Teaching a Formalized Logical Calculus

Asta Halkjær From Alexander Birch Jensen Anders Schlichtkrull

Jørgen Villadsen *

DTU Compute - Department of Applied Mathematics and Computer Science,
Technical University of Denmark, Richard Petersens Plads, Building 324, DK-2800 Kongens Lyngby, Denmark

Classical first-order logic is in many ways central to work in mathematics, linguistics, computer science and artificial intelligence, so it is worthwhile to define it in full detail. We present soundness and completeness proofs of a sequent calculus for first-order logic, formalized in the interactive proof assistant Isabelle/HOL. Our formalization is based on work by Stefan Berghofer, which we have since updated to use Isabelle's declarative proof style Isar (Archive of Formal Proofs, Entry FOL-Fitting, August 2007 / July 2018). We represent variables with de Bruijn indices; this makes substitution under quantifiers less intuitive for a human reader. However, the nature of natural numbers yields an elegant solution when compared to implementations of substitution using variables represented by strings. The sequent calculus considered has the special property of an always empty antecedent and a list of formulas in the succedent. We obtain the proofs of soundness and completeness for the sequent calculus as a derived result of the inverse duality of its tableau counterpart. We strive to not only present the results of the proofs of soundness and completeness, but also to provide a deep dive into a programming-like approach to the formalization of first-order logic syntax, semantics and the sequent calculus. We use the formalization in a bachelor course on logic for computer science and discuss our experiences.

1 Introduction

Classical first-order logic is often used in mathematics, linguistics, philosophy and computer science. It is worthwhile to define it formally and recent advances in proof assistants like Isabelle/HOL have made it feasible [2].

The sequent calculus considered has the special property of an always empty antecedent and a list of formulas in the succedent. We obtain the proofs of soundness and completeness for the sequent calculus as a derived result of the inverse duality of its tableau counterpart.

We strive to not only present the results of the proofs of soundness and completeness, but also to provide a deep dive into a programming-like approach to the formalization of first-order logic syntax, semantics and the sequent calculus. This is advantageous in a bachelor course, in particular when the students are familiar with functional programming.

We have taken the formalization of natural deduction by Berghofer [1] as the starting point, but we have updated the soundness and completeness proofs using Isabelle's declarative proofs style Isar [25]. The soundness and completeness proofs for tableaux and the sequent calculus are new for this paper.

We represent variables with de Bruijn indices; this makes substitution under quantifiers less intuitive for a human reader. However, the nature of natural numbers yields an elegant solution when compared to implementations of substitution using variables represented by strings.

*Corresponding author: jovi@dtu.dk

P. Quaresma, W. Neuper and J. Marcos (Eds.): Theorem proving components for Educational software 2019 (ThEdu'19)
EPTCS 313, 2020, pp. 73–92, doi:10.4204/EPTCS.313.5

The development described in this paper is available online:

https://bitbucket.org/isafol/isafol/src/master/FOL_Berghofer/

```
4253 lines FOL_Berghofer.thy
 867 lines FOL_Tableau.thy     A contribution of this paper
 217 lines FOL_Sequent.thy     A contribution of this paper
 132 lines FOL_Appendix.thy    A contribution of this paper
```

These numbers include blank lines and a few comments. All in all it takes around 5 seconds in real time to verify on a fairly standard computer. The entire formalization is based on the standard theory `Main` (the standard library of Isabelle/HOL which comes with many useful functions and facts about e.g. natural numbers and lists).

We have recently named our system based on the formalization in Isabelle/HOL:

SeCa✓ Sequent Calculus Verifier

For our bachelor course we focus on the fragment (499 lines) of the formalization available here:

https://bitbucket.org/isafol/isafol/src/master/Sequent_Calculus/

Here only the soundness proof is included but in addition a small and a large proof in the sequent calculus is formalized.

The structure of the paper is as follows. Section 2 explains the formalization of the syntax and the semantics of classical first-order logic. Section 3 describes the sequent calculus. Section 4 outlines the formalized soundness and completeness proofs. Section 5 considers teaching the sequent calculus. Section 6 is a discussion of related work. Finally, section 7 is the conclusion.

2 Formalization of Syntax and Semantics

We provide in this section definitions for the syntax and semantics of first-order classical logic without equality. This part of the formalization is based on the work by Berghofer [1].

Throughout the presentation of our formalization, we will state and explain the types of definitions and functions. In particular, the use of type variables requires further explanation. In Isabelle a type variable states an arbitrary type. We will consistently use the type variables `'a`, `'b` and `'c` for our model. The type `'a` specifies the type of function identifiers, e.g. strings or natural numbers. The type `'b` similarly specifies the type of predicate identifiers. Finally, the type `'c` specifies the type of elements in the universe.

2.1 Syntax of first-order logic formulas

This section provides the necessary definitions to construct well-formed formulas of classical first-order logic without equality. We start by defining the syntax for terms of first-order logic:

```
term ::=

    Var nat
    Fun 'a [term, ..., term]
```

The terms are variables and functions as usual. The type `nat` is for natural numbers as we use de Bruijn indices and thus variables appear as natural numbers. Constants are represented as functions with no arguments.

We now turn to define the syntax for formulas of first-order logic:

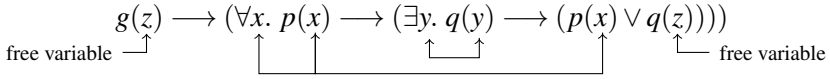
```
form ::=

  ⊥
  ⊤
  Pre 'b ['a term, ..., 'a term]
  Con form form
  Dis form form
  Imp form form
  Neg form
  Uni form
  Exi form
```

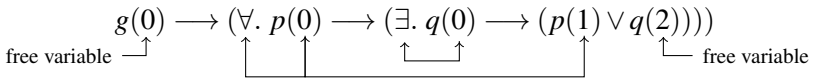
Formulas have the usual first-order logic connectives and also include $\bot$, $\top$ and $\neg$. The constructor `Pre` is for predicates, `Con` is for $\wedge$, `Dis` is for $\vee$, `Imp` is for $\longrightarrow$, `Neg` is for $\neg$, `Uni` is for the universal quantifier $\forall$, and `Exi` is for the existential quantifier $\exists$.

It must be emphasized that all functions and predicates have arbitrarily many arguments.

The use of de Bruijn indices in formulas is most easily explained by an example:



The arrows indicate the references of variables with regard to quantifiers and free variables. Consider below the same formula using de Bruijn indices instead:



Syntactic representations of quantifiers such as $\exists x$ are replaced by $\exists$ as the variable referencing is implicit. A variable 0 is bound by the innermost quantifier, a variable 1 inside two quantifiers is bound by the outermost quantifier, and so on. A variable references a free variable when its index exceeds or equals the number of quantifiers it is bound by. Outside the scope of any quantifier, 0 references a free variable, 1 another free variable, and so on.

We define the syntax of Herbrand terms that are closed by construction (ground terms).

```
hterm ::= HFun 'a [hterm, ..., hterm]
```

Defining the Herbrand terms as a separate type plays well with our use of a type variable to represent the universe, which for the completeness theorem consists of Herbrand terms. Herbrand terms are distinguished from regular terms by having no variables. Alternatively we could introduce and reason about an explicit sub-type of the regular terms.

2.2 Semantics of first-order logic

We present here a formalization of the semantics of first-order classical logic. The semantics describe the truth evaluation of formulas within a given model i.e. for a given universe, environment and interpretation of function and predicate identifiers.

In the context of programming in Isabelle, it is particularly important to understand the components of a model. The environment e maps variables to elements of the universe. The type of elements in the universe is arbitrary and the variables are natural numbers.

The interpretation f maps function identifiers to functions on the universe and g maps predicate identifiers to predicates. Combined they form our model. Given an instance of such a model we can evaluate formulas.

The semantics of first-order formulas is defined below as the function `semantics` along with functions `semantics_term` and `semantics_list` for the semantics of terms and lists of terms, respectively:

```
semantics_term :: (nat ⇒ 'c) ⇒ ('a ⇒ 'c list ⇒ 'c) ⇒ term ⇒ 'c

  semantics_term e f (Var n) = e n
  semantics_term e f (Fun i l) = f i (semantics_list e f l)

semantics_list :: (nat ⇒ 'c) ⇒ ('a ⇒ 'c list ⇒ 'c) ⇒
    term list ⇒ 'c list

  semantics_list e f [] = []
  semantics_list e f (t # l) =
    semantics_term e f t # semantics_list e f l
```

Let us first inspect the types of our environment `e` and interpretation `f`. The environment has the type `(nat ⇒ 'c)` as we map variables (natural numbers) to elements of the universe, i.e. elements of type `'c`. The interpretation of function identifiers `f` has the type `('a ⇒ 'c list ⇒ 'c)`. We map function identifiers of type `'a` along with its arguments to elements of the universe. Recall that arguments to functions are terms which themselves are mapped to elements of the universe by interpretation `f`. Therefore, we need not only the function identifier, but also the interpretation of its arguments. For lists of terms `semantics_list`, we simply map each element to its semantic value recursively.

```
semantics :: (nat ⇒ 'c) ⇒ ('a ⇒ 'c list ⇒ 'c) ⇒
    ('b ⇒ 'c list ⇒ bool) ⇒ form ⇒ bool

  semantics e f g ⊥ = False
  semantics e f g ⊤ = True
  semantics e f g (Pre i l) = g i (semantics_list e f l)
  semantics e f g (Con p q) = (semantics e f g p ∧ semantics e f g q)
  semantics e f g (Dis p q) = (semantics e f g p ∨ semantics e f g q)
  semantics e f g (Imp p q) = (semantics e f g p ⟶ semantics e f g q)
  semantics e f g (Neg p) = (¬ semantics e f g p)
  semantics e f g (Uni p) = (∀z. semantics (shift e 0 z) f g p)
  semantics e f g (Exi p) = (∃z. semantics (shift e 0 z) f g p)
```

For the semantics of formulas we further have as an argument the predicate interpretation `g` of type `('b ⇒ 'c list ⇒ bool)`. Similarly to the function interpretation we map predicate identifiers of type `'b` along with its interpreted term arguments of type `'c list` to Boolean values. The cases for most of the logical connectives are rather trivial. The explicit use of logical connectives on the right hand side of our function definition may at first seem rather confusing. However, these connectives should be understood within the context of Isabelle programming, namely as built-in functions that return the Boolean value of the operator applied to its arguments.

The cases of the existential and universal quantifiers are more challenging. Consider the quantifiers on the right-hand side as programmable functions that work on a set of possible values of `z`. By inspection in Isabelle, and later through understanding the function `shift`, we realize that the type of `z` is `'c`. This means that the existential quantifier checks if there is an element in the universe that makes the formula

p true. Similarly, the universal quantifier checks if all elements make p true. Exactly how we put in z at the places of the quantified variables is handled by shift, which is defined below:

```
shift :: (nat ⇒ 'c) ⇒ nat ⇒ 'c ⇒ (nat ⇒ 'c)

  shift e v z ≡
    (λn. if n < v then e n else if n = v then z else e (n - 1))
```

We can consider shift a function that takes as arguments an existing environment e of type (nat ⇒ 'c), a variable v of type nat and an element of the universe z of type 'c, and the result is an updated environment. In the semantics, shift is always used with v having the value 0. This essentially boils down the line we have to understand to:

```
  shift e 0 z ≡
    (λn. if n < 0 then e n else if n = 0 then z else e (n - 1))
```

Since n is a natural number it further reduces to:

```
  shift e 0 z ≡
    (λn. if n = 0 then z else e (n - 1))
```

Recall how de Bruijn indices are used under quantifiers. When checking if the quantifier holds for a given z, we swap in z for all variables with index 0. In the process of evaluating we essentially eliminate the quantifier, as we now just consider one specific element z of the universe, and hence indices above 0 are subtracted by 1 to adjust variables that were previously within the scope of a quantifier.

The complicated definition of shift is preferred instead of the simpler one since it matches the pattern used in the definition of substitution to be explained later.

3 The Proof System

In this section we go over an informal definition of the sequent calculus. We present the proof system in its entirety in a format one would expect from a textbook. Simultaneously, we present in slightly abbreviated Isabelle syntax the rules of the proof system, and we explain the rules in detail.

For general sequent calculi the most common way of writing a proof state is

$$\Gamma \vdash \Delta$$

where $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$ and $\Delta = \{\delta_1, \ldots, \delta_n\}$ are sets of formulas. The operator $\vdash$ is to be interpreted as an implication with the conjunction of formulas in Γ on the left-hand side and the disjunction of the formulas in Δ on the right-hand side:

$$(\gamma_1 \wedge \ldots \wedge \gamma_n) \longrightarrow (\delta_1 \vee \ldots \vee \delta_n)$$

In our case the sets of formulas Γ and Δ are lists of formulas and Γ is always empty (this restriction makes the system simpler). So instead of the symbol $\vdash$ we use the symbol $\vdash$ and lists of formulas are preferred rather than sets or multisets since lists are well-known from functional programming and always finite.

For certain rules, especially those regarding quantifiers, side conditions apply. In particular, we have conditions regarding substitution under quantifiers and newness of constants and functions. Furthermore, we require functions for list membership and extension. We go through each of the required auxiliary functions and discuss their details.

3.1 Newness of Constants and Functions

Recall that constants are nothing but functions without arguments. For the introduction of a universal quantifier $\forall$ and its negated counterpart $\neg\exists$, we have requirements of newness for the instantiated terms. For a function identifier to be new, it must not occur in any of the formulas in the succedent. Below we define functions for checking newness of a constant or a function with respect to a term and a list of terms, respectively:

```
new_term :: 'a ⇒ term ⇒ bool

  new_term c (Var n) = True
  new_term c (Fun i l) = (if i = c then False else new_list c l)

new_list :: 'a ⇒ term list ⇒ bool

  new_list c [] = True
  new_list c (t # l) = (if new_term c t then new_list c l else False)
```

Given a function identifier `c` of type `'a` and a term `t`, we can trivially determine if `c` is new by checking if `t` is in fact a function with identifier `i` and if `i = c`. For checking a list of terms, we recursively traverse the list and check if newness holds for all terms in the list.

What we call newness is often called freshness but in general we prefer the shorter word and always make everything very explicit.

Note that we here and elsewhere prefer to use if-then-else constructs instead of conjunctions in order to avoid any confusions with the logical connectives of first-order logic.

We now define newness of constants and functions in formulas:

```
new :: 'a ⇒ form ⇒ bool

  new c ⊥ = True
  new c ⊤ = True
  new c (Pre i l) = new_list c l
  new c (Con p q) = (if new c p then new c q else False)
  new c (Dis p q) = (if new c p then new c q else False)
  new c (Imp p q) = (if new c p then new c q else False)
  new c (Neg p) = new c p
  new c (Uni p) = new c p
  new c (Exi p) = new c p
```

We also define newness of constants and functions in lists of formulas:

```
news :: 'a ⇒ form list ⇒ bool

  news c [] = True
  news c (p # x) = (if new c p then news c x else False)
```

The types of `new` and `news` are almost identical to `new_term` and `new_list`, barring the fact that they take a formula as input instead of a term. While most cases require no further explanation, observe that for the binary connectives the term has to be new in both the left-hand and right-hand side. A predicate is clearly the only case where terms appear directly, and we here use the previously defined function `new_list`. Again, we traverse the list recursively to check if newness holds for all formulas in the list.

3.2 Substitution of Variables

For any rule that introduces a quantifier, correct substitution of quantified variables is an integral part of the proof step. Before we define substitution for formulas, we first define a function that increments all variables of a term by 1.

```
inc_term :: term ⇒ term

  inc_term (Var n) = Var (n + 1)
  inc_term (Fun i l) = Fun i (inc_list l)
```

Given a term as input, the result is a new term with any de Bruijn indices incremented by one. We also define the function for a term list:

```
inc_list :: term list ⇒ term list

  inc_list [] = []
  inc_list (t # l) = inc_term t # inc_list l
```

Furthermore, we define substitution in a term as its own function:

```
sub_term :: nat ⇒ term ⇒ term ⇒ term

  sub_term v s (Var n) =
        (if n < v then Var n else if n = v then s else Var (n - 1))
  sub_term v s (Fun i l) = Fun i (sub_list v s l)
```

As indicated by the type, we have as a natural number `v`, the term we substitute as `s`, and the term in which the variable with de Bruijn index `v` is to be substituted. Due to the recursive definition, only the case where this term is a variable gives rise to a substitution. For terms alone we ignore any possible quantifiers. As such, a substitution inside a term can be seen as replacing all free variables with de Bruijn index `v`. Consequently, we decrement all indices greater than `v` while leaving indices below `v` untouched.

We perform substitution on a list of terms by traversing the list recursively:

```
sub_list :: nat ⇒ term ⇒ term list ⇒ term list

  sub_list v s [] = []
  sub_list v s (t # l) = sub_term v s t # sub_list v s l
```

We can now define substitution in a formula as a function:

```
sub :: nat ⇒ term ⇒ form ⇒ form

  sub v s ⊥ = ⊥
  sub v s ⊤ = ⊤
  sub v s (Pre i l) = Pre i (sub_list v s l)
  sub v s (Con p q) = Con (sub v s p) (sub v s q)
  sub v s (Dis p q) = Dis (sub v s p) (sub v s q)
  sub v s (Imp p q) = Imp (sub v s p) (sub v s q)
  sub v s (Neg p) = Neg (sub v s p)
  sub v s (Uni p) = Uni (sub (v + 1) (inc_term s) p)
  sub v s (Exi p) = Exi (sub (v + 1) (inc_term s) p)
```

As in `sub_term` we have as input a natural number `v`, a term `s` and a formula in which the substitution is to be performed. The cases for $\bot$ and $\top$ are trivial, and the cases for the logical connectives `Con`, `Dis`, `Imp` and `Neg` simply call `sub` recursively on their arguments. Substitution for terms is used on predicate arguments in the case for `Pre`. The two cases for the quantifiers require closer examination. We observe that they are almost identical. We reconstruct the formula using `sub` recursively in the constructor argument. In comparison to the other cases, we have `v + 1` instead of `v` and `inc_term s` instead of `s`. This is due to the fact that all de Bruijn indices are incremented when inside quantifiers. As previously described, a free variable `Var 0` will be written `Var 1` when inside a single quantifier. Similarly, we increment variables in the term that is to be substituted in when encountering a quantifier.

3.3 Extending a List of Formulas

As we will uncover later, a rule in the sequent calculus allows for an extension of the list of formulas in the succedent by any number of formulas. This is basically exchange, contraction, and weakening as standard in sequent calculus. We program this as a function `ext` using the following `member` function:

```
member :: form ⇒ form list ⇒ bool

  member p [] = False
  member p (q # x) = (if p = q then True else member p x)
```

Hence `member p x` returns `True` if the formula `p` occurs in `x`.

```
ext :: form list ⇒ form list ⇒ bool

  ext y [] = True
  ext y (p # x) = (if member p y then ext y x else False)
```

Hence `ext y x` returns `True` if every formula in `x` is a member of `y`. The extension function `ext` is simpler than a permutation function (which would be sufficient).

3.4 Inductive Definition of Sequent Calculus

We present here the inductive definition of the sequent calculus proof system. In Figure 1 all rules are presented informally in a format well-known to readers of modern logic books. We have already covered all the auxiliary functions needed for programming the side conditions of difficult rules. The inductive definition for the entire sequent calculus proof system is given below:

```
sequent_calculus (⊢) :: ('a, 'b) form list ⇒ bool

  ⊢ Pre i l # Neg (Pre i l) # x
  ⊢ Neg ⊥ # x
  ⊢ ⊤ # x
  ⊢ p # x ⟹ ⊢ Neg (Neg p) # x
  ⊢ Neg p # Neg q # x ⟹ ⊢ Neg (Con p q) # x
  ⊢ p # q # x ⟹ ⊢ Dis p q # x
  ⊢ Neg p # q # x ⟹ ⊢ Imp p q # x
  ⊢ p # x ⟹ ⊢ q # x ⟹ ⊢ Con p q # x
  ⊢ Neg p # x ⟹ ⊢ Neg q # x ⟹ ⊢ Neg (Dis p q) # x
  ⊢ p # x ⟹ ⊢ Neg q # x ⟹ ⊢ Neg (Imp p q) # x
  ⊢ sub 0 t p # x ⟹ ⊢ Exi p # x
  ⊢ Neg (sub 0 t p) # x ⟹ ⊢ Neg (Uni p) # x
  ⊢ sub 0 (Fun i []) p # x ⟹ news i (p # x) ⟹ ⊢ Uni p # x
  ⊢ Neg (sub 0 (Fun i []) p) # x ⟹ news i (p # x) ⟹ ⊢ Neg (Exi p) # x
  ⊢ x ⟹ ext y x ⟹ ⊢ y
```

We explain here only the most difficult rules. Note that the operator $\Longrightarrow$ is a meta-implication operator in Isabelle. The meta-implication operates on a higher-level than the usual logical operator $\longrightarrow$. The rules are inductively defined by use of this operator and appeal to a top-down construction of proofs, but it can be useful to consider proof construction in the reverse direction as well. Multiple uses of $\Longrightarrow$ indicate that the conclusion follows from multiple assumptions. When constructing the proof bottom-up, leaf rules mark the end of a branch in the proof tree. We have three leaf rules. Two of them are for $\top$ and $\neg\bot$. If either $\top$ or $\neg\bot$ is the first formula in the succedent, clearly its disjunction is always true. The third and possibly most interesting leaf rule is well-known from Gentzen-style systems:

```
⊢ Pre i l # Neg (Pre i l) # x
```

Leaf rules

$$\frac{}{\vdash P(v_1,\ldots,v_k),\neg P(v_1,\ldots,v_k),\Delta}$$

$$\frac{}{\vdash \neg\bot,\Delta} \qquad \frac{}{\vdash \top,\Delta}$$

α-rules

$$\frac{\vdash p,\Delta}{\vdash \neg\neg p,\Delta} \qquad \frac{\vdash \neg p,q,\Delta}{\vdash p \longrightarrow q,\Delta}$$

$$\frac{\vdash \neg p,\neg q,\Delta}{\vdash \neg(p \wedge q),\Delta} \qquad \frac{\vdash p,q,\Delta}{\vdash p \vee q,\Delta}$$

β-rules

$$\frac{\vdash p,\Delta \qquad \vdash q,\Delta}{\vdash p \wedge q,\Delta} \qquad \frac{\vdash \neg p,\Delta \qquad \vdash \neg q,\Delta}{\vdash \neg(p \vee q),\Delta}$$

$$\frac{\vdash p,\Delta \qquad \vdash \neg q,\Delta}{\vdash \neg(p \longrightarrow q),\Delta}$$

Δ-rules

$$\frac{\vdash p[t/0],\Delta}{\vdash (\exists.p),\Delta}$$

$p[t/0]$ is the formula p with the variable 0 substituted by the term t.

$$\frac{\vdash \neg p[t/0],\Delta}{\vdash \neg(\forall.p),\Delta}$$

$p[t/0]$ is the formula p with the variable 0 substituted by the term t.

δ-rules

$$\frac{\vdash p[t/0],\Delta}{\vdash (\forall.p),\Delta}$$

$p[t/0]$ is the formula p with the variable 0 substituted by the term t (a fresh constant).

$$\frac{\vdash \neg p[t/0],\Delta}{\vdash \neg(\exists.p),\Delta}$$

$p[t/0]$ is the formula p with the variable 0 substituted by the term t (a fresh constant).

Extension rule

$$\frac{\vdash x}{\vdash y}$$

Every element in the list of formulas x must be a member of y.

Figure 1: Proof System

The rule states that we can end a branch in the proof tree if a predicate appears as the first formula of the succedent and if the second formula is the negation of the same predicate (with the same arguments). In other similar systems there are no requirements of the order of a predicate and its negation. As we shall see later, this extra condition is circumvented by application of the extension rule. As we are under the assumptions of classical logic, it is trivial to argue that either a predicate or its negation must be true for the same arguments.

The next rule highlighted is for introduction of the existential quantifier ∃:

```
⊢ sub 0 t p # x ⟹ ⊢ Exi p # x
```

We may existentially quantify the formula over all occurrences of a term `t`.

The following rule is for introduction of a universal quantifier ∀:

```
⊢ sub 0 (Fun i []) p # x ⟹ news i (p # x) ⟹ ⊢ Uni p # x
```

Here it is insufficient to require that the formula is provable for some term `t`. We additionally require that the term must be an arbitrarily chosen constant `i` not previously used in the proof.

Finally we highlight the extension rule:

```
⊢ x ⟹ ext y x ⟹ ⊢ y
```

Recall that the function `ext y x` is only true if all formulas in `x` occur in `y`. However, there may be additional formulas in `y` and the ordering can be different. The main purpose of the rule is to rearrange the formulas in the succedent, seeing that every other rule merely considers the first formula in the succedent.

We invite the reader to study the remaining rules of the sequent calculus.

4 Proofs of Soundness and Completeness

Previous sections have covered the formalization of syntax and semantics for classical first-order logic as well as the sequent calculus proof system in the Isabelle/HOL proof assistant. What remains is to prove key properties of the proof system. We present the most important theorems, namely soundness and completeness, and discuss their significance. The proofs are rather extensive, but we discuss them briefly.

Soundness and completeness theorems revolve around the relation between the formulas `p` that are valid, written `semantics e f g p` where `e`, `f` and `g` are universally quantified, and those that can be proved in the sequent calculus, written `⊢ p`.

For the completeness theorem we consider first a restricted form of validity:

```
(≫ p :: (nat, nat) form) ≡
   ∀(e :: _ ⇒ nat hterm) f g. semantics e f g p
```

The universe consists of Herbrand terms using natural numbers as function identifiers as indicated by the type of the environment `nat ⇒ nat hterm`. Furthermore, we restrict the regular function identifiers and predicate identifiers to natural numbers as well, as indicated by the type of formulas `(nat, nat) form`.

This restricted form of validity is used for the completeness theorem:

```
≫ p ⟹ ⊢ [p]
```

The theorem states that if a formula is valid, in the restricted form, then it can also be proved in the sequent calculus. The proof can only be completed in the current setup if we use a restricted validity

term. From a theoretical point of view, however, restricting the universe, the function identifiers and the predicate identifiers to specific types does not weaken the completeness result. In fact, showing completeness under these assumptions is theoretically more challenging. This is best explained by an argument: Consider the amount of formulas that are valid in all universes. Now also consider the amount of formulas that are valid in a specific universe. Clearly, every formula that is valid in all universes is also valid in a specific universe. However, the specific universe may have additional valid formulas. Consequently, if we show that all valid formulas of a specific universe can be proved, we have already proved all the formulas that are valid in all universes.

The soundness theorem is proved for any model:

```
⊢ [q] ⟹ semantics e f g q
```

The theorem states that the sequent calculus only proves valid formulas. Combining the soundness and completeness result we can conclude that the implication holds in both directions:

```
(≫ [p]) ⟷ (⊢ [p])
```

This means that the sequent calculus can prove all the valid formulas. Furthermore, any formula that is proved by the sequent calculus is valid.

We consider first the soundness and completeness of a tableau calculus whose rules are the dual of our sequent calculus rules. We say a tableau is closed if every branch terminates in a leaf rule. The Isabelle theory `FOL_Tableau` contains the tableau formalization. The proof of `TC_soundness` goes by induction on the inference rules for an arbitrary function denotation. Only two of the cases cannot be proven automatically by Isabelle. The theorem `tableau_completeness` states the completeness property.

The Isabelle theory `FOL_Sequent` contains the sequent formalization. The lemma `SC_soundness` states the soundness property. Dually to the tableau, there are only two cases that are not solved automatically by Isabelle. Finally we show a correspondence between the tableau and sequent calculus which allows to prove completeness of the latter. The proof goes by induction over the tableau rules with a bit of massaging of each induction hypothesis to make us able to apply the corresponding sequent calculus rule. The theorem `SC_completeness` states the completeness property.

5 Teaching the Sequent Calculus

We use the formalization in Isabelle/HOL of the Sequent Calculus Verifier (SeCaV) in a bachelor course on logic for computer science and in this section we discuss our experiences.

In fall 2019 we had 52 students in the DTU course 02156 Logical Systems and Logic Programming (5 ECTS). It is not a mandatory course. Each week we offer 2 hours of lectures and 2 hours of exercises, for 13 weeks in total, plus 4 mandatory assignments and a 2 hour written exam without computer.

We cover some example proofs in the lecture and explain in details the formalization in Isabelle/HOL of the Sequent Calculus Verifier (SeCaV) available here: `https://bitbucket.org/isafol/isafol/src/master/Sequent_Calculus/SeCaV.thy`

In the first week the student proves the following 7 formulas:

$$p \to p \qquad p \to \neg\neg p \qquad \neg\neg p \to p \qquad \forall x p(x) \to p(a) \qquad \forall x p(x) \to \exists x p(x)$$
$$p \to q \to p \qquad p \wedge (p \to q) \to q$$

In the second week the student proves the following 7 formulas:

$$(p \to q) \to p \to q \qquad p \to (p \to q) \to q \qquad \forall x \forall y p(x,y) \to \forall x p(x,x) \qquad p \wedge q \to r \to p \wedge r$$
$$p(a) \wedge (p(a) \to \forall x p(x)) \to \forall x p(x) \qquad \neg p \vee \neg q \to \neg(p \wedge q \qquad (p \to q \to r) \to (p \to q) \to p \to r$$

This year we recommended using refutation and the systematic construction of a semantic tableau before proving the formula in the sequent calculus. It is possible to prove each of the formulas in 16 or fewer steps. The students prove the formulas on paper with the formalization as a reference. We display the main definitions on the projector screen and solve example proofs on the blackboard. All in all it is a nice mixture of old and new technology and we have two teaching assistants in the room in order to help the students.

As part of the final assignment we asked two questions involving SeCaV:

- Question 1

  Consider the following formula: $(\forall x p(x) \wedge \forall x q(x)) \to \forall x (p(x) \wedge q(x))$

  Use refutation and the systematic construction of a semantic tableau. State whether this shows that the formula is valid or not. Show a proof in the Gentzen system $\mathscr{G}$ if the formula is valid — and prove the formula in SeCaV too.

- Question 2

  Prove the following formulas in SeCaV using the steps in the previous question:

  - $(\forall x p(x) \wedge \forall x q(x)) \to \forall x (p(x) \wedge q(x))$
  - $p \wedge q \to q$
  - $p(a,a) \to \exists x \exists y p(x,y)$
  - $(\forall x p(x) \vee \forall x q(x)) \to \forall x (p(x) \vee q(x))$
  - $p \vee (p \to q)$
  - $(p \to q) \vee (q \to r)$

We have collected anonymous answers to two questionnaires. In the first week we had 26 answers and in the second week we had 23 answers. Even though the course has 52 students we estimate that only 45 students are active. However, at our university a course becomes mandatory for a student upon registration for the exam the first time. Furthermore, at most three exam attempts are allowed. We are aware that many of the students follow another course with a large assignment exactly in the second week of our coverage of the sequent calculus.

Most students are in the 5th semester (3rd year of the bachelor programme). A few are just in the 3rd semester and some are in the 7th semester or later. About 60% are on the BSc in Software Technology programme. Almost none had used a proof assistant.

We asked the students about how well they understood the Sequent Calculus Verifier (SeCaV), just after a short repetition of the topics of the first week and again after further explanations in the second week. On a scale from 0 to 9, where 0 is not-at-all, 5 is medium and 9 is fully, the median increased from 4 to 5 but still with answers from 0 to 7. The average started at 3.6 and ended at 4.0 (note that this is still before the exercise session in the second week). For comparison, we also asked about the understanding of a Hilbert system for propositional logic — here the average was 5.0 (median 6 with answers from 0 to 7).

Statistics about the course evaluations and the course grades are publicly available. The course evaluation was very positive. The questions were changed recently so it is difficult to compare with previous years.

In Denmark we use a so-called 7-step-scale, designed to be compatible with the ECTS grading scale:

A For an excellent performance displaying a high level of command of all aspects of the relevant material, with no or only a few minor weaknesses.

B For a very good performance displaying a high level of command of most aspects of the relevant material, with only minor weaknesses.

C For a good performance displaying good command of the relevant material but also some weaknesses.

D For a fair performance displaying some command of the relevant material but also some major weaknesses.

E For a performance meeting only the minimum requirements for acceptance.

Fx For a performance which does not meet the minimum requirements for acceptance.

F For a performance which is unacceptable in all aspects.

The grades have improved this year and we think that this could be related to the use of SeCaV.

Year	A	B	C	D	E	Fx	F	Total
2013	12	14	13	7	1	6	1	54
2014	11	13	21	8	0	8	0	61
2015	14	14	25	5	0	3	0	61
2016	18	17	11	16	0	2	0	64
2017	11	23	23	11	0	5	0	73
2018	10	16	13	1	0	5	1	46
2019	20	13	6	4	0	1	1	45

The slight decrease in the number of students might be due to the introductory machine learning course in the same time slot. That course has increased from 88 students in 2013 to 533 students in 2019.

6 Related Work

A number of formalizations in proof assistants of sequent calculi appear in the literature. Ridge and Margetson [11, 16, 17] formalized in Isabelle a sequent calculus that is also implemented as a verified prover. The calculus is for classical first-order logic formulas in negation normal form and the language of terms consists of only variables. Blanchette, Popescu and Traytel [4] formalized in Isabelle a general framework for soundness and completeness proofs. In the supplementary material to another paper [3], Blanchette and Popescu provide an instance with a formalized tableau for many-sorted first-order logic in negation normal form with equality, but this is not kept up to date with the current version of Isabelle. Braselmann and Koepke [6, 5] formalized in Mizar a sequent calculus for classical first-order logic and proved it sound and complete. Schlöder and Koepke [21] proved it complete for also uncountable languages. Ilik, Lee and Herbelin [9] introduced a Kripke-style semantics for classical first-order logic, and Ilik [8] formalized in Coq the completeness of a sequent calculus with respect to this semantics. Persson [14] formalized, in ALF, a sequent calculus for intuitionistic first-order logic and proved it sound.

Herbelin, Kim and Lee [7] formalized in Coq a sequent calculus for intuitionistic first-order logic with implication and universal quantification as the only logical symbols, and proved it sound and complete with respect to a Kripke-style semantics.

Formalizations of other proof systems for first-order logic also appear, such as axiomatic systems for classical logic (in Isabelle by Jensen, Larsen, Schlichtkrull and Villadsen [10, 20]), natural deduction for classical logic (in Isabelle/HOL by Berghofer [1, 23, 22] and in Phox by Raffali [15]), natural deduction for intuitionistic logic (in ALF by Persson [14]), resolution (by Schlichtkrull [18] in Isabelle/HOL and also in Isabelle/HOL by Schlichtkrull, Blanchette, Traytel and Waldmann [19]) and superposition (by Peltier [13] in Isabelle/HOL). Paulson's formalization in Isabelle/HOL of Gödel's Incompleteness Theorems [12] does not include a proof of completeness of a proof system.

We are not aware of other educational uses of proof assistants based on formalized soundness and completeness proofs, with the exception of our Natural Deduction Assistant (NaDeA) [23]. We first and foremost developed the Sequent Calculus Verifier (SeCaV) in order to obtain a simpler approach to teaching logic, in particular the intuitionistic flavor of natural deduction can complicate proofs and in general sequent calculus is better geared towards automation.

7 Conclusion and Future Work

We have presented a formalization in Isabelle/HOL of a sequent calculus for first-order logic with a programming-like approach to the syntax, the semantics and the proof system. The resulting system, called Sequent Calculus Verifier (SeCaV), is different from our Natural Deduction Assistant (NaDeA) [23], a tool for teaching natural deduction rather than sequent calculus.

We use the formalization in a bachelor course on logic for computer science and have polished the soundness and completeness proofs in order to make them easier to understand for students and researchers. The formalization of the sequent calculus uses no higher-order functions, that is, no function takes a function as argument or returns a function as its result. This can be advantageous in a bachelor course, in particular if the students are not familiar with functional programming.

Future work also includes the extension to a prover (a program that tries to prove a formula using the sequent calculus), perhaps by extending our verified simple prover for first-order logic without constants and functions [24].

Appendix: Formalization in Isabelle/HOL

We show the complete formalization of the sequent calculus except for the proofs of soundness and completeness (hence about 5000 lines are omitted including the formalizations of tableaux).

First the syntax and the Herbrand terms used in the completeness theorem:

```
datatype 'a "term" =
  Var nat |
  Fun 'a ‹'a term list›

datatype ('a, 'b) form =
  FF ("⊥") |
  TT ("⊤") |
  Pre 'b ‹'a term list› |
  Con ‹('a, 'b) form› ‹('a, 'b) form› |
  Dis ‹('a, 'b) form› ‹('a, 'b) form› |
  Imp ‹('a, 'b) form› ‹('a, 'b) form› |
  Neg ‹('a, 'b) form› |
  Uni ‹('a, 'b) form› |
  Exi ‹('a, 'b) form›

datatype 'a hterm = HFun 'a ‹'a hterm list›
```

Then the semantics (with a definition for handling the variable environment):

```
definition shift where
  ‹shift e v z ≡ λn. if n < v then e n else if n = v then z else e (n - 1)›

fun semantics_term and semantics_list where
  ‹semantics_term e f (Var n) = e n› |
  ‹semantics_term e f (Fun i l) = f i (semantics_list e f l)› |
  ‹semantics_list e f [] = []› |
  ‹semantics_list e f (t # l) = semantics_term e f t # semantics_list e f l›

fun semantics where
  ‹semantics e f g ⊥ = False› |
  ‹semantics e f g ⊤ = True› |
  ‹semantics e f g (Pre i l) = g i (semantics_list e f l)› |
  ‹semantics e f g (Con p q) = (semantics e f g p ∧ semantics e f g q)› |
  ‹semantics e f g (Dis p q) = (semantics e f g p ∨ semantics e f g q)› |
  ‹semantics e f g (Imp p q) = (semantics e f g p ⟶ semantics e f g q)› |
  ‹semantics e f g (Neg p) = (¬ semantics e f g p)› |
  ‹semantics e f g (Uni p) = (∀z. semantics (shift e 0 z) f g p)› |
  ‹semantics e f g (Exi p) = (∃z. semantics (shift e 0 z) f g p)›
```

Auxiliary functions for new constants and functions:

```
fun new_term and new_list where
  ‹new_term c (Var n) = True› |
  ‹new_term c (Fun i l) = (if i = c then False else new_list c l)› |
  ‹new_list c [] = True› |
  ‹new_list c (t # l) = (if new_term c t then new_list c l else False)›

fun new where
  ‹new c ⊥ = True› |
  ‹new c ⊤ = True› |
  ‹new c (Pre i l) = new_list c l› |
  ‹new c (Con p q) = (if new c p then new c q else False)› |
  ‹new c (Dis p q) = (if new c p then new c q else False)› |
  ‹new c (Imp p q) = (if new c p then new c q else False)› |
  ‹new c (Neg p) = new c p› |
  ‹new c (Uni p) = new c p› |
  ‹new c (Exi p) = new c p›

fun news where
  ‹news c [] = True› |
  ‹news c (p # x) = (if new c p then news c x else False)›
```

Auxiliary functions for substitution for variables:

```
fun inc_term and inc_list where
  ‹inc_term (Var n) = Var (n + 1)› |
  ‹inc_term (Fun i l) = Fun i (inc_list l)› |
  ‹inc_list [] = []› |
  ‹inc_list (t # l) = inc_term t # inc_list l›

fun sub_term and sub_list where
  ‹sub_term v s (Var n) = (if n < v then Var n else if n = v then s else Var (n - 1))› |
  ‹sub_term v s (Fun i l) = Fun i (sub_list v s l)› |
  ‹sub_list v s [] = []› |
  ‹sub_list v s (t # l) = sub_term v s t # sub_list v s l›

fun sub where
  ‹sub v s ⊥ = ⊥› |
  ‹sub v s ⊤ = ⊤› |
  ‹sub v s (Pre i l) = Pre i (sub_list v s l)› |
  ‹sub v s (Con p q) = Con (sub v s p) (sub v s q)› |
  ‹sub v s (Dis p q) = Dis (sub v s p) (sub v s q)› |
  ‹sub v s (Imp p q) = Imp (sub v s p) (sub v s q)› |
  ‹sub v s (Neg p) = Neg (sub v s p)› |
  ‹sub v s (Uni p) = Uni (sub (v + 1) (inc_term s) p)› |
  ‹sub v s (Exi p) = Exi (sub (v + 1) (inc_term s) p)›
```

Auxiliary functions for expanding a list of formulas (thinning and reordering):

```
fun member where
  ‹member p [] = False› |
  ‹member p (q # x) = (if p = q then True else member p x)›

fun ext where
  ‹ext y [] = True› |
  ‹ext y (p # x) = (if member p y then ext y x else False)›
```

The sequent calculus as an inductive definition:

```
inductive sequent_calculus ("⊩ _" 0) where
  ‹⊩ Pre i l # Neg (Pre i l) # x› |
  ‹⊩ Neg ⊥ # x› |
  ‹⊩ ⊤ # x› |
  ‹⊩ p # x ⟹ ⊩ Neg (Neg p) # x› |
  ‹⊩ Neg p # Neg q # x ⟹ ⊩ Neg (Con p q) # x› |
  ‹⊩ p # q # x ⟹ ⊩ Dis p q # x› |
  ‹⊩ Neg p # q # x ⟹ ⊩ Imp p q # x› |
  ‹⊩ p # x ⟹ ⊩ q # x ⟹ ⊩ Con p q # x› |
  ‹⊩ Neg p # x ⟹ ⊩ Neg q # x ⟹ ⊩ Neg (Dis p q) # x› |
  ‹⊩ p # x ⟹ ⊩ Neg q # x ⟹ ⊩ Neg (Imp p q) # x› |
  ‹⊩ sub 0 t p # x ⟹ ⊩ Exi p # x› |
  ‹⊩ Neg (sub 0 t p) # x ⟹ ⊩ Neg (Uni p) # x› |
  ‹⊩ sub 0 (Fun i []) p # x ⟹ news i (p # x) ⟹ ⊩ Uni p # x› |
  ‹⊩ Neg (sub 0 (Fun i []) p) # x ⟹ news i (p # x) ⟹ ⊩ Neg (Exi p) # x› |
  ‹⊩ x ⟹ ext y x ⟹ ⊩ y›
```

The main theorem with the proofs omitted:

```
abbreviation herbrand_valid ("≫ _" 0) where
  ‹(≫ p :: (nat, nat) form) ≡ ∀(e :: _ ⇒ nat hterm) f g. semantics e f g p›

theorem complete_sound: ‹≫ p ⟹ ⊩ [p]› ‹⊩ [q] ⟹ semantics e f g q›
```

And finally a straightforward corollary:

```
corollary ‹(≫ p) ⟷ (⊩ [p])›
```

The corollary is not as general as the theorem with respect to soundness.

A simple proof of the corollary is "`using complete_sound by fast`" and it can be found automatically using the Sledgehammer panel.

It is possible to check the entire formalization by opening the following file `https://bitbucket.org/isafol/isafol/src/master/FOL_Berghofer/FOL_Appendix.thy` in Isabelle/HOL.

References

[1] Stefan Berghofer (2007): *First-Order Logic According to Fitting*. Archive of Formal Proofs. http://isa-afp.org/entries/FOL-Fitting.shtml, Formal proof development.

[2] Jasmin Christian Blanchette (2019): *Formalizing the Metatheory of Logical Calculi and Automatic Provers in Isabelle/HOL (Invited Talk)*. In: Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP), pp. 1–13, doi:10.1145/3293880.3294087.

[3] Jasmin Christian Blanchette & Andrei Popescu (2013): *Mechanizing the Metatheory of Sledgehammer*. In: Frontiers of Combining Systems - 9th International Symposium, FroCoS 2013, Nancy, France, September 18-20, 2013. Proceedings, pp. 245–260, doi:10.1007/978-3-642-40885-4_17.

[4] Jasmin Christian Blanchette, Andrei Popescu & Dmitriy Traytel (2017): *Soundness and Completeness Proofs by Coinductive Methods*. Journal of Automated Reasoning 58(1), pp. 149–179, doi:10.1007/s10817-016-9391-3.

[5] Patrick Braselmann & Peter Koepke (2005): *Gödel's Completeness Theorem*. Formalized Mathematics 13(1), pp. 49–53.

[6] Patrick Braselmann & Peter Koepke (2005): *A Sequent Calculus for First-Order Logic*. Formalized Mathematics 13(1), pp. 33–39.

[7] Hugo Herbelin, Sun Young Kim & Gyesik Lee (2017): *Formalizing the meta-theory of first-order predicate logic*. Journal of the Korean Mathematical Society 54(5), pp. 1521–1536, doi:10.4134/JKMS.j160546.

[8] Danko Ilik (2010): *Constructive Completeness Proofs and Delimited Control*. Ph.D. thesis, École Polytechnique. https://tel.archives-ouvertes.fr/tel-00529021/document.

[9] Danko Ilik, Gyesik Lee & Hugo Herbelin (2010): *Kripke models for classical logic*. Annals of Pure and Applied Logic 161(11), pp. 1367–1378, doi:10.1016/j.apal.2010.04.007.

[10] Alexander Birch Jensen, John Bruntse Larsen, Anders Schlichtkrull & Jørgen Villadsen (2018): *Programming and verifying a declarative first-order prover in Isabelle/HOL*. AI Communications 31(3), pp. 281–299, doi:10.3233/AIC-180764.

[11] James Margetson & Tom Ridge (2004): *Completeness theorem*. Archive of Formal Proofs. http://isa-afp.org/entries/Completeness.html, Formal proof development.

[12] Lawrence C. Paulson (2013): *Gödel's Incompleteness Theorems*. Archive of Formal Proofs. http://isa-afp.org/entries/Incompleteness.html, Formal proof development.

[13] Nicolas Peltier (2016): *A Variant of the Superposition Calculus*. Archive of Formal Proofs. http://isa-afp.org/entries/SuperCalc.shtml, Formal proof development.

[14] Henrik Persson (1996): *Constructive completeness of intuitionistic predicate logic*. Ph.D. thesis, Chalmers University of Technology. http://web.archive.org/web/20001011101511/http://www.cs.chalmers.se/~henrikp/Lic/.

[15] Christophe Raffalli (2005, possibly earlier): *Krivine's abstract completeness proof for classical predicate logic*. https://github.com/craff/phox/blob/master/examples/complete.phx.

[16] Tom Ridge (2004): *A Mechanically Verified, Efficient, Sound and Complete Theorem Prover For First Order Logic*. Archive of Formal Proofs. http://isa-afp.org/entries/Verified-Prover.shtml, Formal proof development.

[17] Tom Ridge & James Margetson (2005): *A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic*. In: Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings, pp. 294–309, doi:10.1007/11541868_19.

[18] Anders Schlichtkrull (2018): *Formalization of the Resolution Calculus for First-Order Logic*. Journal of Automated Reasoning 61(1), pp. 455–484, doi:10.1007/s10817-017-9447-z.

[19] Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel & Uwe Waldmann (2018): *Formalizing Bachmair and Ganzinger's Ordered Resolution Prover*. In Didier Galmiche, Stephan Schulz & Roberto Sebastiani, editors: *Automated Reasoning*, Springer, pp. 89–107, doi:10.1007/978-3-319-94205-6_7.

[20] Anders Schlichtkrull, Jørgen Villadsen & Andreas Halkjær From (2019): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational Software (ThEdu)*, *EPTCS* 290, pp. 1–13, doi:10.4204/EPTCS.290.1.

[21] Julian J. Schlöder & Peter Koepke (2012): *The Gödel completeness theorem for uncountable languages*. *Formalized Mathematics* 20(3), pp. 199–203, doi:10.2478/v10037-012-0023-z.

[22] Jørgen Villadsen, Andreas Halkjær From, Alexander Birch Jensen & Anders Schlichtkrull: *NaDeA: A Natural Deduction Assistant with a Formalization in Isabelle*. `https://nadea.compute.dtu.dk`.

[23] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2019): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational Software (ThEdu)*, *EPTCS* 290, pp. 14–29, doi:10.4204/EPTCS.290.2.

[24] Jørgen Villadsen, Anders Schlichtkrull & Andreas Halkjær From (2018): *A Verified Simple Prover for First-Order Logic*. In Boris Konev, Josef Urban & Philipp Rümmer, editors: *6th Workshop on Practical Aspects of Automated Reasoning (PAAR)*, *CEUR Workshop Proceedings* 2162, Aachen, pp. 88–104. Available at `http://ceur-ws.org/Vol-2162/`.

[25] Markus Wenzel (1999): *Isar - A Generic Interpretative Approach to Readable Formal Proof Documents*. In: *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, pp. 167–184, doi:10.1007/3-540-48256-3_12.