Gamma\rangle}; \\
 & & \quad p.(p : \mathit{prodApp}(\Gamma_P; \\
 & & \quad\quad x.(x \to s_{2,\langle\rangle}); \\
 & & \quad\quad \mathtt{Ind}[\Gamma_P](\Gamma_{I,\langle\Gamma_P\rangle}, I : A_{\langle\Gamma_P\rangle}, \Delta_{I,\langle\Gamma_P\rangle} := \Gamma_{C,\langle\Gamma_P,\Gamma_I,I,\Delta_I\rangle})I \\
 & & \quad)) \\
 & &) \\
\Gamma & \vdash & \mathit{ForAll1TConstr}(\Gamma_F; \\
 & & \quad \mathtt{Ind}[\Gamma_P](\Gamma_{I,\langle\Gamma_P\rangle}, I : A_{\langle\Gamma_P\rangle}, \Delta_{I,\langle\Gamma_P\rangle} := \Gamma_{C,\langle\Gamma_P,\Gamma_I,I,\Delta_I\rangle}); \\
 & & \quad f, v, C.(f : \mathit{ElimCaseTypeDep}(C;\Gamma_{\mathit{pred},\langle\Gamma\rangle}, P_{\langle\Gamma\rangle}, \Delta_{\mathit{pred},\langle\Gamma\rangle}; c) \\
 & &)
\end{array}}{\Gamma \vdash \mathit{Elim}(c;\Gamma_{\mathit{pred},\langle\Gamma\rangle}, P_{\langle\Gamma\rangle}, \Delta_{\mathit{pred},\langle\Gamma\rangle}; \mathit{ElimCases}(\Gamma_{F,\langle\Gamma\rangle})) : (P_{\langle\Gamma\rangle}\ \Gamma_{\mathit{args},\langle\Gamma\rangle}\ c)}$$

with the following reductions:

$$
\begin{aligned}
&\mathit{Elim}(\texttt{Ind}[\Gamma_P](\Gamma_I, := \Gamma_C, c : C_{\langle \Gamma_P, \Gamma_I \rangle}, \Delta_C) c\ \Gamma_{\mathit{args}}; \Gamma_{\mathit{pred}}; \mathit{ElimCases}(\Gamma_F)) \quad \leftrightarrow \\
&\mathit{ElimBracket}(\\
&\quad \texttt{Ind}[\Gamma_P](\Gamma_I, := \Gamma_C, c : C_{\langle \Gamma_P, \Gamma_I \rangle}, \Delta) C; \\
&\quad \mathit{FunElim}(\\
&\qquad \mathit{mainType}(\texttt{Ind}[\Gamma_P](\Gamma_I, := \Gamma_C, c : C_{\langle \Gamma_P, \Gamma_I \rangle}, \Delta_C) C); \\
&\qquad \Gamma_{\mathit{pred}}; \\
&\qquad \mathit{ElimCases}(\Gamma_F) \\
&\quad); \\
&\quad \mathit{BackHyp}(\Gamma_P, \Gamma_I, \Gamma_C \vdash \mathit{Back}(\Delta_C \vdash \mathit{BackIn}(\Gamma_F))) \\
&) \Gamma_{\mathit{args}}
\end{aligned}
$$

where the purpose of BackHyp, Back and BackIn is to pick the element of F relevant to $c : C$. F has one element per each constructor, so this construct peels as many elements from F tail as Δ_C has, and then the last element of the rest of F is the term relevant to $c : C$. We had to incorporate Γ_P, Γ_I, and Γ_C into the definition because Δ_C is not closed relative to them and cannot be used independently.

mainType picks out an inductive type from the "main" positive position

of its argument:

$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma)(x : A)B[x])) \leftrightarrow$$
$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma, x : A)B[x]))$$

$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma)(A \to B))) \leftrightarrow$$
$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma)B))$$

$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma)(\Delta)B[x])) \leftrightarrow$$
$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma, \Delta)B[x]))$$

$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma)(Aa))) \leftrightarrow$$
$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma)(A)))$$

$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma)(A\Delta))) \leftrightarrow$$
$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((\Gamma)(A)))$$

$$mainType(\texttt{Ind}[\Gamma_P](\Gamma_I, I : A, \Delta_I := \Gamma_{C,\langle I\rangle})((\Gamma_{\langle I\rangle})I)) \leftrightarrow$$
$$\texttt{Ind}[\Gamma_P](\Gamma_I, I : A, \Delta_I := \Gamma_{C,\langle I\rangle})I$$

After mainType, FunElim does its job:

$$FunElim(\texttt{Ind}[\Gamma_P](\Gamma_I, I : A, \Delta_I := \Gamma_{C,\langle I\rangle})I) \leftrightarrow$$

$$\lambda\Gamma_P, c : (\texttt{Ind}[\Gamma_P](\Gamma_I, I : A, \Delta_I := \Gamma_{C,\langle I\rangle})I\ \Gamma_P).Elim(c; \Gamma_{pred}; ElimCases(\Gamma_F))$$

and for ElimBracket we need the following reductions:

$$ElimBracket(\texttt{Ind}[\Gamma_P](\Gamma_I, I : A, \Delta_I := \Gamma_{C,\langle I\rangle})(((\Gamma)(I\Gamma_{args,\langle\Gamma\rangle})) \to C[I]); F; f) \quad \leftrightarrow$$

$$(p : (\Gamma)(\texttt{Ind}[\Gamma_P](\Gamma_I, I : A, \Delta_I := \Gamma_{C,\langle I\rangle})I\ \Gamma_{args,\langle\Gamma\rangle}))$$

$$)\ ElimBracket(\texttt{Ind}[\Gamma_P](\Gamma_I, I : A, \Delta_I := \Gamma_{C,\langle I\rangle})C[I];$$

$$F;$$

$$f\ p\ (\lambda\Gamma.(F\ \Gamma_{args,\langle\Gamma\rangle}\ (p\ \Gamma)))$$

$$)$$

$$ElimBracket(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)((x : M_{\langle\rangle})C[x]); F; f) \ \leftrightarrow$$

$$(x : M_{\langle\rangle})ElimBracket(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)C[x]; F; f\ x)$$

$$ElimBracket(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)(M_{\langle\rangle})C); F; f) \ \leftrightarrow$$

$$(x : M_{\langle\rangle})ElimBracket(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)C; F; f\ x)$$

$$ElimBracket(\texttt{Ind}[\Gamma_P](\Gamma_I := \Gamma_C)(I\ \Gamma_{args,\langle\rangle}); F; f) \ \leftrightarrow$$

$$f.$$

The rules governing fixpoint turned out to be even more complex, see [Giménez, 1995]. On the other hand, for any reasonable inductive type, one can come up with a reasonable fixpoint definition relatively easily, which is the direction Martin-Löf's ITT [Martin-Löf, 1984] took — he defines a new type and defines a fixpoint operation for it (if any). The same approach was used by LEGO [Luo, 1990], which is based on Calculus of Constructions as Coq. Even Coq lacks a general form of fixpoint well-formedness rule for mutually inductive types; instead, it offers a tool to generate rule schemas for particular inductive definitions. We suggest the same approach for our implementation of CIC in MetaPRL— let users define the fixpoint operations for their types.

1.4 Summary

We implemented most of the rules of CIC except for the fixpoint rules. The complexity of these rules was increasingly high and we were overwhelmed with the complexity of the fixpoint part.

Nevertheless, a MetaPRL approach allows us to add new rules on the spot, so potential users can add specific instances of these rules as they see fit.

One possible application of this work is to perform re-validation of the

existing library of Coq proofs. The more recent versions of Coq have a tool for generating XML representations of proofs but those proofs lack typing information, therefore, in order to run them against our implementation, we need to implement automatic type deduction for CIC terms. This, of course, is a standard feature of Coq, and a topic for future work. Another option is to make Coq able to export proofs with full typing information.

Chapter 2

Translation of $\mathsf{S4}^{\mathsf{J}}_n$ cut-free proofs into $\mathsf{S4}_n\mathsf{LP}$ proof

2.1 Introduction

$\mathsf{S4}^{\mathsf{J}}_n$ [Artemov and Nogina, 2005; Artemov, 2006; Artemov, 2007; Fitting, 2007c; Fitting, 2007d] is an epistemic logic of justified knowledge with n agents. The general study of knowledge and belief is called *epistemology*, and the subfield of modal logic that studies knowledge and beliefs is called *epistemic logic*. We know that the philosophers of ancient Greece already pondered such questions as: 'What do people know?,' 'What is knowledge?,' and 'What does it mean to say that someone knows something?' One of the earliest works on epistemic logic dates back to [von Wright, 1951].

One interesting aspect of epistemology is the study of knowledge in a

group setting. People (or agents) in a group interact with each other and it is interesting to observe the knowledge aspects of this interaction. Different people in a group can know different things so we can try to model certain aspects of this situation, e.g.

- How can one person's knowledge affect the knowledge of others?
- Can some knowledge be shared?

One of the basic attempts to formalize shared knowledge is the notion of *common knowledge*.

The basic modal operator of epistemic logic is usually written as K, and read as 'it is known that.' If we have a group of n people (*agents*), then n modal operators can be used: $K_1, K_2, \ldots, K_n$ and $K_i\phi$ stands for 'agent i knows ϕ.' Two different agents can have different knowledge, i.e., if one of them knows something, it does not necessary follow that other agents know it, too. The *common knowledge* modality C can be defined informally as

$E\phi = K_1\phi \wedge K_2\phi \wedge \ldots \wedge K_n\phi$

$C\phi \Leftrightarrow \phi \wedge E\phi \wedge E^2\phi \wedge \ldots \wedge E^n\phi \wedge \ldots$.

n-agent epistemic logic with common knowledge is instrumental for reasoning about puzzles such as Muddy Children, Wise Men, etc. [Fagin *et al.*, 1995].

The traditional approach to formalize common knowledge is to use a Fixed-Point Axiom

$$C\phi \leftrightarrow E(\phi \wedge C\phi)$$

along with an Induction Rule

$$\frac{\phi \to E(\psi \wedge \phi)}{\phi \to C\psi}$$

that captures the greatest solution of the corresponding fixed-point equation [Fagin *et al.*, 1995].

This kind of system is not easy to work with — it has no traditional cut elimination and, consequently, there is little hope for automatic proof search.

Dating back to the 70s, McCarthy undertakes alternative attempts to describe common knowledge axiomatically [McCarthy *et al.*, 1979], but this work wasn't continued.

An approach similar to McCarthy's was taken in $\mathsf{S4}^{\mathsf{J}}_n$ and is actively being developed at this time. $\mathsf{S4}^{\mathsf{J}}_n$ is strictly weaker than system $\mathsf{S4}^{\mathsf{C}}_n$ [Antonakos, 2006], but seems sufficient for all practical applications. $\mathsf{S4}^{\mathsf{J}}_n$ axiomatics allow standard methods of proof theory, and allows the drawing of parallels with Artemov's logic of explicit proofs LP.

There is an automatic prover for multi-agent logic with justified knowledge $\mathsf{S4}^{\mathsf{J}}_n$ in the MetaPRL logical framework [Bryukhov, 2005; Bryukhov, 2006]; its output is a Gentzen-style cut-free proof. Such proofs can be translated into $\mathsf{S4}_n\mathsf{LP}$ proofs with justified knowledge modality replaced by evidence terms. It would be interesting to make such a translation automatically, and this is what we do. Since known translation to Gentzen-style proofs might result in exponential blow-up of the proof length, we'll be look-

ing for a translation to Hilbert-style proofs, which is known to result in only a polynomial blow-up in an S4-to-LP case [Brezhnev and Kuznets, 2006].

2.2 Overview of $\mathsf{S4}_n\mathsf{LP}$ logic

$\mathsf{S4}_n\mathsf{LP}$ is a multi-agent logic of evidence-based knowledge, with knowledge operators of n agents $K_1, K_2, K_3, \ldots, K_n$, and evidence assertions of the form $t : A$, where t is an evidence term and A is a formula. Evidence term t is built from constants $a, b, c, \ldots$ and variables $x, y, z, \ldots$ with the help of binary operators '$\cdot$' (application), '$+$' (union), and unary operator '!' (inspection). Formulas of $\mathsf{S4}_n\mathsf{LP}$ are defined by the following grammar:
$\bot \mid S \mid A \to B \mid A \wedge B \mid A \vee B \mid \neg A \mid K_i A \mid t : A$, where t is an evidence and S is a sentence variable.

Evidence operation has highest precedence, and all other connectives have standard precedence order.

Hilbert style axioms and rules of $\mathsf{S4}_n\mathsf{LP}$:

I. Classical propositional logic

A1. $A \to (B \to A)$

A2. $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$

A3. $A \wedge B \to A$

A4. $A \wedge B \to B$

A5. $A \to (B \to (A \wedge B))$

A6. $A \to (A \vee B)$

A7. $B \to (A \vee B)$

A8. $(A \to C) \to ((B \to C) \to (A \vee B \to C))$

A9. $(A \to B) \to ((A \to \neg B) \to \neg A)$

A10. $\neg\neg A \to A$

R1. $(A \to B), A \vdash B$ (*modus ponens*)

II. Knowledge principles

B1$_i$. $K_i(A \to B) \to (K_i A \to K_i B)$

B2$_i$. $K_i A \to A$

B3$_i$. $K_i A \to K_i K_i A$ (*positive introspection*)

R2$_i$. $A \vdash K_i A$ (*knowledge generalization*)

for each individual knowledge operator K_i.

III. Evidence Principles

E1. $s : (A \to B) \to (t : A \to (s \cdot t) : B)$ (*application*)

E2. $t : A \to !t : (t : A)$ (*inspection*)

E3. $s : A \to (s + t) : A, \quad t : A \to (s + t) : A$ (*union*)

E4. $t : A \to A$ (*reflexivity*)

R3. $\vdash c : A$, where A is an axiom from I-IV and c is a proof constant (*evidence for axioms*).

IV. Principle connecting evidence and knowledge

C1. $t : A \to K_i A$ (*undeniability of evidence*)

All axioms are schemas in the language of $\mathsf{S4}_n\mathsf{LP}$. All rules are applied across sections I - IV. The system is closed under substitutions of evidence terms for evidence variables and formulas for propositional variables. Deduction theorem $\Gamma, A \vdash B \Rightarrow \Gamma \vdash A \to B$ holds.

Lemma 1 (Lifting Lemma) *[Artemov, 2001; Artemov, 2004]* If $A_1, \ldots, A_n, y_1 : B_1, \ldots, y_m : B_m \vdash F$, then for some evidence term $t = t(x_1, \ldots, x_n, y_1, \ldots, y_m)$,

$$x_1 : A_1, \ldots, x_n : A_n, y_1 : B_1, \ldots, y_m : B_m \vdash t(x_1, \ldots, x_m, y_1, \ldots, y_n) : F$$

Lemma 2 *For any proof variables $\vec{x_i}$ and any formulas $\vec{B_i}$, there exists a proof term $s = s(x_1, \ldots, x_n)$ such that*

$$\mathsf{LP} \vdash x_1 : B_1 \wedge \ldots \wedge x_n : B_n \to s(x_1, \ldots, x_n) : (x_1 : B_1 \wedge \ldots x_n : B_n).$$

2.3 Overview of $\mathsf{S4}^{\mathsf{J}}_n$ system

$\mathsf{S4}^{\mathsf{J}}_n$ is a forgetful evidence-based logic with $n+1$ modalities $K_1, \ldots, K_n, J$. JA reads as 'A is justified' and is a forgetful projection of evidence assertion $t : A$.

The forgetful version of *undeniability of evidence* principle for $\mathsf{S4}^{\mathsf{J}}_n$ is $JA \to K_i A$, for all $i = 1, \ldots, n$.

Below we present a Gentzen-style formulation [Troelstra and Schwichtenberg, 2000] of $\mathsf{S4}^{\mathsf{J}}_n$ called $\mathsf{S4}^{\mathsf{J}}_n\mathsf{G}$ [Artemov, 2004; Brezhnev and Kuznets, 2006].

A *sequent* is a pair of finite sets of $\mathsf{S4}^{\mathsf{J}}_n$ formulas presented as $\Gamma \Rightarrow \Delta$. Axioms of Gentzen-style $\mathsf{S4}^{\mathsf{J}}_n$ are the sequents $S, \Gamma \Rightarrow \Delta, S$ and $\bot, \Gamma \Rightarrow \Delta$, where S is a propositional variable.

Gentzen style rules of $\mathsf{S4}^{\mathsf{J}}_n$:

$$\frac{\Gamma \Rightarrow \Delta, A}{\neg A, \Gamma \Rightarrow \Delta}(\neg \Rightarrow) \qquad \frac{A, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \neg A}(\Rightarrow \neg)$$

$$\frac{A, B, \Gamma \Rightarrow \Delta}{A \wedge B, \Gamma \Rightarrow \Delta}(\wedge \Rightarrow) \qquad \frac{\Gamma \Rightarrow \Delta, A \quad \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \wedge B}(\Rightarrow \wedge)$$

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

$$\frac{\Gamma \Rightarrow \Delta, A \quad B, \Gamma \Rightarrow A}{A \to B, \Gamma \Rightarrow \Delta}(\to \Rightarrow) \qquad \frac{A, \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \to B}(\Rightarrow \to)$$

and $n+1$ pairs of modal rules:

$$\frac{A, \Box A, \Gamma \Rightarrow \Delta}{\Box A, \Gamma \Rightarrow \Delta}(\Box \Rightarrow) \qquad \frac{J\Gamma, \Box\Delta \Rightarrow A}{J\Gamma, \Box\Delta, \Pi \Rightarrow \Sigma, \Box A}(\Rightarrow \Box)$$

where $\Box \in \{K_1, \ldots, K_n, J\}$ and $\Box\{A_1, \ldots, A_m\} = \{\Box A_1, \ldots, \Box A_m\}$.

2.4 Main Definitions and Facts

Our goal is to replace justifiable knowledge, modality J, with explicit justifications, i.e., proof polynomials (proof terms). In other words, instead of knowing that something is 'justified,' we would like to have its actual justification. An algorithm that accomplishes this is called *realization procedure*. An exponential version of such an algorithm was first described for LP in [Artemov, 1995], and a polynomial version was given in [Brezhnev and Kuznets, 2006]. Its extension to $\mathsf{S4}_n\mathsf{LP}$ is relatively straightforward. Since

this procedure is quite tedious, it would be nice to have a program that performs this realization for us.

Before we describe the algorithm, allow us to present some definitions and state the theorems and facts that we will need.

Definition 1 *To* realize *a modal formula F from $\mathsf{S4}^{\mathsf{J}}_n$ in $\mathsf{S4}_n\mathsf{LP}$ means to substitute proof polynomials for all occurrences of J in F.*

Definition 2 *A realization r is called* normal *if all negative occurrences of J are realized by proof variables.*

We limit ourselves to normal realizations only. Such a limitation arises from the fact that negative occurrences of J modalities constitute arguments of Skolem functions. Arguments emerge from universal quantifiers and Skolem functions from existential quantifiers. [Artemov, 1995; Artemov, 2001; Brezhnev and Kuznets, 2006].

Example 3 *Consider a formula $\mathsf{J}A \to \mathsf{J}B$. This can be read as "for any proof of A there exists a proof of B" or $\forall x.Proof(x, A) \to \exists y.Proof(y, B)$, where x is a proof variable (supplied) and y is a proof polynomial which may depend on x. The skolemized form of this formula is $x : A \to f(x) : B$, and x is an argument of Skolem function f.*

Thus it is reasonable to demand that these Skolem arguments are realized by proof variables rather than by more complicated proof terms (proof

polynomials).

Realization theorem. [Artemov, 2004] There is an algorithm that given a cut-free $\mathsf{S4}^{\mathsf{J}}_n$-derivation of a formula φ, retrieves a $\mathsf{S4}_n\mathsf{LP}$-derivation of a formula ψ such that $(\psi)^\circ = \varphi$.

Translation $^\circ$ maps $t : \varphi$ to $J\varphi$ and commutes with all other connectives.

The algorithm presented is of exponential complexity.

There are two earlier realization algorithms for $\mathsf{S4}$ and LP. In [Artemov, 1995] the realization algorithm produces a Hilbert-style derivation, and in [Artemov, 2001] the realization procedure produces a Gentzen-style derivation. Kuznets showed that both variants give an exponential blow-up of the derivation size and Brezhnev suggested a modification [Brezhnev and Kuznets, 2006] to the procedure from [Artemov, 1995] that produces at most a polynomial overhead. In [Brezhnev and Kuznets, 2006], the realization theorem states that if $\mathsf{S4} \vdash F$, then $\mathsf{LP} \vdash F^r$ for some normal realization and a new realization algorithm that produces proof polynomials of at most quadratic length is given. The realization algorithm described below utilizes all of these results.

2.5 Realization Algorithm

The realization procedure works by induction on the depth of the $\mathsf{S4}^{\mathsf{J}}_n$ derivation tree. It runs through the Getzen-style proof of a formula F in $\mathsf{S4}^{\mathsf{J}}_n$ and

simultaneously constructs a realization and Hilbert-style proof of the realized formula. We also keep track of all instances of the evidence for axiom rule R3 used in this Hilbert-style proof, i.e., of constant specification.

We start with definitions of positive and negative occurrences of modality J in a formula and in a sequent, as adapted from [Artemov, 1995]:

- An indicated (outer) occurrence of J in $\mathsf{J}F$ is positive;
- A corresponding occurrence of J in F and $G \to F$, $G \vee F$, $G \wedge F$, $\mathsf{J}F$, and $\Gamma \Rightarrow \Delta, F$ has the same polarity;
- Corresponding occurrences of J in F and $\neg F$, $F \to G$, and $F, \Gamma \Rightarrow \Delta$ have opposite polarities.

In a cut-free derivation, the rules respect polarities. Occurrences of J introduced by $(\Rightarrow \mathsf{J})$ are positive:

$$\frac{J\Gamma \Rightarrow A}{J\Gamma, \Pi \Rightarrow \Sigma, \mathbf{J}\mathbf{A}}(\Rightarrow \mathsf{J}).$$

All occurrences of J-modality in a given derivation tree of $\Rightarrow F$ are divided into families of related occurrences. Each occurrence of J in a side formula G (i.e., from Γ and Δ) in the premise of the rule is related only to the corresponding occurrence of J in G in the conclusion of the rule. Similarly, each occurrence of J in an active formula of the rule, i.e., in a formula in the premise that is transformed by the rule, is related only to the corresponding occurrence of J in the principal formula of the rule, i.e. in the result of

transformation. For example, in the ($\mathsf{J} \Rightarrow$)-rule, formulas A and $\mathsf{J}A$ in the premise sequent are the active formulas, and formula $\mathsf{J}A$ in the conclusion sequent is the principal formula. This relationship is extended by reflexivity and transitivity. Therefore all related occurrences are naturally split into *families of related occurrences.*

Since rules in the cut-free Gentzen system respect polarities, each family consists of J's of the same polarity. We call a family *positive* if it consists of positive J's, and *negative* if it consists of negative J's.

J modalities from the same family correspond to the same occurrence of J in the proof, so we realize them by the same proof polynomial that explicates this J. In addition, due to the normality condition, all J's from a negative family have to be realized by the same proof variable.

Proofs (derivations) of formulas in a Getzen system, $\mathsf{S4}^{\mathsf{J}}_n\mathsf{G}$ in particular, can be viewed as derivation trees. Nodes are triples: the current sequent, name of the rule and the principal formulas, and axioms as leaves. It imposes a tree structure on each family of J's, with leaves as those nodes where J's of a particular family are first introduced (at a leaf of the derivation tree or in a ($\Rightarrow \Box$) rule).

Comment: It is not necessary to carry sequents in every node - they can be reconstructed from rule names, principle formulas, and full sequents at the root. The implemented realization algorithm uses the full information, whereas the gen_prover (see below) produces a proof in the latter, more compact, format.

A positive family of J's is *essential* if at least one of its leaves corresponds to a principal J in a $(\Rightarrow \Box)$ rule, and is *non-essential* otherwise.

Realization algorithm:

(by recursion on the derivation tree structure)

In our system $\mathsf{S4}^{\mathsf{J}}_n\mathsf{G}$, there are only three ways of introducing new J-modalities:

- by an axiom;
- inside a formula by which a sequent is 'weakened' in a $(\Rightarrow \Box)$ rule;
- the outer J in the principal formula of a $(\Rightarrow \Box)$ rule.

Let us enumerate all $(\Rightarrow \mathsf{J})$ rules in the derivation tree and associate provisional variable u_i with the principal J of the i-th rule. All of the provisional variables will be replaced with proof polynomials by the end of the algorithm.

Stage 1 *Every negative family and non-essential family of* J*'s is realized by a fresh proof variable. All* J*'s from such a family will be realized by a proof variable corresponding to that family.*

Stage 2 *Pick an essential positive family of* J*'s. Enumerate all the occurrences of* $(\Rightarrow \Box)$ *rules that introduce* J*'s from this family as the principles:* $i_1 < i_2 < \ldots < i_k$*. All such* J*'s are initially realized by provisional term*

$u_{i_1} + u_{i_2} + \ldots u_{i_k}$, *where addition is associated to the left and* u_i*'s are fresh provisional variables.*

We also initialize a substitution σ*, which acts on these provisional variables, to be the empty substitution. At the end of the realization procedure, this substitution will assign a certain proof polynomial to each provisional variable. As a result, essential positive* J*'s will also be realized by proof polynomials that contain no provisional variables.*

Next, each $\mathsf{S4}_n^{\mathsf{J}}$ *formula* G *occurring in the sequent derivation is translated into an* $\mathsf{S4}_n\mathsf{LP}$ *formula* G_r *as follows: each occurrence of* J *in* G *is replaced by a proof polynomial* $t\sigma$ *that possibly contains provisional variables, where* t *is the term realizing the family of that* J*, and* σ *is the current state of the substitution acting on provisional variables. This substitution is appended during the realization procedure, namely, during processing of* $(\Rightarrow \Box)$ *rules.*

Stage 3 *For each sequent in the initial derivation we will construct*

- *an* $\mathsf{S4}_n\mathsf{LP}$ *formula* C *that corresponds to that sequent,*
- *a proof polynomial* t *that contains no provisional variables, and*
- *a Hilbert-style derivation of* $t : C$

recursively on the structure of the derivation tree of $\Rightarrow F$*.*

Kuznets and Brezhnev had the idea to use the polynomials t. They are used while processing the $(\Rightarrow \Box)$ rules of the initial $\mathsf{S4}_n^{\mathsf{J}}$ derivation, and are a vital part of eliminating exponential blow-up.

Base: Let C be a sequent formula. Any sequent formula $\Gamma \Rightarrow \Delta$, where $\Gamma = \{A_1, \ldots, A_n\}$, and $\Delta = \{B_1, \ldots, B_m\}$,

$$A_1, A_2, \ldots, A_n \Rightarrow B_1, B_2, \ldots, B_m$$

is translated into a formula

$$(\ldots(A_1^r \wedge A_2^r) \wedge \ldots) \wedge A_n^r \rightarrow (\ldots(B_1^r \vee B_2^r) \vee \ldots) \vee B_m^r \ .$$

The antecedent and the consequent of a sequent are multisets, so the order of formulas is irrelevant in both, but normal Hilbert-style operations do not possess such freedom. Thus we need to force some order on A_i^r's and on B_i^r's, so we can use any ordering that allows efficient sorting. This ordering should be uniform for all sequents. This is important for Cook and Reckhow's idea [Cook and Reckhow, 1974] of implementing each step of Gentzen-style derivation by several steps of the corresponding Hilbert-style derivation, overwise the formulas on different branches of the tree might not match. The lexicographical order is a natural one.

An empty consequent constitutes empty disjunction and is translated as $\bot$. An empty antecedent constitutes empty conjunction and is translated as $\top$. Therefore $\Rightarrow F$ is translated as $\top \rightarrow F^r$.

Translation of two axioms of $\mathsf{S4}_n^{\mathsf{J}}\mathsf{G}$:

(A) $A_1, \ldots, A_{i-1}, S, A_i, \ldots, A_n \Rightarrow B_1, \ldots, B_{j-1}, S, B_j, \ldots, B_m$

is translated as

$A_1^r \wedge \ldots \wedge A_{i-1}^r \wedge S \wedge A_i^r \wedge \ldots \wedge A_n^r \rightarrow B_1^r \vee \ldots \vee B_{j-1}^r \vee S \vee B_j^r \vee \ldots \vee B_m^r$,

in particular, $S \Rightarrow S$ is translated as $S^r \rightarrow S^r$;

(B) $\bot, A_1, \ldots, A_n \Rightarrow B_1, \ldots, B_m$

is translated as

$\bot \wedge A_1^r \wedge \ldots \wedge A_n^r \rightarrow B_1^r \vee \ldots \vee B_m^r$;

in particular, $\bot \Rightarrow$ is translated as $\bot \rightarrow \bot$.

(Assuming that A_k's and B_l's, A_k^r's and B_l^r's are already ordered alphabetically, and $\bot$ is the first symbol of the alphabet, disjunctions and conjunctions are associated to the left.)

Each translated implication C of this type is clearly derivable in $\mathsf{S4}_n\mathsf{LP}$. After application of the Lifting Lemma to this derivation, we get a ground proof polynomial s and a derivation of $s : C$.

Induction Step:

Propositional rule with one premise:

$$\frac{\Gamma \Rightarrow \Delta}{\Gamma' \Rightarrow \Delta'} .$$

Let C and C' be translations of $\Gamma \Rightarrow \Delta$ and $\Gamma' \Rightarrow \Delta'$ respectively. By induction hypothesis, there is a term t_C and a derivation l_C of $t_C : C$. By propositional reasoning, there is a derivation of $C \rightarrow C'$. Using the Lifting

Lemma, we get a ground term t_R and a derivation l_R of $t_R : (C \to C')$. Concatenating l_C with l_R and appending the result with the following sequence

$\ldots$ (*derivation* l_R)

`n.` $t_R : (C \to C')$

$\ldots$ (*derivation* l_C)

`m.` $t_C : C$

`m+1.` $t_R : (C \to C') \to (t_C : C \to t_R \cdot t_C : C')$ (axiom $E1$)

`m+2.` $t_C : C \to t_R \cdot t_C : C'$ (MP from n and m+1)

`m+3.` $t_R \cdot t_C : C'$ (MP from m and m+2) ,

we obtain the term $t_{C'} = t_R \cdot t_C$ and the derivation $l_{C'}$ of $t_R \cdot t_C : C'$.

A case of a propositional rule with two premises are handled in a similar way.

Let us consider $(\Box \Rightarrow)$ rule for $\Gamma = \{B_1, \ldots, B_n\}$, and $\Delta = \{D_1, \ldots, D_m\}$:

$$\frac{A, \mathsf{J}A, B_1, \ldots, B_n \Rightarrow D_1, \ldots, D_m}{\mathsf{J}A, B_1, \ldots, B_n \Rightarrow D_1, \ldots, D_m}(\Box \Rightarrow) \ .$$

Without loss of generality, let's assume that the translation of the premise is

$$C = B_1^r \wedge \ldots \wedge B_{i-1}^r \wedge A^r \wedge B_i^r \wedge \ldots \wedge B_{j-1}^r \wedge x : A^r \wedge B_j^r \wedge \ldots \wedge B_n^r \to D,$$

where $D = D_1^r \vee \ldots \vee D_m^r$ and x is the proof variable associated with the

negative family of the outer J-modality in JA. Then the translation of the conclusion is

$$C' = B_1^r \wedge \ldots \wedge B_{j-1}^r \wedge x : A^r \wedge B_j^r \wedge \ldots \wedge B_n^r \rightarrow D.$$

Since $\mathsf{S4}_n\mathsf{LP} \vdash x : A^r \rightarrow A^r$ (reflexivity principle E4), it is easy to derive $C \rightarrow C'$. Then, using the Lifting Lemma, we obtain a ground term $t_{(\Box\Rightarrow)}$ and a derivation $l_{(\Box\Rightarrow)}$ of $t_{(\Box\Rightarrow)} : (C \rightarrow C')$. The rest is the same as with the one-premise propositional rules.

The only rule that is treated differently is $(\Rightarrow \Box)$, which includes two cases: $(\Rightarrow K_i)$ and $(\Rightarrow \mathsf{J})$. Let's consider the first one: for $\Gamma = \{B_1, \ldots, B_n\}$, $\Delta = \{D_1, \ldots, D_m\}$, $\Sigma = \{E_1, \ldots, E_o\}$, and $\Pi = \{A_1, \ldots, A_r\}$

$$\frac{J\Gamma, K_i\Delta \Rightarrow A}{J\Gamma, K_i\Delta, \Pi \Rightarrow \Sigma, K_iA}(\Rightarrow K_i) \ .$$

Without loss of generality, let's assume that the translation of the premise is

$$C = \vec{x} : \Gamma^r \wedge K_i\Delta^r \rightarrow A^r$$

and $\vec{x} = (x_1, \ldots, x_n)$, where x_i are distinct proof variables associated with the negative families of the outer J-modality in JΓ.

Then the translation of the conclusion is

$$C' = \vec{x} : \Gamma^r \wedge K_i\Delta^r \wedge \Pi^r \rightarrow \Sigma^r \vee K_iA^r \ .$$

By induction hypothesis, we have a term t_C and a derivation l_C of

$$t_C : (\vec{x} : \Gamma^r \wedge K_i : \vec{D^r} \to A^r)$$

$\ldots$ (*derivation* l_C)

n. $t_C : (\vec{x} : \Gamma^r \wedge K_i\Delta^r \to A^r)$

n+1. $t_C : (\vec{x} : \Gamma^r \wedge K_i\Delta^r \to A^r) \to K_i(\vec{x} : \Gamma^r \wedge K_i\Delta^r \to A^r)$ (axiom $C1$)

n+2. $K_i(\vec{x} : \Gamma^r \wedge K_i\Delta^r \to A^r)$ (MP from n and n+1)

n+3. $K_i(\vec{x} : \Gamma^r \wedge K_i\Delta^r \to A^r) \to (K_i(\vec{x} : \Gamma^r \wedge K_i\Delta^r) \to K_iA^r)$ (axiom $B1_i$)

n+4. $K_i(\vec{x} : \Gamma^r \wedge K_i\Delta^r) \to K_iA^r$ (MP from n+2 and n+3)

n+5. $\left(\bigwedge K_ix_j : B_j^r\right) \wedge \left(\bigwedge K_iK_iD_l^r\right) \to K_i(\vec{x} : \Gamma^r \wedge K_i\Delta^r)$ (simple $\mathsf{S4}_n\mathsf{LP}$ reasoning)

n+6. $\left(\bigwedge K_ix_j : B_j^r\right) \wedge \left(\bigwedge K_iK_iD_l^r\right) \to K_iA^r$ (*syllogism* from n+4 and n+5)

n+7. $x_j : B_j^r \to K_ix_j : B_j^r$ (an easy $\mathsf{S4}_n\mathsf{LP}$ fact)

n+8. $K_iD_l^r \to K_iK_iD_l^r$ (axiom $B3_i$)

n+9. $\left(\bigwedge x_j : B_j^r\right) \wedge \left(\bigwedge K_iD_l^r\right) \to \left(\bigwedge K_ix_j : B_j^r\right) \wedge \left(\bigwedge K_iK_iD_l^r\right)$

(propositional reasoning from n+7 and n+8)

n+10. $\left(\bigwedge x_j : B_j^r\right) \wedge \left(\bigwedge K_iD_l^r\right) \to K_iA^r$ (*syllogism* from n+6 and n+9)

n+11. $\left(\bigwedge x_j : B_j^r\right) \wedge \left(\bigwedge K_iD_l^r\right) \wedge \Pi \to \Sigma \vee K_iA^r$

(propositional reasoning from n+10)

n+12. $t_{n+11} : (\left(\bigwedge x_j : B_j^r\right) \wedge \left(\bigwedge K_iD_l^r\right) \wedge \Pi \to \Sigma \vee K_iA^r)$

(Lifting Lemma from n+11)

We obtained the ground term t_{n+11} and a derivation of $t_{n+11} : (C \to C')$.

Now, let us consider the $(\Rightarrow \mathsf{J})$ rule:

for $\Delta = \{D_1, \ldots, D_m\}$, $\Sigma = \{E_1, \ldots, E_o\}$, and $\Pi = \{A_1, \ldots, A_r\}$

$$\frac{\mathsf{J}D_1, \ldots, \mathsf{J}D_m \Rightarrow A}{\mathsf{J}D_1, \ldots, \mathsf{J}D_m, A_1, \ldots, A_r \Rightarrow E_1, \ldots, E_o, JA}(\Rightarrow \mathsf{J}) \ .$$

All J's in $\mathsf{J}D_i$'s are negative and belong to different families, so they are realized by distinct proof variables x_i's. Let k be the number of this $(\Rightarrow \mathsf{J})$ rule and let its family be realized by $u_{s_1} + \ldots + u_k + \ldots u_{s_l}$. By induction hypothesis, we have a term t_C and a derivation l_C of

$$t_C : (x_1 : D_1^r \wedge \ldots \wedge x_m : D_m^r \rightarrow A^r).$$

By Lemma 2, we construct a term $s = s(x_1, \ldots, x_m)$ and a derivation l_1 of

$$x_1 : D_1^r \wedge \ldots \wedge x_m : D_m^r \rightarrow s : (x_1 : D_1^r \wedge \ldots \wedge x_m : D_m^r).$$

Note that s does not contain any provisional variables. Now it is easy to append derivations l_C and l_1 (we'll use vector notation for conjunction):

$\ldots$ (*derivation* l_C)

n. $t_C : (\vec{x} : \vec{D^r} \rightarrow A^r)$

$\ldots$ (*derivation* l_1)

m. $\vec{x} : \vec{D^r} \rightarrow s : (\vec{x} : \vec{D^r})$

m+1. $t_C : (\vec{x} : \vec{D^r} \rightarrow A^r) \rightarrow (s : (\vec{x} : \vec{D^r}) \rightarrow t_C \cdot s : A^r)$ (axiom $E1$)

m+2. $s : (\vec{x} : \vec{D^r}) \rightarrow t_C \cdot s : A^r$ (MP from n and m+1)

m+3. $\vec{x} : \vec{D^r} \rightarrow t_C \cdot s : A^r$ (*syllogism* from m and m+2)

. . . (`using axiom` $E3$ several times)

`k.` $\vec{x} : \vec{D^r} \to (u_{s_1}\sigma + \ldots + t_C \cdot s + \ldots u_{s_l}\sigma) : A^r$.

Moreover this derivation is easy to append to obtain derivation l_2 of a formula C' that (modulo permutations) looks like

$$\vec{x} : \vec{D^r} \wedge \vec{A} \to \vec{E} \vee (u_{s_1}\sigma + \ldots + t_C \cdot s + \ldots u_{s_l}\sigma) : A^r$$

(here $\vec{x} : \vec{D^r}$ and $\vec{A}$ stand for conjunctions, and $\vec{E}$ for disjunctions).

We then use the Lifting Lemma to reproduce a ground term $t_{C'}$ and a derivation $l_{C'}$ of

$$t_{C'} : (\vec{x} : \vec{D^r} \wedge \vec{A} \to \vec{E} \vee (u_{s_1}\sigma + \ldots + t_C \cdot s + \ldots u_{s_l}\sigma) : A^r) \ .$$

While lifting l_2 [Brezhnev and Kuznets, 2006], there is no need to lift its initial part l_C since the only formula we use for the second part is $t_C : (\vec{x} : \vec{D^r} \to A^r)$; this formula is easily lifted by adding to l_C the following two formulas:

$t_C : (\vec{x} : \vec{D^r} \to A^r) \to !t_C : t_C : (\vec{x} : \vec{D^r} \to A^r)$, and

$!t_C : t_C : (\vec{x} : \vec{D^r} \to A^r)$,

the latter being the desired lifted version. This procedure produces a ground term because t_C is ground. Also, this modification renders the whole procedure polynomial in the size of the original $\mathsf{S4}^{\mathsf{J}}_n$-derivation. In the original algorithm [Artemov, 1995], each time a $(\Rightarrow \Box)$ rule is processed, most formulas in the initial derivation are replaced by three formulas in the lifted

one, which leads to exponential growth in the number of ($\Rightarrow \Box$) rules. We then append σ by a new substitution: $\sigma = \sigma + \{u_k \leftarrow t_C \cdot s\}$, and apply this substitution throughout the derivation ($\mathsf{S4}_n\mathsf{LP}$ is known to be closed under substitutions). After that, there are no occurrences of u_k remaining in the derivation. As a result, we eliminated one provisional variable.

Final Touch: At the end of the procedure, the entire derivation tree of $\Rightarrow F$ is translated – all ($\Rightarrow \Box$) rules have been processed and there are no provisional variables remaining. Thus, F^r is simply an $\mathsf{S4}_n\mathsf{LP}$ formula. Moreover, we have a Hilbert-style derivation l_t of $t : (\top \rightarrow F^r)$ for some ground term t. To acquire F^r, we do the following:

$\ldots$ (derivation l_t)

`n.` $t : (\top \rightarrow F^r)$

`n+1.` $t : (\top \rightarrow F^r) \rightarrow (\top \rightarrow F^r)$ (axiom $E4$)

`n+2.` $(\top \rightarrow F^r)$ (MP from n and n+1)

`n+3.` $\top$

`n+4.` F^r (MP from n+2 and n+3).

2.6 Notes on implementation

First, we tried to implement the realization algorithm in Prolog. The reasoning behind the use of Prolog is that in proof search problem, Prolog implementations greatly outperform other languages in terms of compactness. For example, provers for intuitionistic logic and S4 can fit on a few pages,

whereas OCaml prover requires many pages of code. This makes Prolog ideal for experimental work because altering a page or two of code is much easier than altering many thousands of lines. We wanted to see if our problem at hand could also benefit from the use of Prolog. But it quickly turned out that either our expertise in Prolog is lacking or Prolog is not the right tool for the job.

The first challenge that we faced was evaluating box families. This involves assigning a family identifier to each box. Proofs have branches, and families grow by transitive extension. We saw two options: either keep families as variables and let Prolog unify them, or assign a unique identifier to each box and maintain a disjoint set of sets (families) of box identifiers that are related (belong to the same family). This disjoint set has to support the following operations: addition of new representatives to sets, merging of sets, and search for a set by its representative. The latter approach was used in our OCaml version of the algorithm, but we wanted to utilize Prolog's powers of unification, search, and backtracking. We tried the former approach but it was not going very well, perhaps due to our lack of expertise. It was not clear how to terminate family variables with literals (and, maybe, operations), so that families from two branches unify, if needed. Families could not remain as variables forever — they had to end up as unique ground terms for the next stage of the realization procedure.

Another problem that we faced with Prolog was its lack of record/structure type. When data items become 6-dimensional tuples with two dimensions

being 4-dimensional tuples, it starts to feel too verbose.

We ended up implementing the procedure in OCaml. The biggest hurdle was, of course, implementing the part that is considered trivial on paper: evaluating box families. As previously stated, we used disjoint set of sets of unique identifiers given to each box. What actually occurs, is that each box is replaced with $\Pr(Provisional, F)$ where $Provisional$ is this unique identifier. The algorithm recursively walks over the proof tree, assigns these identifiers, and collects information about which identifiers fall into which family. At each step representing application of a rule, we have to track how each formula above the line (of the rule) was transformed and then transform the formulas below the line accordingly.

The next major simplification step the paper took is that of skipping much Hilbert-style reasoning. We do the same for all classical reasoning that justifies a shift from rule assumptions to rule conclusion; we say that there is a (fresh) proof constant justifying the implication, and deduce such implications using reflexivity axiom $t : A \to A$.

The rest is a more or less straightforward implementation of the paper algorithm, extended to a multi-agent case.

Using OCaml had a certain convenience — there was already an $\mathsf{S4}^{\mathsf{J}}_n$ prover [Bryukhov, 2006] in MetaPRL Logical Framework, which is also implemented in OCaml. This prover produces Gentzen-style proofs, exactly what polynomial realization procedures needs. It was trivial to connect them.

We also used Ocsigen web server (written in OCaml, (`http://ocsigen.`

org)) to expose our work to casual users over the Web `http://yegor.org/s4nlp/theorem`.

At this point, we realized that the produced proofs were way too long, both in the number of steps and the length of produced formulas. We partially address both.

First of all, as suggested by Melvin Fitting, we introduced two one-step rules: Deduction and Lifting. After each application of the Lifting lemma or Deduction Theorem, we retain the full chain of reasoning for validation purposes, but collapse them for purposes of display. Therefore, one sees the following:

$$\begin{array}{lll} k. & A_1, \ldots, A_m, \ldots, A_n \vdash & B \\ k+1. & A_{m+1}, \ldots, A_n \vdash & A_m \to \ldots \to A_1 \to B \text{ (by Deduction)} \end{array}$$

and

$$\begin{array}{lll} k. & x_1 : A_1, \ldots, x_n : A_n \vdash & B \\ k+1. & x_1 : A_1, \ldots, x_n : A_n \vdash & c(x_1, \ldots, x_n) : B \text{ (by Lifting).} \end{array}$$

Second, we assigned fresh (shortcut) constants to each proof term appearing in the proof and with length more than 5. We list these assignments at the end of the proof and provide a hyperlink from each occurrence of such a constant to its definition.

We also inserted dummy steps to mark the boundaries between individual Gentzen proofs steps and certain stages of $\Rightarrow \Box$ rule realization. Such dummy

steps are labeled with Gentzen rules or realization stage names instead of Hilbert rules or axioms. These dummy rules are also rendered with normal font size and all intermediate steps – with a smaller font.

2.7 Examples

In this section, we will go over a few simple examples first, followed by several classical puzzles.

2.7.1 Graphs

Some of the examples of realization are accompanied with the three types of graphs.

The first graph shows the growth of the number of steps in a Hilbert-style proof as a function of the number of steps in the original Gentzen-style proof. According to [Brezhnev and Kuznets, 2006] we should observe $O(n^2)$ - dependency on this graph.

The second graph shows the growth of the total length of all formulas in a Hilbert-style proof as a function of the total length of all formulas in the original Gentzen-style proof. According to [Brezhnev and Kuznets, 2006] we should observe $O(n^6)$ - dependency on this graph.

And the last graph shows the growth of the external (outer) terms' sizes in a Hilbert-style proof as a function of the number of steps in the original Gentzen-style proof. According to [Brezhnev and Kuznets, 2006] we should

observe $O(n^2)$ - dependency on this graph.

One can see the bumps on the graphs. The bumps are the result of $(\Rightarrow \Box)$ rule realization. We observe that all other rules are linear in the number of steps.

When a Gentzen-style proof has branches, we show them in different colors, and the longest branch on the graph goes all the way to the root of the proof.

Now let us proceed to specific examples.

2.7.2 Self-Referential Example

$\mathsf{J}\neg(s \to \mathsf{J}s) \Rightarrow$ (self referential example, [Brezhnev and Kuznets, 2006]). The linearized Gentzen $\mathsf{S4}_n^{\mathsf{J}}$ proof is

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{a, J\neg(a \to Ja) \Rightarrow Ja, a}\,(\text{Axiom})}{J\neg(a \to Ja) \Rightarrow a \to Ja, a}\,(\Rightarrow\to)}{\neg(a \to Ja), J\neg(a \to Ja) \Rightarrow a}\,(\neg\Rightarrow)}{J\neg(a \to Ja) \Rightarrow a}\,(\Box\Rightarrow)}{a, J\neg(a \to Ja) \Rightarrow Ja}\,(\Rightarrow J)}{J\neg(a \to Ja) \Rightarrow (a \to Ja)}\,(\Rightarrow\to)}{\neg(a \to Ja), J\neg(a \to Ja) \Rightarrow}\,(\neg\Rightarrow)}{J\neg(a \to Ja) \Rightarrow}\,(\Box\Rightarrow)$$

A corresponding Hilbert-style $\mathsf{S4}_n\mathsf{LP}$ proof of this sequent is given below, where Tt stand for constant terms for certain propositional tautologies and, in the case of $(\Box\Rightarrow)$ rule, simple propositional consequences of $t : A \to A$

axiom. C_1 through C_{20} are witnesses for axioms; the numbering of the axioms can be found in Appendix B.

1. $\vdash (V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s))) \rightarrow !V_1 : (V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s)))))$

[Axiom 14]

2. $\vdash (\mathbf{V_1} : (\neg(\mathbf{s} \rightarrow (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s}))) \rightarrow !\mathbf{V_1} : (\mathbf{V_1} : (\neg(\mathbf{s} \rightarrow (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})))))$

[lemma3, 1 hyp]

3. $\vdash Tt : (((s \wedge V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s)))) \rightarrow (s \vee (C_{41}!V_1) : (s))))$

[CS for classical tautology or $t : A \rightarrow A$]

4. $\vdash \mathbf{Tt} : (((\mathbf{s} \wedge \mathbf{V_1} : (\neg(\mathbf{s} \rightarrow (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})))) \rightarrow (\mathbf{s} \vee (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s}))))$

[Gentzen Axiom]

5. $\vdash Tt : ((((s \wedge V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s)))) \rightarrow (s \vee (C_{41}!V_1) : (s))) \rightarrow$
$(V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s))) \rightarrow (s \vee (s \rightarrow (C_{41}!V_1) : (s))))))$

[CS for classical tautology or $t : A \rightarrow A$]

6. $\vdash (Tt : ((((s \wedge V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s)))) \rightarrow (s \vee (C_{41}!V_1) : (s))) \rightarrow$
$(V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s))) \rightarrow (s \vee (s \rightarrow (C_{41}!V_1) : (s))))) \rightarrow$
$(Tt : (((s \wedge V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s)))) \rightarrow (s \vee (C_{41}!V_1) : (s)))) \rightarrow$
$(TtTt) : ((V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s))) \rightarrow (s \vee (s \rightarrow (C_{41}!V_1) : (s))))))))$

[Axiom 12]

7. $\vdash (Tt : (((s \wedge V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s)))) \rightarrow (s \vee (C_{41}!V_1) : (s)))) \rightarrow$
$(TtTt) : ((V_1 : (\neg(s \rightarrow (C_{41}!V_1) : (s))) \rightarrow (s \vee (s \rightarrow (C_{41}!V_1) : (s))))))$

[MP on 5 and 6]

8. $\vdash (TtTt) : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \vee (s \to (C_{41}!V_1) : (s)))))$

[MP on 4 and 7]

9. $\vdash (\mathbf{TtTt}) : ((\mathbf{V_1} : (\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s}))) \to (\mathbf{s} \vee (\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})))))$

[**by** $(\Rightarrow\to)$]

10. $\vdash Tt : (((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \vee (s \to (C_{41}!V_1) : (s)))) \to$
$((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to s)))$

[CS for classical tautology or $t : A \to A$]

11. $\vdash (Tt : (((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \vee (s \to (C_{41}!V_1) : (s)))) \to$
$((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to s))) \to$
$((TtTt) : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \vee (s \to (C_{41}!V_1) : (s))))) \to$
$(Tt(TtTt)) : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to s))))$

[Axiom 12]

12. $\vdash ((TtTt) : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \vee (s \to (C_{41}!V_1) : (s))))) \to$
$(Tt(TtTt)) : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to s)))$

[MP on 10 and 11]

13. $\vdash (Tt(TtTt)) : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to s))$

[MP on 9 and 12]

14. $\vdash (\mathbf{Tt}(\mathbf{TtTt})) : (((\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})) \wedge \mathbf{V_1} : (\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})))) \to$
$\mathbf{s}))$ [**by** $(\neg \Rightarrow)$]

15. $\vdash Tt : ((((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to s) \to$
$(V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to s)))$ [CS for classical tautology or $t : A \to A$]

16. $\vdash (Tt : ((((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to s) \to$
$(V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to s))) \to ((Tt(TtTt)) : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge$
$V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to s)) \to C_{41} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to s))))$
[Axiom 12]

17. $\vdash ((Tt(TtTt)) : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to$
$s)) \to C_{41} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to s)))$ [MP on 15 and 16]

18. $\vdash C_{41} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to s))$ [MP on 14 and 17]

19. $\vdash C_{41} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to s))$ [by $\Box \Rightarrow$]

20. $\vdash (C_{41} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to s)) \to (!V_1 : (V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s)))$ [Axiom 12]

21. $\vdash (!V_1 : (V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s))$ [MP on 19 and 20]

22. $\vdash ((!V_1 : (V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s)) \to (V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s))) \to (!V_1 : (V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s))))$
[Axiom 1]

23. $\vdash (V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (!V_1 : (V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to$
$(C_{41}!V_1) : (s)))$ [MP on 21 and 22]

24. $\vdash ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (!V_1 : (V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to$
$(C_{41}!V_1) : (s))) \to ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to !V_1 : (V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s))))) \to (V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (C_{41}!V_1) : (s))))$ [Axiom 2]

25. $\vdash ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to !V_1 : (V_1 : (\neg(s \to (C_{41}!V_1) : (s))))) \to$
$(V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (C_{41}!V_1) : (s)))$
[MP on 23 and 24]

26. $\vdash (V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (C_{41}!V_1) : (s))$ [MP on 2 and 25]

27. $\vdash (\mathbf{V_1} : (\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s}))) \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s}))$ [**syllogism**]

28. $\vdash Tt : (((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (C_{41}!V_1) : (s)) \to ((s \wedge V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s)))$ [CS for classical tautology or $t : A \to A$]

29. $\vdash (Tt : (((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (C_{41}!V_1) : (s)) \to ((s \wedge V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s)))) \to ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to$
$(C_{41}!V_1) : (s)) \to ((s \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s))))$
[Axiom 13]

30. $\vdash ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (C_{41}!V_1) : (s)) \to ((s \wedge V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s)))$ [MP on 28 and 29]

31. $\vdash ((s \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s))$ [MP on 27 and 30]

32. $\vdash C_{42} : (((s \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s)))$ [Lift]

33. $\vdash \mathbf{C_{42}} : (((\mathbf{s} \wedge \mathbf{V_1} : (\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})))) \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})))$ [**by** $(\Rightarrow \mathbf{J})$]

34. $\vdash Tt : ((((s \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s)) \to (V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s))) \to (s \to (C_{41}!V_1) : (s)))))$
[CS for classical tautology or $t : A \to A$]

35. $\vdash (Tt : ((((s \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s)) \to (V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s))) \to (s \to (C_{41}!V_1) : (s))))) \to (C_{42} : (((s \wedge V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s))) \to C_{43} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to$
$(s \to (C_{41}!V_1) : (s))))))$ [Axiom 12]

36. $\vdash (C_{42} : (((s \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to (C_{41}!V_1) : (s))) \to$
$C_{43} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \to (C_{41}!V_1) : (s)))))$ [MP on 34 and 35]

37. $\vdash C_{43} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \to (C_{41}!V_1) : (s))))$

[MP on 33 and 36]

38. $\vdash \mathbf{C_{43}} : ((\mathbf{V_1} : (\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s}))) \to (\mathbf{s} \to (\mathbf{C_{41}V_1}) : (\mathbf{s}))))$

[**by** $(\Rightarrow\to)$]

39. $\vdash Tt : (((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \to (C_{41}!V_1) : (s))) \to ((\neg(s \to$
$(C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot)))$

[CS for classical tautology or $t : A \to A$]

40. $\vdash (Tt : (((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \to (C_{41}!V_1) : (s))) \to ((\neg(s \to$
$(C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot))) \to (C_{43} : ((V_1 : (\neg(s \to$
$(C_{41}!V_1) : (s))) \to (s \to (C_{41}!V_1) : (s)))) \to C_{44} : (((\neg(s \to$
$(C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot))))$ [Axiom 12]

41. $\vdash (C_{43} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to (s \to (C_{41}!V_1) : (s)))) \to$
$C_{44} : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot)))$

[MP on 39 and 40]

42. $\vdash C_{44} : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot))$

[MP on 38 and 41]

43. $\vdash \mathbf{C_{44}} : (((\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})) \wedge \mathbf{V_1} : (\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s})))) \to \bot))$

[**by** $(\neg \Rightarrow)$]

44. $\vdash Tt : ((((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot) \to$
$(V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to \bot)))$ [CS for classical tautology or $t : A \to A$]

45. $\vdash (Tt : ((((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot) \to$
$(V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to \bot))) \to (C_{44} : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge$
$V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot)) \to C_{45} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to \bot))))$
[Axiom 12]

46. $\vdash (C_{44} : (((\neg(s \to (C_{41}!V_1) : (s)) \wedge V_1 : (\neg(s \to (C_{41}!V_1) : (s)))) \to \bot)) \to$
$C_{45} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to \bot)))$ [MP on 44 and 45]

47. $\vdash C_{45} : ((V_1 : (\neg(s \to (C_{41}!V_1) : (s))) \to \bot))$ [MP on 43 and 46]

48. $\vdash \mathbf{C_{45}} : ((\mathbf{V_1} : (\neg(\mathbf{s} \to (\mathbf{C_{41}}!\mathbf{V_1}) : (\mathbf{s}))) \to \bot))$ [**by** $(\Box \Rightarrow)$]

Constants:

$C_{41} = (Tt\ (Tt\ (Tt\ Tt)))$

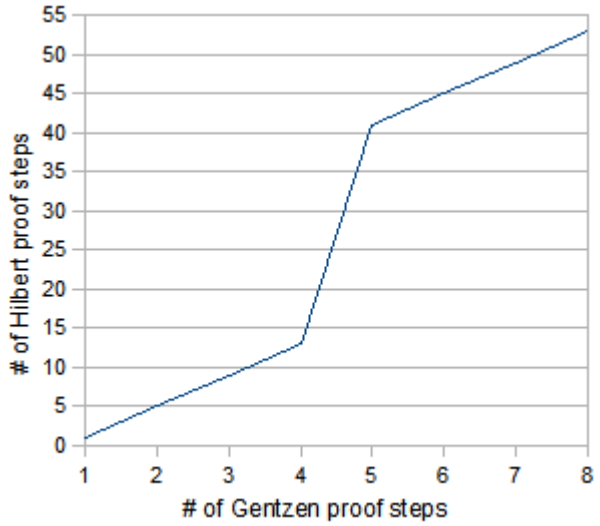
$C_{42} = ((C_{13}\ !Tt)\ ((C_2\ (C_1\ (C_{12}\ !(Tt\ (Tt\ (Tt\ Tt))))))\ C_{14}))$

$C_{43} = (Tt\ ((C_{13}\ !Tt)\ ((C_2\ (C_1\ (C_{12}\ !(Tt\ (Tt\ (Tt\ Tt))))))\ C_{14})))$

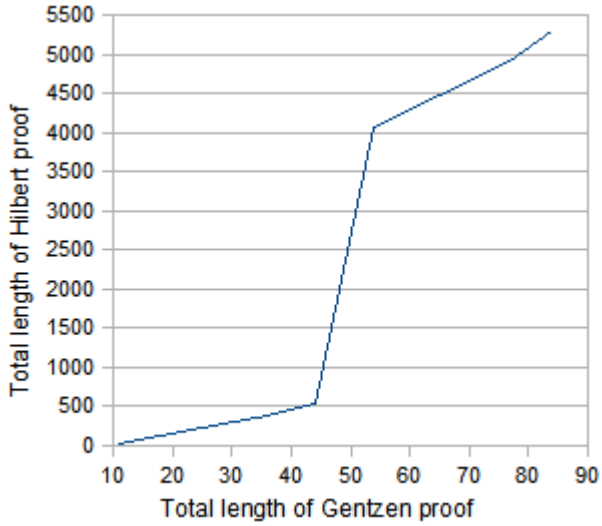
$C_{44} = (Tt\ (Tt\ ((C_{13}\ !Tt)\ ((C_2\ (C_1\ (C_{12}\ !(Tt\ (Tt\ (Tt\ Tt))))))\ C_{14}))))$

$C_{45} = (Tt\ (Tt\ (Tt\ ((C_{13}\ !Tt)\ ((C_2\ (C_1\ (C_{12}\ !(Tt\ (Tt\ (Tt\ Tt))))))\ C_{14})))))$

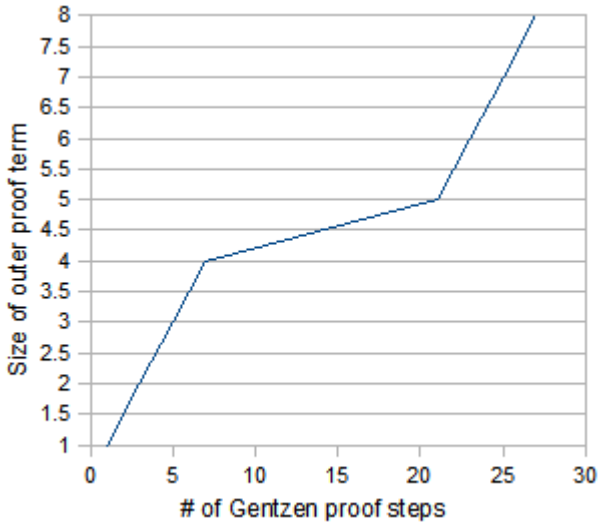
The first graph:



The second graph:



The third graph:



2.7.3 Smaller examples

We also tried the following trivial examples. We do not show the proofs because they are not particularly interesting.

- $\mathsf{J}A \wedge \mathsf{J}B \Rightarrow \mathsf{J}(A \wedge B)$

  realization: $(V_0 : A \wedge V_1 : B) \rightarrow (C_{42}\ ((C_5\ !V_0)\ !V_1)) : (A \wedge B)$

- $\mathsf{J}(A \wedge B) \Rightarrow \mathsf{J}A \wedge \mathsf{J}B$

  realization:

  $V_0 : (A \wedge B) \rightarrow (((T_1\ (Tt\ Tt))\ !V_0) : A \wedge ((T_4\ (Tt\ Tt))\ !V0) : B)$,

  with proof constants for simple non-propositional tautologies:

  $T_1 : ((A \wedge B) \wedge V_0 : (A \wedge B) \rightarrow A) \rightarrow (V_0 : (A \wedge B) \rightarrow A)$,

  $T_4 : ((A \wedge B) \wedge V_0 : (A \wedge B) \rightarrow B) \rightarrow (V_0 : (A \wedge B) \rightarrow B)$

- $\mathsf{J}(A \wedge B) \Rightarrow \mathsf{J}A$

  realization: $V0 : (A \wedge B) \rightarrow ((T_1\ (Tt\ Tt))\ !V_0) : A$,

  with proof constant for simple non-propositional tautologies:

  $T_1 : ((A \wedge B) \wedge V_0 : (A \wedge B) \rightarrow A) \rightarrow (V_0 : (A \wedge B) \rightarrow A)$

- $\mathsf{J}(A \vee B) \Rightarrow A, B$

  realization: $V_0 : (A \vee B) \rightarrow (A \vee B)$

- $\mathsf{J}A \vee \mathsf{J}B \Rightarrow \mathsf{J}(A \vee B)$

  realization:

  $(V_0 : A \vee V_2 : B) \rightarrow (((Tt\ (T_2\ Tt))\ !V_0) + ((Tt\ (T_5\ Tt))\ !V_2)) : (A \vee B)$,

  with proof constants for simple non-propositional tautologies:

  $T_2 : ((A \wedge V_0 : A) \rightarrow (A \vee B)) \rightarrow (V_0 : A \rightarrow (A \vee B))$,

  $T_5 : ((B \wedge V_2 : B) \rightarrow (A \vee B)) \rightarrow (V_2 : B \rightarrow (A \vee B))$

- $\mathsf{J}A \Rightarrow \mathsf{J}(A \vee B)$

  realization: $V_0 : A \rightarrow ((Tt\ (T_2\ Tt))\ !V_0) : (A \vee B)$,

  with proof constant for simple non-propositional tautologies:

  $T_2 : ((A \wedge V_0 : A) \rightarrow (A \vee B)) \rightarrow (V_0 : A \rightarrow (A \vee B))$

- $A, \neg A \Rightarrow$

  realization: $(A \wedge \neg A) \rightarrow$

- $A \Rightarrow \neg\mathsf{J}\neg A$

realization: $A \rightarrow \neg V_0 : (\ \neg A)$

- $\Rightarrow K_1(A \rightarrow A)$

  realization: $\neg\bot \rightarrow K1(A \rightarrow A)$

- $\mathsf{J}A \Rightarrow K_1\ A$

  realization: $V_0 : A \rightarrow K_1\ A$

- $K_2\ \mathsf{J}A \Rightarrow K_1\ A$

  realization: $K_2\ V_0 : A \rightarrow K_1\ A$

- $\mathsf{J}\ K_2\ \mathsf{J}A \Rightarrow K_1\ A$

  realization: $V_2 : (K_2\ V_0 : A) \rightarrow K_1\ A$

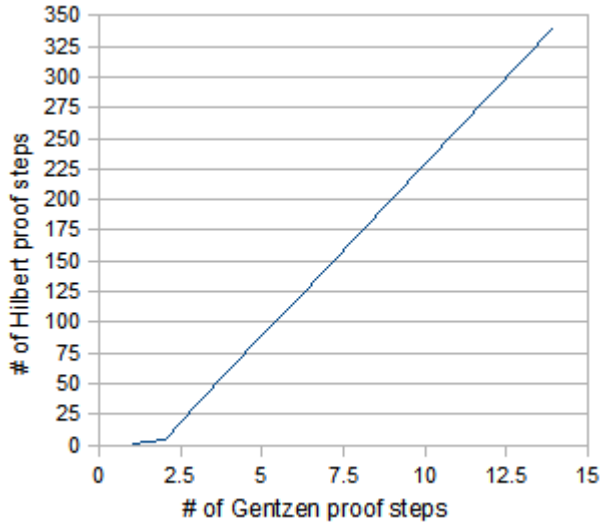
2.7.4 Complexity of $(\Rightarrow \Box)$ rules

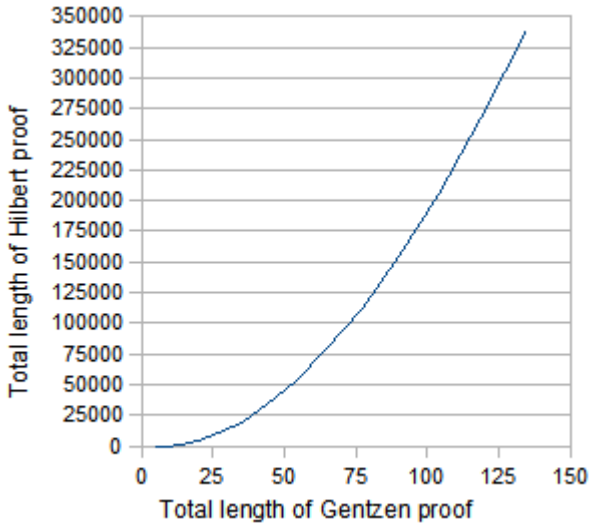
As mentioned, realization of the $(\Rightarrow \Box)$ rules causes complexity leaps in derivations. We decided to take a look at the following two examples, hwere there are many of J's (K's) are introduced to the right.

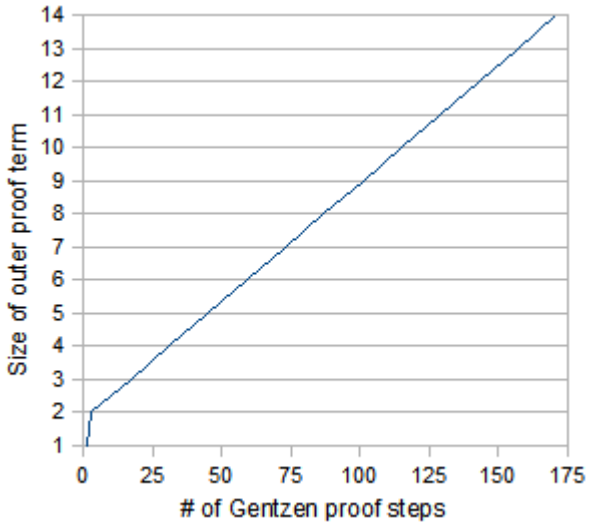
(A) $\mathsf{J}A \rightarrow \mathsf{JJJJJJJJJJJ}A$

The realized formula: $V_0 : A \rightarrow C_{51}!V_0 : (C_{50}!V_0 : (C_{49}!V_0 : (C_{48}!V_0 : (C_{47}!V_0 : (C_{46}!V_0 : (C_{45}!V_0 : ((C_{44}!V_0) : (C_{43}!V_0 : (C_{42}!V_0 : (C_{41}!V_0 : (((Tt\ Tt)\ !V_0) : A)))))))))))$.

Here are the graphs:


350
325
300
275
250
225
200
175
150
125
100
75
50
25
0
# of Hilbert proof steps
0
2.5
5
7.5
10
12.5
15
# of Gentzen proof steps

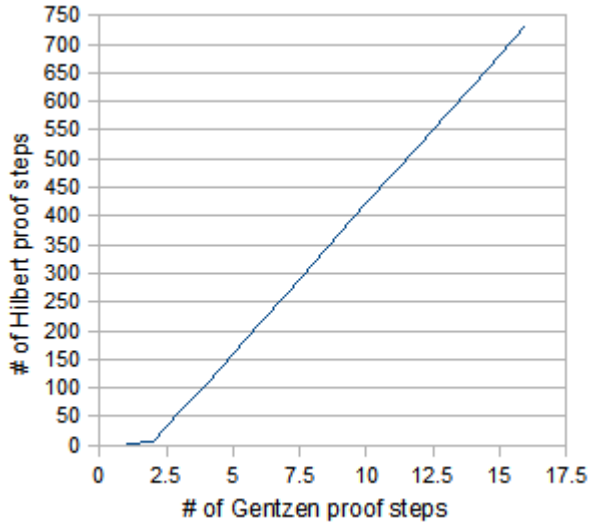

350000
325000
300000
275000
250000
225000
200000
175000
150000
125000
100000
75000
50000
25000
0
Total length of Hilbert proof
0
25
50
75
100
125
150
Total length of Gentzen proof

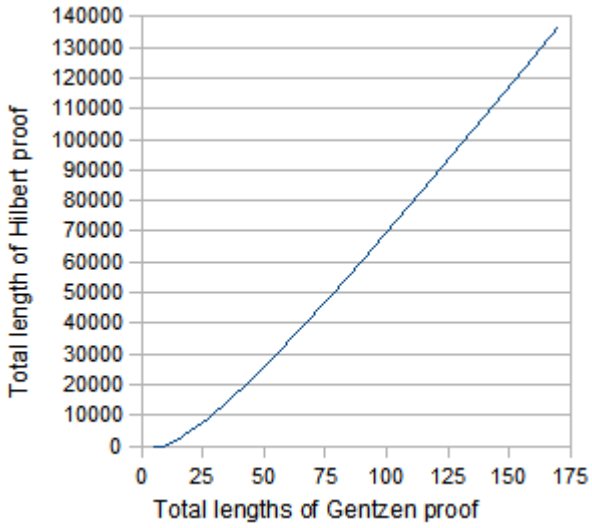


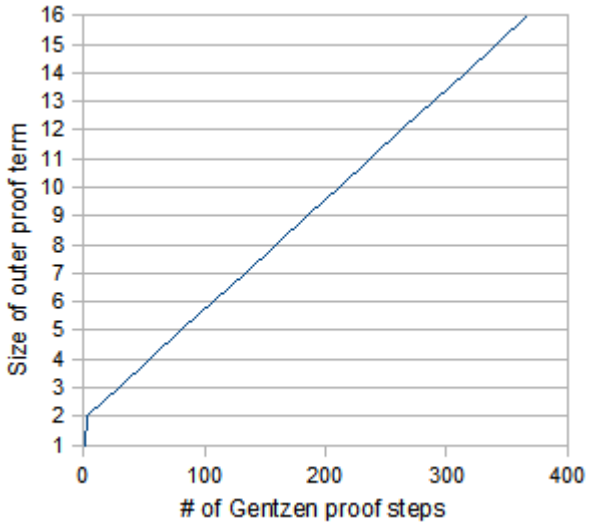
It may not be visible on the second graph but in fact, the total length of the formula here grows faster than a quadratic function and slower than a cubic function.

(B) $K_1A \to K_1K_1K_1K_1K_1K_1K_1K_1K_1K_1K_1K_1K_1A$

Here are the graphs:


750
700
650
600
550
500
450
400
350
300
250
200
150
100
50
0
# of Hilbert proof steps
0
2.5
5
7.5
10
12.5
15
17.5
# of Gentzen proof steps


140000
130000
120000
110000
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Total length of Hilbert proof
0
25
50
75
100
125
150
175
Total lengths of Gentzen proof



2.7.5 Realization of the Wise Men Puzzle

The Wise Men puzzle is a classic puzzle studied in epistemology [Fagin *et al.*, 1995].

There are three wise men that can see each other. It is common knowledge that there are two white hats and three red hats. The king puts a hat on the head of each wise man and asks each in turn if he knows the color of the hat on his head. The first wise man says that he does not know; the second man says that he does not know; then the third wise man says that he knows.

The implementation of the Wise Men Puzzle: In the first three lines,

```
let m1 = Atom(add "m1")
let m2 = Atom(add "m2")
let m3 = Atom(add "m3") ,
```

we create three propositional variables which are reserved for three wise men wearing hats, where m_i encodes that i^{th} man wears a red hat. Then, in

```
let kw i a = Or(Box(Modal i, a), Box(Modal i, Neg a)) ,
```

we create a helper-function `kw(i,a)` $= K_i a \vee K_i \neg a$, where `kw` stands for '**k**nows **w**hether a or $\neg a$.'

For example, `kw(1,`m_2`)` means that the first man knows whether or not he wears a red hat. The next line,

```
let kao = And(kw 1 m2, And(kw 1 m3, And(kw 2 m1, kw 2 m3))) ,
```

creates a propositional helper-formula `kao` $= (K_1 m_2 \vee K_1 \neg m_2) \wedge (K_1 m_3 \vee K_1 \neg m_3) \wedge (K_2 m_1 \vee K_2 \neg m_1) \wedge (K_2 m_3 \vee K_2 \neg m_3)$, where `kao` stands for '**k**nows **a**bout **o**thers,' and states that m_1 knows which hats m_2 and m_3 wear, and m_2 knows which hats m_1 and m_3 wear. We don't state knowledge of the third man because it is not needed for the deduction. Initially, we have the following state:

```
let w0 = And(Box(Modal 0, kao), Box(Modal 0,
          Or(m1, Or(m2, m3)))) ,
```

i.e., `w0` $=$ `J`$kao \wedge$ `J`$(m_1 \vee m_2 \vee m_3)$, which simply states that `kao` is common knowledge and it is common knowledge that at least one of these men wears a red hat. This is the way we encode the fact that there are two white and three red hats, i.e., all three hats worn cannot be white.

After two men speak, the situation is:

```
let w2 = And(w0, And(Box(Modal 0, Neg(kw 1 m1)),
                     Box(Modal 0, Neg(kw 2 m2)))) ,
```

i.e., $\texttt{w2} = \texttt{w0} \wedge \mathsf{J}\neg\texttt{kw}(1, m_1) \wedge \mathsf{J}\neg\texttt{kw}(2, m_2)$ — in addition to `w0`, it is common knowledge that the first man does not know which hat he is wearing, and it is common knowledge that the second man does not know which hat he is wearing either.

Then,

```
let _=
   let h = w2 in
   let c = Box(Modal 0, m3) in
   let htm = formula2term h in
   let ctm = formula2term c in .
```

Convert formulas `w2` and $\mathsf{J}m_3$ (it is common knowledge that third man wears a red hat) into MetaPRL terms expected by `gen_prover`:

```
   let infs = gen_prover (Some 100) Jlogic_sig.S4 [htm] [ctm] in
```

where we ask `gen_prover` to find a Gentzen-style $\mathsf{S4}^{\mathsf{J}}_n$ proof of $\texttt{w2} \Rightarrow \mathsf{J}m_3$.

`gen_prover` returns a tree of rule names, possibly with parameters, but our realization code expects a more complete form, namely a tree with full information about sequents in every node/step of proof. So we convert former to latter with function `fill_sequents` using the final sequent as a starting point. Here:

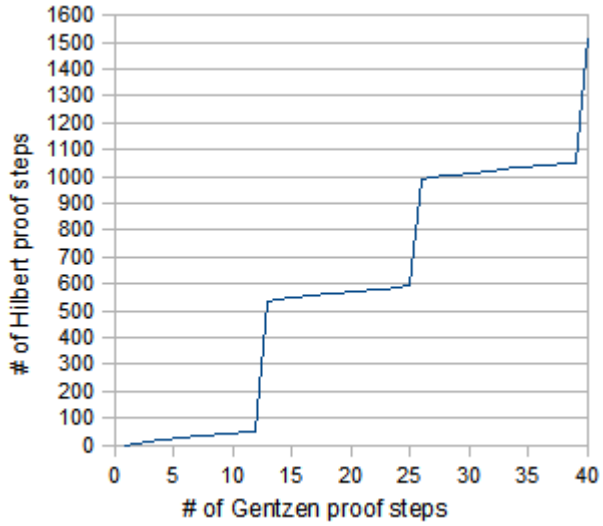
```
match infs with
   [inf] ->
      printf "Filling in sequents\n";
      let g = fill_sequents (FSet.singleton h)
              (FSet.singleton c) inf in
      realize g
 | _ -> raise (Invalid_argument "resulting inference has more than
               one root") .
```

the returned proof is fed to function `fill_sequents` along with hypotheses `w2` and conclusion Jm_3.

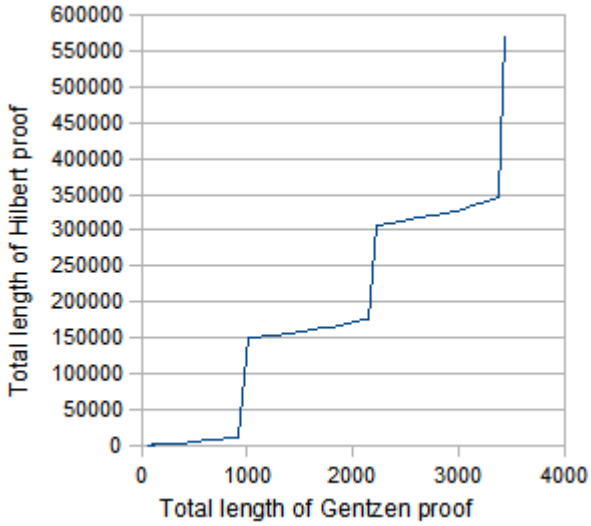
This more complete proof is provided to the function `realize` for the realization itself. The resulting realized formula is

$\Big(\big(\big(V_0 : (\text{kw } 1\ m_2 \wedge (\text{kw } 1\ m_3 \wedge (\text{kw } 2\ m_1 \wedge \text{kw } 2\ m_3))) \ \wedge$
$V_1 : (m_1 \vee (m_2 \vee m_3))\big) \wedge V_2 : (\neg(\text{kw } 1\ m_1))\big) \wedge V_3 : (\neg(\text{kw } 2\ m_2))\Big) \rightarrow$
$(C_{71}\ ((C_5\ ((C_5\ ((C_5\ !V_0)\ !V_1))\ !V_2))\ !V_3)) : m_3.$

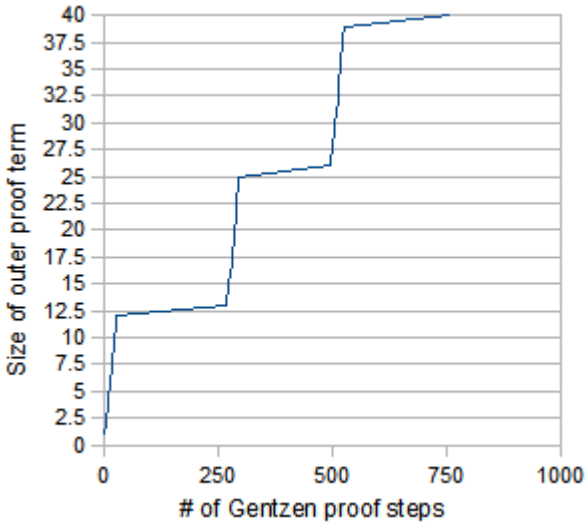
The first graph:



The second graph:



The third graph:



The online realization can be seen at `http://yegor.org/s4nlp/wisemen`.

2.7.6 Realization of the Wise Girls Puzzle

Two Wise Girls puzzle [Yasugi and Oda, 2002] is a simpler variant of the three wise men puzzle.

Two girls are seated one behind the other, facing the same direction. The girls are put white hats on their heads. The first girl can see the second girls hat but not conversely, and neither can see her own hat. The girls are told by the observer that at least one of them wears a white hat. The observer then asks the first girl: "Do you know if your hat is white?" She answers: "No! I do not know." The observer then asks the second girl the same question and she answers: "Yes, I know."

For this puzzle, the online realization procedure was used.

Two propositional variables, $\mathtt{g}_1$ and $\mathtt{g}_2$, are used for two girls, where $\mathtt{g}_i$ stands for 'i^{th} girl wears a white hat.' The list of hypotheses is

```
J (kw 1 g2), J(not kw 1 g1), J(g1 or g2) ,
```

where the first one means that it is common knowledge that the first girl knows what color hat is on the second girl; the second one states that it is common knowledge that the first girl does not know what color hat she is wearing; and the last one states that it is common knowledge that at least one of the girls wears a white hat.

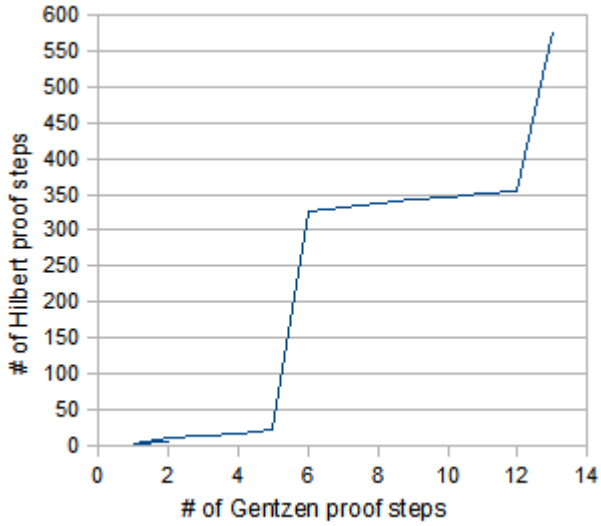
The conclusion states that the second girl knows that she is wearing a white hat:

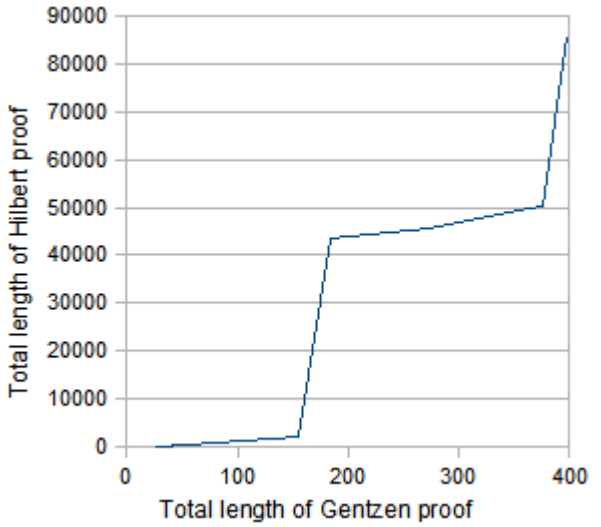
```
K 2 g2 .
```

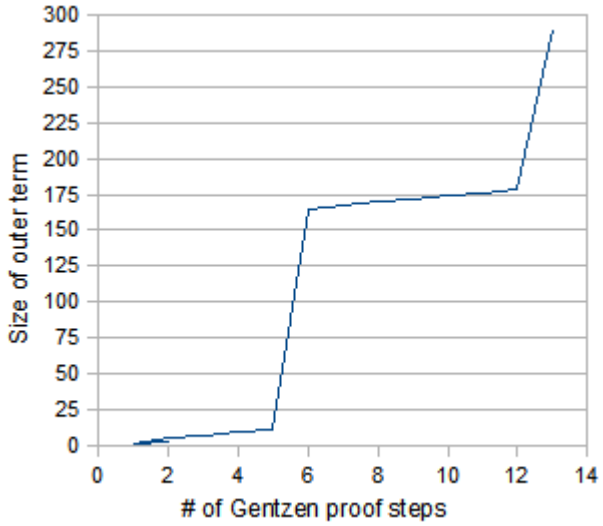
Provided with the above hypotheses and conclusion, the algorithm produces a proof term. The realized formula is

$(((V_0 : (g_1 \vee g_2) \wedge V_1 : \text{kw } 1\ g_2) \wedge V_2 : \neg(\text{kw } 1\ g_1))) \rightarrow K_2\ g_2$.

Graphs for the wise girls:


600
550
500
450
400
350
300
250
200
150
100
50
0
# of Hilbert proof steps
0
2
4
6
8
10
12
14
# of Gentzen proof steps


90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Total length of Hilbert proof
0
100
200
300
400
Total length of Gentzen proof



2.7.7 Realization of the Muddy Children Puzzle

N children are playing together [Fagin *et al.*, 1995]. Their mother tells them that if they get dirty, there will be severe consequences. Now, it so happens that during their play, some of the children, say k of them, get mud on their foreheads. Each can see mud on the others, but not on his own forehead, so no one says anything. Along comes their father who says, "At least one of you has mud on your forehead," thus expressing a fact known to each of them before he speaks (if $k > 1$). The father then asks the following question repeatedly : "Do any of you know whether or not you have mud on your own forehead?" Assuming that the children are all perceptive, intelligent, truthful, and that they answer simultaneously, what will happen? There is a 'proof' that the first $k-1$ times he asks the question, they will all answer "No," but the k^{th} time, the children with muddy foreheads will all answer

"Yes."

For this puzzle, a straightforward epistemic logic formalization can lead to inconsistent sets of hypotheses [Bryukhov, 2006; Fitting, 2006; Fitting, 2007a]. For example, the sets

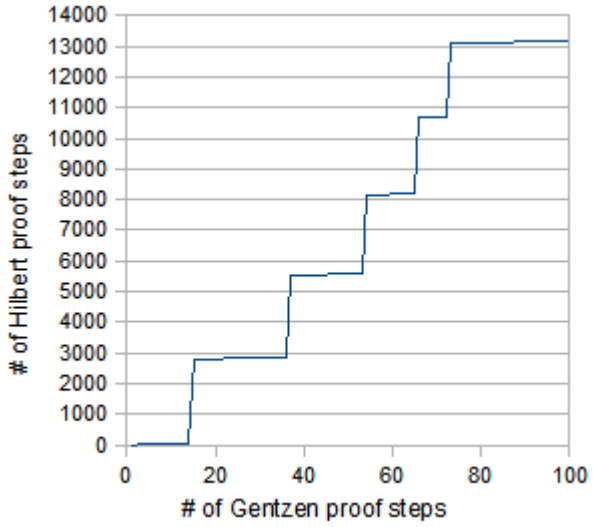
$$J(c_1 \vee c_2 \vee \ldots \vee c_n),$$

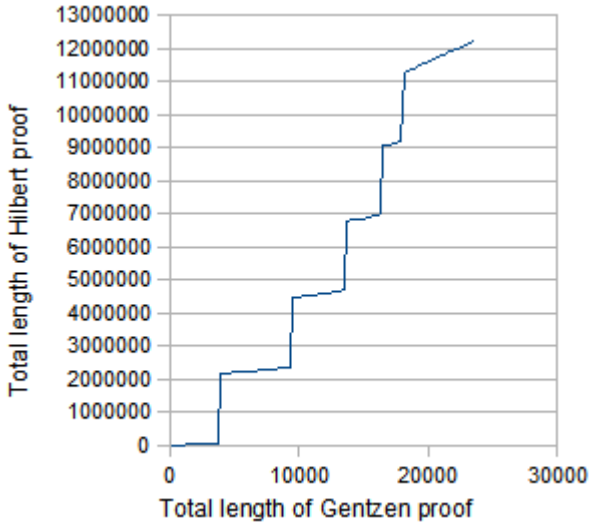
$$J(\text{kw } i \; C_j) \text{ for all } i \neq j,$$

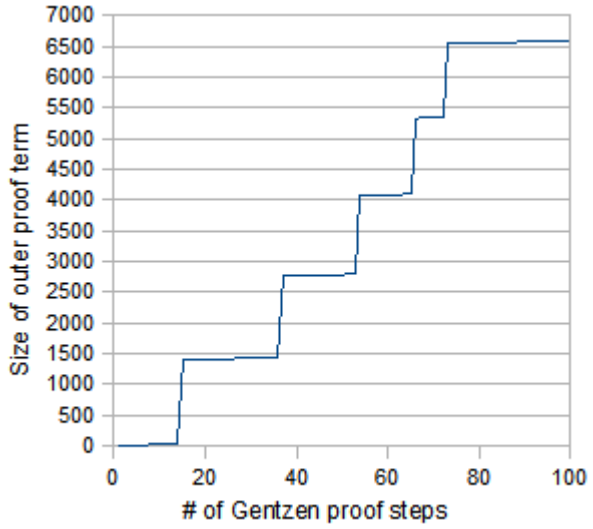
$$J\neg(\text{kw } i \; C_i) \text{ for all } i,$$

are inconsistent for all $n \geq 2$.

For the sake of observing the complexity of the realization process, one can find the complexities of this contradictory formalization for five children in the following graphs.







[McCarthy *et al.*, 1978] uses a version of logic similar to $\mathsf{S4}_n^{\mathsf{J}}$ with all modalities graded by time to present a model-based solution of Muddy Children. This solution has a model hence it avoids introduction of a contradiction. For reference we present it in Appendix A.

Here we present three of the possible formalizations of the Muddy Chil-

dren puzzle for three children, using McCarthy's idea. Graphs for each complexity measure are given after the last formalization, we merged them for the ease of comparison.

We use the following notation:

- K-modality has two indices now: time and agent, i.e. $K_{t,a}A$ stands for 'At time t, agent a knows A.'
- $E_tA \ ::= \ K_{t,a_1}A \wedge \ldots \wedge K_{t,a_n}$ stands for 'Everybody knows A at time t.'
- $\mathtt{kw}_{t,a} \ A \ ::= \ K_{t,a}A \vee K_{t,a}\neg A$, i.e. 'At the moment t agent a knows whether or not A holds.'

We use the McCarthy's idea only, not his system. We stay in $\mathsf{S4}^{\mathsf{J}}_n$, no new axioms or rules related to time are introduced, and $K_{t,a}$ is just a syntactic sugar for $K_{(t-1)n+a}$, where n is the number of agents.

Without loss of generality, let us assume that the first and third children are dirty, and the second one is clean, i.e., $c_1, \neg c_2, c_3$.

(A) Longer version

Hypotheses:

1. $c_1 \wedge \neg c_2 \wedge c_3$
2. $\mathtt{kw}_{3,2} \ c_1$;
3. $\mathtt{kw}_{3,2} \ c_3$;

4. $\mathtt{kw}_{3,2}\ (\mathtt{kw}_{2,1}\ \ c_1)$;

5. $K_{3,2}($

6. $\quad (\mathtt{kw}_{2,1}\ c_3) \wedge (\mathtt{kw}_{2,1}\ c_2) \wedge$

7. $\quad K_{2,1}($

8. $\quad\quad \neg(\mathtt{kw}_{1,1}\ c_1) \wedge \neg(\mathtt{kw}_{1,2}\ c_2) \wedge \neg(\mathtt{kw}_{1,3}\ c_3) \wedge$

9. $\quad\quad (E_1(c_1 \vee c_2 \vee c_3)) \wedge$

10. $\quad\quad (\mathtt{kw}_{1,1}\ c_2) \wedge (\mathtt{kw}_{1,1}\ c_3) \wedge$

11. $\quad\quad (\mathtt{kw}_{1,2}\ c_1) \wedge (\mathtt{kw}_{1,2}\ c_3) \wedge$

12. $\quad\quad (\mathtt{kw}_{1,3}\ c_1) \wedge (\mathtt{kw}_{1,3}\ c_2)$

$\quad))$

13. $K_{3,2}((c_1 \wedge c_2 \wedge c_3) \rightarrow \neg\ \mathtt{kw}_{2,1}\ \ c_1)$

Conclusion: $\mathtt{kw}_{3,2}\ c_2$

Lines 2–3 At moment 3, the second child knows whether or not the first and the third children are muddy.

Line 4 At moment 3, the second child knows whether or not the first child at moment 2 knew wheather or not he was muddy.

Line 5 At moment 3, the second child knows ...

[**Line 6**] ... at moment 2, the first child knew whether the second and the third children are muddy or not, and

[**Line 7**] ... at moment 2, the first child knows ...

[**Line 8**] ... at the first moment, nobody knew whether or not they were muddy, and

[**Line 9**] ... at the first moment, everybody knew that at least one of them was muddy, and

[**Line 10**] ... at the first moment, the first child knew whether or not the second and third children were muddy, and

[**Line 11**] ... at the first moment, the second child knew whether or not the first and the third children were muddy, and

[**Line 12**] ... at the first moment, the third child knew whether or not the first and the second children were muddy.

Line 13 At moment 3, the second child knows that if all of them are dirty, the first child at moment 2 did not know whether or not he was muddy.

Conclusion At moment 3, the second child knew whether or not he was muddy.

The last hypothesis cannot be relaxed because in situations where a child says that (s)he does not know if (s)he is muddy, (s)he cannot really derive it in $\mathsf{S4}_n^{\mathsf{J}}$. So when the system perform case analysis, we have to help it with such implications.

(B) A short version:

If we make second,third and fourth premises stronger, We then obtain a shorter version:

Hypotheses:

1. $c_1 \wedge \neg c_2 \wedge c_3$
2. $K_{3,2}\ c_1$
3. $K_{3,2}\ c_3$
4. $K_{3,2}\ (K_{2,1} c_1)$
5. $K_{3,2}\ ((c_1 \wedge c_2 \wedge c_3) \rightarrow \neg \mathtt{kw}_{2,1}\ c_1)$

Conclusion: $\mathtt{kw}_{3,2}\ c_2$.

(C) A short version with J:

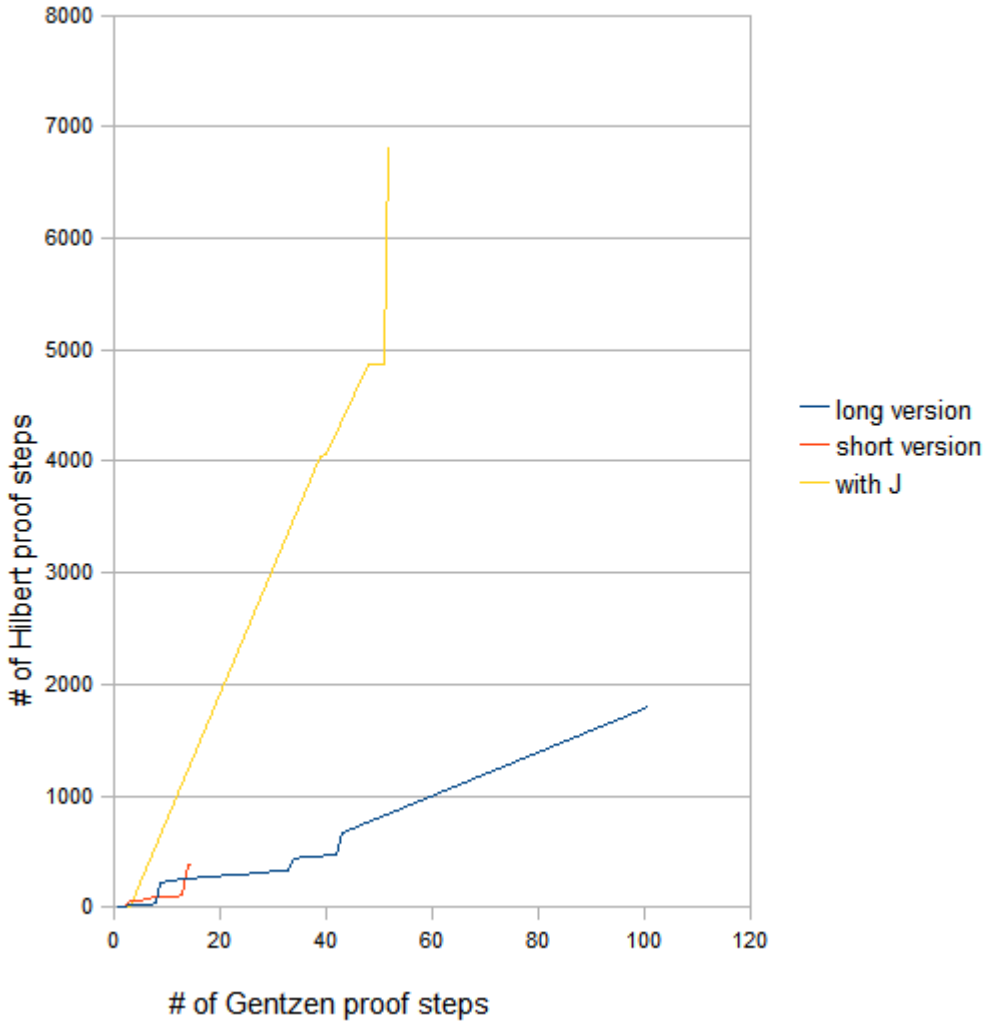
We would like to see some modalities realized as proof terms, so we replaced all modalities that represent facts that everyone knows (due to public announcements or general conditions of the puzzle) with J. We also strengthen the conclusion to state that at the third moment, everyone knows that everyone knows about themselves, so we have to add lines (6), (7) to bring the knowledge of the first and third children from step 2 to step 3, and we add (9) to reflect the fact that if the second child learns about himself, he will announce it. We did not perform this transformation with the longer version because the prover was overwhelmed by the complexity.

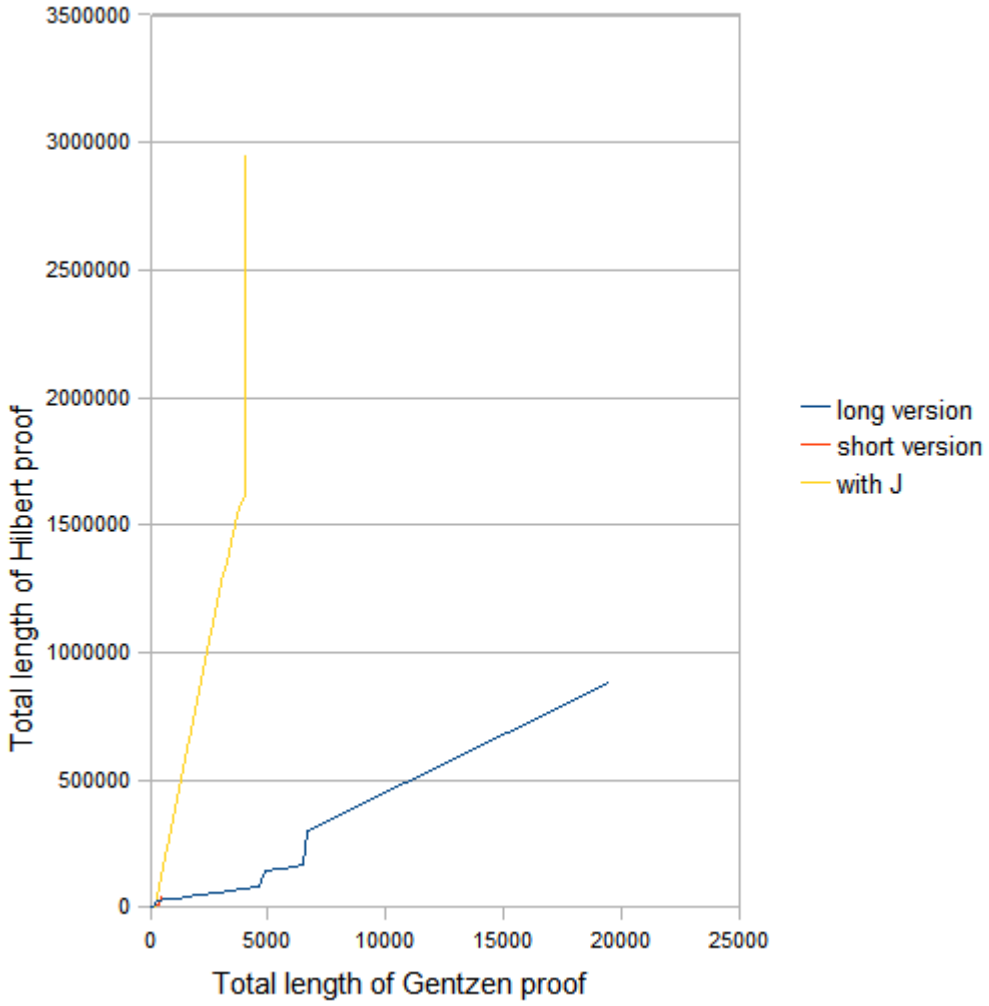
Hypotheses:

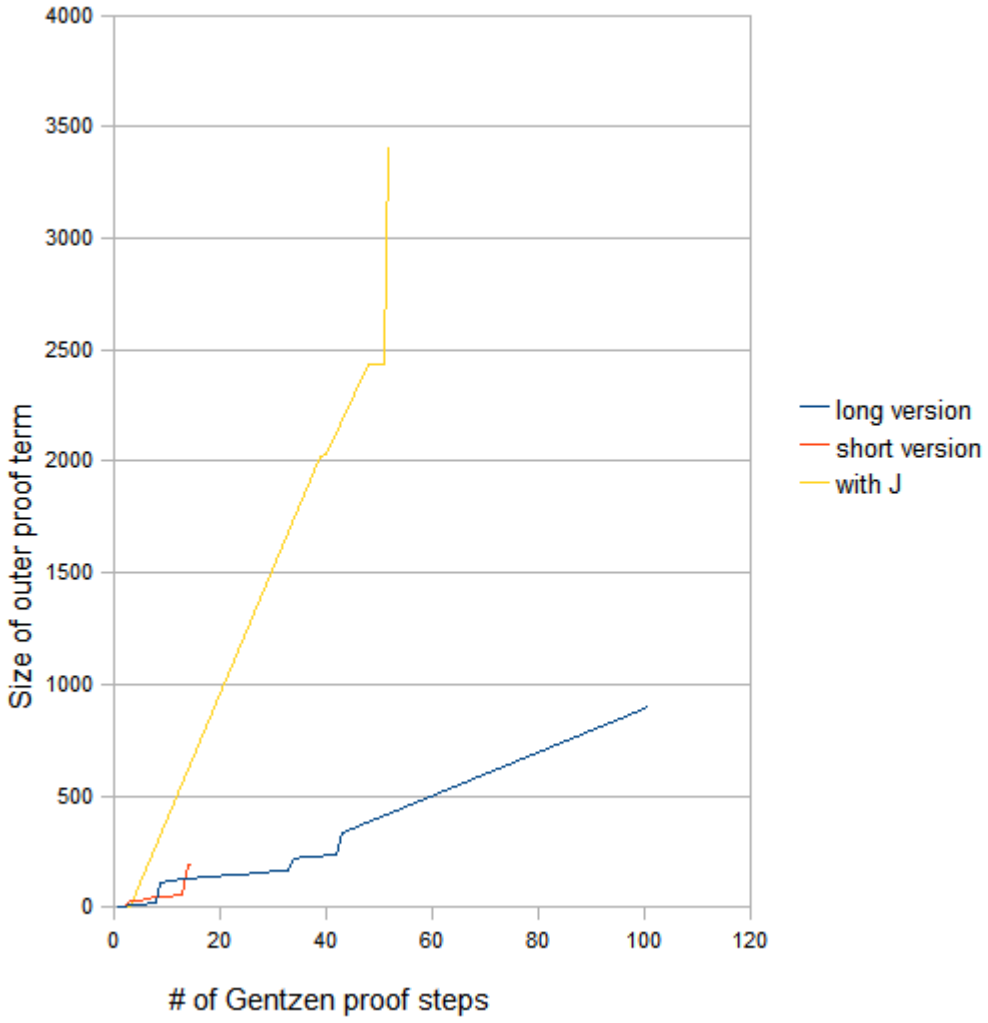
1. $c_1 \wedge \neg c_2 \wedge c_3$
2. $J(\mathtt{kw}_{3,2}\ c_1)$
3. $J(\mathtt{kw}_{3,2}\ c_3)$
4. $J(K_{2,1}\ c_1)$
5. $J(K_{2,3}\ c_3)$
6. $J(K_{2,1}c_1 \rightarrow K_{3,1}c_1)$
7. $J(K_{2,3}c_3 \rightarrow K_{3,3}c_3)$
8. $J((c_1 \wedge c_2 \wedge c_3) \rightarrow \neg(\mathtt{kw}_{2,1}\ c_1))$
9. $J(K_{3,2}c_2 \rightarrow J(K_{3,2}\ c_2))$

Conclusion: $J(K_{3,2}\neg c_2 \wedge K_{3,1}c_1 \wedge K_{3,3}c_3)$.

Here are the graphs for all three cases:


8000
7000
6000
5000
4000
3000
2000
1000
0
# of Hilbert proof steps
0
20
40
60
80
100
120
# of Gentzen proof steps
long version
short version
with J


3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Total length of Hilbert proof
0
5000
10000
15000
20000
25000
Total length of Gentzen proof
long version
short version
with J



2.8 Conclusion

Following [Brezhnev and Kuznets, 2006] and [Artemov, 2004], a polynomial realization algorithm for $\mathsf{S4}_n\mathsf{LP}$ was implemented in MetaPRL Logical Framework and connected with $\mathsf{S4}^{\mathsf{J}}_n$ prover [Bryukhov, 2006]. This procedure was

run on several interesting examples presented in this text. Performance-wise, the bottleneck was always on $\mathsf{S4}^{\mathsf{J}}_n$ prover's side. On the other hand, even small $\mathsf{S4}^{\mathsf{J}}_n$ Gentzen-style proofs result in long $\mathsf{S4}_n\mathsf{LP}$ Hilbert-style proofs which are beyond human comprehension. Though we made certain efforts to mitigate this issue, the most usable results of each run of the algorithm are the lengths of the proof and outer term, and realized formula itself.

We used Ocsigen Web-server (`http://ocsigen.org`) to make this work available as a simple Web application at `http://yegor.org/s4nlp/theorem` .

2.9 Acknowledgements

The author is grateful to her scientific advisor Distinguished Professor Sergei Artemov for his wise supervision. She is also very thankful to Yegor Bruykhov for support and valuable insights. She is indebted to Karen Kletter for editing the linguistic aspects of the article.

Appendix A

This is the description of the **KT5** logical system given by John McCarthy in his *On the Model Theory of Knowledge* paper [McCarthy *et al.*, 1978], where **KT5** is **K5** with time.

K-modality now has two indices: time and agent, i.e. $K_{t_1,a_1}A$ stands for 'At time t_1 agent a_1 knows A.' And J-modality has one index: time, i.e. J_t means that at time t everybody knows ....

Axioms:

M_1. $\neg\neg A \rightarrow A$

M_2. $A \rightarrow (B \rightarrow A)$

M_3. $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

M_4. $K_{t,a}A \rightarrow A$

M_5. $J_tA \rightarrow J_tK_{t,a}A$

M_6. $(K_{t_1,a}A \rightarrow B) \rightarrow (K_{t_2,a}A \rightarrow K_{t_2,a}B)$, where $t_1 \leq t_2$

M_7. $\neg K_{t,a} \rightarrow K_{t,a}\neg K_{t,a}A$

Inference rules:

R_1. $(A \to B), A \vdash B$ (modus ponens)

R_4. $A \vdash K_{t,a}A$ ($K_{t,a}$ necessitation)

Definition. K is a *knowledge set* for $K_{t,a}$ if K satisfies the following conditions:

(KS_1) K is consistent.

(KS_2) $K = K_{t,a}\overline{K}$, where $\overline{K} = \{A|K \vdash A\}$, i.e., closed with respect to all modalities.

(KS_3) if $K \vdash K_{t,a}A_1 \vee \ldots \vee K_{t,a}A_n$, then $K \vdash A_i$ for some i $(1 \leq i \leq n)$.

Definition. B is a *knowledge base* for $K_{t,a}$ if B satisfies the following conditions:

(KB_1) B is consistent;

(KB_2) $B \subset K_{t,a}\overline{B}$, where $\overline{B} = \{A|B \vdash A\}$;

(KB_3) if $B \vdash K_{t,a}A_1 \vee \ldots \vee K_{t,a}A_n$, then $B \vdash A_i$ for some i $(1 \leq i \leq n)$.

By (KS_2) and (KB_2), we see that any element in K (or B) has the form $K_{t,a}A$. It is easy to see that if B is a knowledge base for $K_{t,a}$, then $K_{t,a}\overline{B}$ is a knowledge set for $K_{t,a}$

Lemma 3 *The following conditions are equivalent:*

1. *If* $\Gamma \nvdash A$ *then* $\Gamma \vdash \neg K_{t,a}A$;
2. *If* $\Gamma \vdash K_{t,a}A_1 \vee \ldots \vee K_{t,a}A_n$ *then* $\Gamma \vdash A_i$ *for some* i $(1 \leq i \leq n)$;
3. *If* $\Gamma \vdash K_{t,a}A$, *then* $\Gamma \vdash A$ *or* $\Gamma \vdash \neg A$.

The puzzle of unfaithful wives is usually stated as follows:

There was a country which was inhabited by one million married couples. Of these one million wives, 40 were unfaithful. The situation was that each husband knew whether the other men's wives were unfaithful but not if his own wife was unfaithful. One day (call it the first day), the king of the country issued the following decree:

(A) There is at least one unfaithful wife.

(B) Each husband knows whether or not other men's wives are unfaithful.

(C) Every night (from tonight), each man must perform his deduction, based on his current knowledge, and try to prove whether his wife is unfaithful or not.

(D) Each man who has succeeded in proving that his wife is unfaithful must chop off her head the next morning.

(E) Every morning, each man must determine whether or not somebody chopped off his wife's head.

(F) Each man's knowledge prior to this decree's issuance consists of only the knowledge about other men's wives' unfaithfulness.

The problem is, 'what will happen under this situation?' The answer is that on the 41^{st} day, 40 unfaithful wives will have been beheaded.

Let's assume that there are k ($k \geq 1$) married couples in the country. Let a_i denote the i^{th} husband, p_i stand for 'a_i's wife is unfaithful', and $n \in \mathbb{N}^+$ denote the day.

Let $\pi_{\epsilon_1,\ldots,\epsilon_k} = \bigwedge_{i=1}^{k} p_i^{\epsilon_i}$, where $\epsilon_i \in \{0,1\}$ and $p_i^1(p_i^0)$ denotes $p_i(\neg p_i)$.

We put $\Pi = Image(\pi)$ and $\Pi_o = \Pi - \{\bigwedge_{i=1}^{k} p_i^0\}$, i.e., Π_0 excludes the case in which all wives are faithful.

Now, let Γ denote what the King publicized on the first day, and $B_\pi(K_{n,a_i})$ $(i = 1,\ldots,k)$ denote a knowledge base for K_{n,a_i} under the situation $\pi = \pi_{\epsilon_1,\ldots,\epsilon_k} \in \Pi_0$.

Let

$$\lceil B_\pi(K_{n,a_i}) \vdash A \rceil = \begin{cases} \top & \text{if } B_\pi(K_{n,a_i}) \vdash A \\ \bot & \text{otherwise} \end{cases}$$

$$\lceil B_\pi(K_{n,a_i}) \nvdash A \rceil = \begin{cases} \top & \text{if } B_\pi(K_{n,a_i}) \nvdash A \\ \bot & \text{otherwise} \quad . \end{cases}$$

Let's put $(k) = \{1,\ldots,k\}$. Then, as formalization of the puzzle, the following identities are postulated:

$$B_\pi(K_{n,a_i}) = K_{1,a_i}\Gamma \bigcup \{K_{1,a_i}\ p_j^{\epsilon_j} \mid j \neq i, j \in (1,\ldots,k)\}, \text{ where } \pi = \pi_{\epsilon_1,\ldots,\epsilon_k} \quad Eq(\pi,i,1)$$

states that knowledge base for agent a_i on the n^{th} day under the situation π is following: on the first day every husaband knows what the King publicized, and each husband knows whether other men's wives are unfaithful or not.

$$B_\pi(K_{n+1,a_i}) = K_{n+1,a_i}B_\pi(K_{n,a_i})$$
$$\bigcup \{[K_{n+1,a_i}][K_{n,a_j}]\ p_j \mid B_\pi(K_{n,a_j}) \vdash a_j, j \in (k)\}$$
$$\bigcup \{[K_{n+1,a_i}]\neg[K_{n,a_j}]\ p_j \mid B_\pi(K_{n,a_j}) \nvdash a_j, j \in (k)\} \quad Eq(\pi, i, n+1)$$

states that the knowledge base for agent a_i on the $n+1^{th}$ day under the situation π is following: on the $(n+1)^{st}$ day each husband has the knowledge he acquired on the previous day, and each hasband sees whether somebody chopped off his wife's head.

$$\Gamma = \{J_1 \bigvee_{i=1}^{k} p_i\} \bigcup \{J_1 K_{1,a_i}\ p_j \mid j \neq i, i \in (k), j \in (k)\}$$
$$\bigcup \{J_1(\pi \rightarrow (\lceil B_\pi(K_{n,a_i}) \vdash p_i \rceil \rightarrow J_{n+1}K_{n,a_i}\ p_i)) \mid \pi \in \Pi_0, i \in (k), n \in \mathbb{N}^+\}$$
$$\bigcup \{J_1(\pi \rightarrow (\lceil B_\pi(K_{n,a_i}) \nvdash p_i \rceil \rightarrow J_{n+1}\neg K_{n,a_i}\ p_i)) \mid \pi \in \Pi_0, i \in (k), n \in \mathbb{N}^+\}$$
$$\bigcup \{J_1(\lceil B_\pi(K_{n,a_i}) \vdash A \rceil \rightarrow J_{n+1}(\pi \rightarrow K_{n,a_i}\ A)) \mid \pi \in \Pi_0, i \in (k)\} \qquad Eq(*)$$

States that on the first day the King announced the following:

On the first day it is common knowledge that at least one wife is unfaithful, and

it is common knowledge that on the first day, each husband knows whether other men's wives are unfaithful or not, and

it is common knowledge that under situation π, if on the n^{th} day a husband derives that his wife was unfaithful, then on the next day it becomes common knowledge that he knows it, and

it is common knowledge that under situation π, if on the n^{th} day a hus-

band does not derive that his wife was unfaithful, then on the next day it becomes common knowledge that he does not know it, and

it is common knowledge that if a husband derives anything, say A, then on the next day it becomes common knowledge that given the situation π, he can derive it.

Since meta-notions such as knowledge base and provability ($\vdash$) cannot be expressed directly in our language, we were forced to interpret the King's decree into Γ in a somewhat indirect fashion.

Now, if we read $Eq(*)$ as the definition of Γ, we find that the definition is circular since in order that Γ may be definable by $Eq(*)$, it is necessary that $B_\pi(K_{n,a_i})$ are already defined, whereas $B_\pi(K_{n,a_i})$ are defined in terms of Γ in $Eqs(\pi, i, n)$. So, we will treat these equations as a system

$$\Sigma = \{Eq(\pi, i, n) \mid \pi \in \Pi_0, i \in (k), n \in \mathbb{N}^+\} \bigcup \{Eq(*)\}$$

of equations with the unknowns $\{B_\pi(K_{n,a_i}) \mid \pi \in \Pi_0, i \in (k), n \in \mathbb{N}^+\}$ and Γ. We will solve Σ under the following conditions:

- For any $\pi \in \Pi_0$, $\Gamma \cup \{\pi\}$ is consistent.
- For any $\pi \in \Pi_0$ and for i^th agent on n^{th} day, $B_\pi(K_{n,a_i})$ is his knowledge base.

These conditions are natural in view of the intended meanings of Γ and $B_\pi(K_{n,a_i})$.

Appendix B

Here is the list of 20 axioms:

$C_1: \ A \to (B \to A)$

$C_2: \ (A \to (B \to C)) \to ((A \to B) \to (A \to C))$

$C_3: \ A \wedge B \to A$

$C_4: \ A \wedge B \to B$

$C_5: \ A \to (B \to (A \wedge B))$

$C_6: \ A \to (A \vee B)$

$C_7: \ B \to (A \vee B)$

$C_8: \ (A \to C) \to ((B \to C) \to (A \vee B \to C))$

$C_9: \ (A \to C) \to ((A \to \neg B) \to \neg A)$

$C_{10}: \ \neg\neg A \to A$

$C_{11}: \ A \vee \neg A$

$C_{12}: \ s : (A \to B) \to (t : A \to (s \cdot t) : B)$

$C_{13}: \ t : A \to A$

$C_{14}: \ t : A \to !t : t : A$

$C_{15}:\ t : A \to (s + t) : A$

$C_{16}:\ s : A \to (s + t) : A$

$C_{17}:\ K_i(A \to B) \to (K_iA \to K_iB)$

$C_{18}:\ K_iA \to A$

$C_{19}:\ K_iA \to K_iK_iA$

$C_{20}:\ t : A \to K_iA$

Bibliography

[Altenkirch, 1993] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, Department of Computer Science, The King's Blgs, Mayfield Rd, Edinburgh EH9 3JZ, November 1993.

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

[Antonakos, 2007] Evangelia Antonakos. Justified and common knowledge: Limited conservativity. In Sergei N. Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2007, New York, NY, USA, June 4-7, 2007, Proceedings*, volume 4514 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2007.

[Artemov and Nogina, 2005] Sergei Artemov and Elena Nogina. Introducing justification into epistemic logic. *J. Log. and Comput.*, 15(6):1059–1073, 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, 2006] Sergei Artemov. Justified common knowledge. *Theoretical Computer Science*, 357(1):4–22, 2006.

[Artemov, 2007] Sergei Artemov. Justification logic. Technical Report TR-2007019, CUNY Ph.D. Program in Computer Science, 2007.

[Artemov, 2008] Sergei Artemov. Symmetric logic of proofs. 4800:5871, 2008.

[Barendregt, 1992a] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992.

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

[Barras and Werner, 1997] Bruno Barras and Benjamin Werner. Coq in coq. March 1997.

[Barras, 1999] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[Barthe *et al.*, 2003] G. Barthe, M. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions, 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.

[Brezhnev and Kuznets, 2006] Vladimir Brezhnev and Roman Kuznets. Making knowledge explicit: how hard it is. *Theoretical Computer Science*, 357(1):23–34, 2006.

[Brezhnev, 2000] Vladimir N. Brezhnev. On explicit counterparts of modal logics. Technical Report CFIS 2000–05, Cornell University, 2000.

[Bryukhov, 2005] Yegor Bryukhov. Automatic proof search in logic of justified common knowledge. In Holger Schlingloff, editor, *Proceedings of Methods for Modalities Workshop 2005*. Humboldt University, 2005.

[Bryukhov, 2006] Yegor Bryukhov. *Integration of Decision Procedures into High-Order Interactive Provers*. PhD thesis, The Graduate School and University Center, CUNY, 2006.

[Cook and Reckhow, 1974] Stephen Cook and Robert Reckhow. On the lengths of proofs in the propositional calculus (preliminary version). In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 135–148, New York, NY, USA, 1974. ACM.

[Coq, a] Coq proof assistant web site. `http://pauillac.inria.fr/coq/`.

[Coq, b] Coq reference manual. `http://coq.inria.fr/doc/main.html`.

[Coq, c] Coq tutorial. `http://coq.inria.fr/doc/tutorial.html`.

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

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

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

[Fitting, 2006] Melvin Fitting. Modal logic notes, January 2006. First Indian Winter School on Logic and Its Relationship with Other Dsciplines, IIT Bombay.

[Fitting, 2007a] Melvin Fitting. Modal proof theory. In P. Blackburn, J. F. K. van Benthem, and F. Walter, editors, *Handbook of modal logic*, pages 85–138. Elsevier, 2007.

[Fitting, 2007b] Melvin Fitting. Realizations and LP. March 2007. unpublished.

[Fitting, 2007c] Melvin Fitting. Realizing substitution instances of modal theorems. Technical Report TR-2007006, CUNY Ph.D. Program in Computer Science, March 2007.

[Fitting, 2007d] Melvin Fitting. S4LP and local realizability. Technical Report TR-2007020, CUNY Ph.D. Program in Computer Science, November 2007.

[Giménez, 1995] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *TYPES '94: Selected papers from the International Workshop on Types for Proofs and Programs*, pages 39–59, London, UK, 1995. Springer-Verlag.

[Giménez, 1998] Eduardo Giménez. Structural recursive definitions in type theory. In *Automata, Languages and Programming*, pages 397–408, 1998.

[Girard, 1986] J-Y. Girard. The system F of variable types: Fifteen years later. *Journal of Theoretical Computer Science*, 45:159–192, 1986.

[Girard, 1987] J-Y. Girard. *Proof Theory and Logical Complexity*, volume 1. Bibliopolis, Napoli, 1987.

[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.*, 2003a] Jason Hickey, Nathaniel Gray, Aleksey Nogin, Cristian Tapus, and Xin Yu. Introduction to MetaPRL theorem prover. Tutorial given at TPHOLs 2003, Rome, Italy, September 2003.

[Hickey *et al.*, 2003b] Jason Hickey, Aleksey Nogin, Brian Aydemir, et al. Introduction to MetaPRL theorem prover. Tutorial given at TPHOLs 2003, Rome, Italy, September 2003.

[Hickey *et al.*, 2003c] 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 David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

[Hickey, 2001] Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[Hickey, 2003] Jason Hickey. MetaPRL – a modular logical environment. Talk given at TPHOLs 2003, Rome, Italy, September 2003.

[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.

[Krupski, 2006a] Nikolai V. Krupski. On the complexity of the reflected logic of proofs. *Theoretical Computer Science*, 357(1–3):136–142, July 2006.

[Krupski, 2006b] Vladimir N. Krupski. Referential logic of proofs. *Theoretical Computer Science*, 357(1–3):143–166, July 2006.

[Kuznets, 2008] Roman Kuznets. *Complexity Issues in Justificaion Logic*. PhD thesis, The Graduate School and University Center, CUNY, 2008.

[Luo and Pollack, 1992] Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, 1992.

[Luo, 1990] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, 1990.

[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.

[Martin-Löf, 1984] Per Martin-Löf. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory, Lecture Notes. Bibliopolis, Napoli, 1984.

[McCarthy *et al.*, 1978] John McCarthy, Masahiko Sato, Takeshi Hayashi, and Shigeru Igarashi. On the model theory of knowledge. Technical report, Stanford, CA, USA, 1978.

[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.

[Met,] MetaPRL proof assistant web site. `http://metaprl.org/default.html`.

[Meyer and Hoek, 1995] John-Jules Ch Meyer and Wiebe Van Der Hoek. *Epistemic Logic for AI and Computer Science*. Cambridge University Press, New York, NY, USA, 1995.

[Milnikel, 2007] Robert [S.] Milnikel. Derivability in certain subsystems of the Logic of Proofs is Π_2^p-complete. *Annals of Pure and Applied Logic*, 145(3):223–239, March 2007.

[Nordström *et al.*, 2000] B. Nordström, K. Petersson, and J. M. Smith. Martin-Löf's type theory, 2000.

[Novak and Bryukhov, 2004] Natalia Novak and Yegor Bryukhov. Implementing the calculus of inductive constructions in the MetaPRL framework. In Konrad Slind, editor, *Emerging Trends. Prooceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2004)*. University of Utah, 2004.

[Otten and Kreitz, 1996a] Jens Otten and Christoph Kreitz. T-string-unification: Unifying prefixes in non-classical proof methods. In U. Moscato, editor, *5th 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.

[Paulin-Mohring, 1993] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. F. Groote, editors, *Proceedings 1st Int. Conf. on Typed Lambda Calculi and Applications, TLCA'93, Utrecht, The Netherlands, 16–18 March 1993*, volume 664, pages 328–345. Springer-Verlag, Berlin, 1993.

[Paulson, 1998] Lawrence C. Paulson. A fixedpoint approach to (co)inductive and (co)datatype definitions, 1998.

[Pollack, 1995] Robert Pollack. *The Theory of LEGO:A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, April 1995.

[Renne, 2008] Bryan Renne. *Dynamic Epistemic Logic with Justification*. PhD thesis, CUNY Graduate Center, May 2008.

[Rubtsova, 2006] Natalia [M.] Rubtsova. Evidence reconstruction of epistemic modal logic S5. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science — Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, June 8–12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 313–321. Springer, 2006.

[Schmidt, 2009] Renate Schmidt. A list of computational tools useful for modal logics, and related logics, 2009. `http://www.cs.man.ac.uk/~schmidt/tools/`.

[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.

[Seldin, 2001] Jonathan P. Seldin. Type theories from Barendregt's cube for theorem provers, 2001.

[Troelstra and Schwichtenberg, 2000] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory (2nd ed.).* Cambridge University Press, New York, NY, USA, 2000.

[von Wright, 1951] Georg Henrik von Wright. An essay in modal logic, 1951.

[Wallen, 1987] Lincoln A. Wallen. *Automated proof search in non-classical logics: efficient matrix proof methods for modal and intuitionistic logics*. PhD thesis, Edinburgh University, 1987.

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

[Yasugi and Oda, 2002] Mariko Yasugi and Sobei H. Oda. symposium articles : A note on the wise girls puzzle. *Economic Theory*, 19(1):145–156, 2002.

[Yavorskaya (Sidon), 2006] Tatiana Yavorskaya (Sidon). Multi-agent explicit knowledge. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *Computer Science — Theory and Applications, First International Computer Science Symposium in Russia, CSR 2006, St. Petersburg, Russia, June 8–12, 2006, Proceedings*, volume 3967 of *Lecture Notes in Computer Science*, pages 369–380. Springer, 2006. Later journal version published as [Yavorskaya (Sidon), 2008].

[Yavorskaya (Sidon), 2008] Tatiana Yavorskaya (Sidon). Interacting explicit evidence systems. *Theory of Computing Systems*, 43(2):272–293, August 2008. Published online in October 2007.