SeCaV: A Sequent Calculus Verifier in Isabelle/HOL

Asta Halkjær From   Frederik Krogsdal Jacobsen   Jørgen Villadsen
DTU Compute - Department of Applied Mathematics and Computer Science - Technical University of Denmark

We describe SeCaV, a sequent calculus verifier for first-order logic in Isabelle/HOL, and the SeCaV Unshortener, an online tool that expands succinct derivations into the full SeCaV syntax. We leverage the power of Isabelle/HOL as a proof checker for our SeCaV derivations and outline how the design of our rules makes this predictable. The interactive features of Isabelle/HOL make our system transparent. For instance, the user can simply click on a side condition to see its exact definition. Our formalized soundness and completeness proofs pertain exactly to the calculus as exposed to the user and not just to some model of our tool. Users can also write their derivations in the SeCaV Unshortener, which provides a lighter syntax, and expand them for later verification. We have used both tools in our teaching.

1 Introduction

Classical first-order logic plays an important role in mathematical logic and often occupies a central part in textbooks and courses on the subject. The sequent calculus is used to exemplify formal deduction and to show theoretical results in proof theory. It is instructive to write out concrete derivations in the calculus to get a feel for the rules and the method of reasoning. While such derivations can be done with pen and paper and checked for mistakes by human eyes, we argue that there is benefit in computer assistance.

To this end, we introduce SeCaV, a sequent calculus verifier built on top of Isabelle/HOL. SeCaV presents everything within the same unified system: the syntax of formulas, the proof rules, their side conditions, and the way derivations are written. Moreover, it provides immediate feedback to the user on the correctness of their derivations. In combination, this empowers the users when learning to write derivations and gives them an independence that is harder to achieve without computer assistance. We recall Nipkow [12] on the analogy between proof assistants and video games and especially the benefits of immediate feedback:

> *This is in contrast to the usual system of homework that is graded by a teaching assistant and returned a week later, long after the student struggled with it, and at a time when the course has moved on. This delay significantly reduces the impact that any feedback scribbled on the homework may have.*

In this paper we include the teaching and learning aspects only as background motivation, since this is a system description and we have discussed the teaching and learning aspects elsewhere [7, 8]. Our main focus is on the definition of the system itself and especially the benefits of building it on top of Isabelle/HOL. While many tools are implemented independently and perhaps modeled in a proof assistant, we aim to show the benefits of working entirely within Isabelle/HOL.

A completely novel development is the SeCaV Unshortener, a web application that allows experienced users to forgo immediate feedback and in return write more succinct derivations. Such a derivation is automatically expanded into the full SeCaV syntax which can then be verified for correctness.

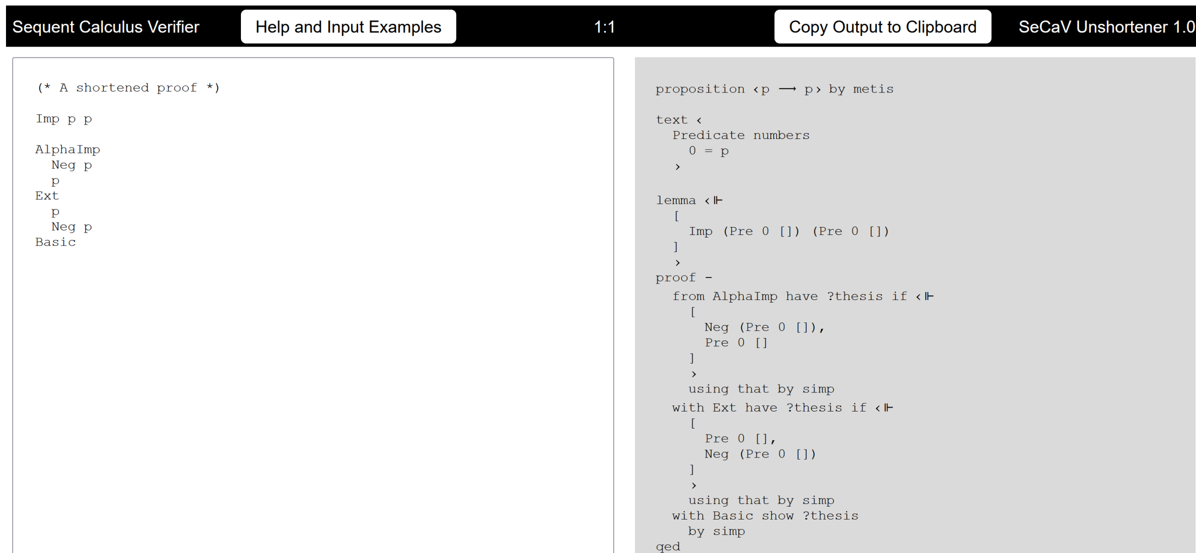

Figure 1: The SeCaV Unshortener generating the example in Figure 6.

The SeCaV Unshortener, shown in Figure 1, allows proofs to be written in a much more compact syntax, cf. Section 5.

We used the SeCaV system in our BSc course “Logical Systems and Logic Programming” in the fall of 2020. 71 students took the 2-hour exam where the exercises in SeCaV were worth 20% of the grade. We also used the SeCaV system in our MSc course “Automated Reasoning” in the spring of 2021, mainly in order to bridge the gap between our micro provers for propositional logic and our Natural Deduction Assistant (NaDeA), cf. Section 2. Here, the students were also introduced to the recent SeCaV Unshortener. 34 students took the 2-hour exam where the exercises in SeCaV were worth 25% of the grade.

The SeCaV system is available online (tested with Isabelle2020 and Isabelle2021):

`https://github.com/logic-tools/secav`

The two relevant files are `SeCaV.thy`, which defines the sequent calculus and proves soundness, and `Sequent_Calculus_Verifier.thy`, which builds on our existing work [9] to prove completeness, cf. Section 4.

The SeCaV Unshortener 1.0 is available online (tested with the Chrome, Edge, Firefox and Safari browsers):

`https://secav.compute.dtu.dk/`

Version 1.0 is fully functional and has an online tutorial with examples. The online tutorial can be used to learn how to actually use the SeCaV system, while the present paper is a description of the system.

We continue by discussing existing work (Section 2) before introducing our system via a number of examples (Section 3). We then introduce SeCaV formally, explain our design considerations, outline the soundness and completeness results, and emphasize the benefits of the Isabelle/HOL integration (Section 4). Next, we give an overview of the SeCaV Unshortener (Section 5) before concluding with thoughts about future work (Section 6).

2 Related Work

There are many tools for sequent calculus, both online and offline. We shall see that, while SeCaV defines one logic and one calculus, the transparency of the system makes the idea of extending or deviating from the system tangible. We have previously explored how proof assistants in general make the different layers and elements of logic visceral [8].

The web application Logitext (`http://logitext.mit.edu`) allows users to derive a sequent by clicking the connective they want to apply a rule to. As such, the rules are almost hidden away from the user who simply sees the appearance of new sub-derivations. Sequoia [13] allows users to input their own rules in a LaTeX format and build derivations from them. It also checks certain meta-theoretical properties of the stated calculus. The online application was unavailable at the time of writing. The `Carnap.io` site [10] allows users to specify their own logic as well as proof system, but in Haskell, which is then compiled to a web application. The offline Sequent Calculus Trainer [6] guides the user away from dead ends by alerting them if the current sequent is determined to be unprovable. AXolotl [4, 5] is an Android app that supports sequent calculus derivations in a classical notation. It is designed to facilitate self-study. Unlike SeCaV, none of these tools provide any formal guarantees of their correctness. Each of them is a bespoke application in a regular programming language.

The Incredible Proof Machine [2, 3] is "an interactive visual theorem prover which represents proofs as port graphs." It distinguishes itself by having a model of this proof representation formalized in Isabelle/HOL and shown to be as strong as natural deduction. Unlike SeCaV, the formalized metatheoretical results only apply to a model of the system.

Our Natural Deduction Assistant (NaDeA) [17] presents natural deduction in a more traditional style. Its metatheory is formalized in Isabelle/HOL and the web application supports exporting proofs that can be verified in Isabelle/HOL, alleviating the problem of potential bugs in the online tool.

Our Students' Proof Assistant [14] exists entirely inside Isabelle/HOL, where it defines a proof assistant within the proof assistant. This helps make proof assistants and their design concrete, but makes the proving experience less natural than using the outer proof assistant directly as done in SeCaV.

Finally, we mention our micro provers for propositional logic [16] whose formalized soundness and completeness results take up only a few dozen lines of Isabelle/HOL. They are based on sequent calculus and can work as a first example in a course, before the full power of first-order logic and SeCaV is introduced.

3 Examples

We use a simple programming-like syntax for formulas in SeCaV and abbreviate it further in the SeCaV Unshortener.

Figure 2 gives an example derivation of the formula $p(a,b) \vee \neg p(a,b)$. The formula is stated on line 3 as the sole member of the one-sided sequent spanning lines 2–4. Recall that such a sequent is understood as a disjunction of formulas. On line 3, the disjunction $\vee$ is written using the constructor *Dis* applied to two arguments separated by a space. For predicates, the constructor *Pre* takes a list of terms as arguments. Here we use *Fun 0* [] and *Fun 1* [], two function symbols taking no arguments, to represent the constants informally called a and b. We make this syntax precise in Section 4.

The first rule application in Figure 2 occurs on line 7. We apply the *AlphaDis* rule backwards, stating that our goal follows from the sequent listed on lines 9–10. This sequent fits the shape of our *Basic* axiom since it starts with a formula that also occurs negated. Lines 14–15 finish the derivation based on this.

```
1  lemma ‹⊩
2    [
3     Dis (Pre 0 [Fun 0 [], Fun 1 []]) (Neg (Pre 0 [Fun 0 [], Fun 1 []]))
4    ]
5    ›
6  proof -
7    from AlphaDis have ?thesis if ‹⊩
8      [
9       Pre 0 [Fun 0 [], Fun 1 []],
10      Neg (Pre 0 [Fun 0 [], Fun 1 []])
11     ]
12     ›
13     using that by simp
14   with Basic show ?thesis
15     by simp
16 qed
```

Figure 2: A sample SeCaV derivation in Isabelle/HOL.

```
1 Dis p[a, b] (Neg p[a, b])
2
3 AlphaDis
4   p[a, b]
5   Neg p[a, b]
6 Basic
```

Figure 3: The sample SeCaV derivation in Figure 2 written in the syntax of the SeCaV Unshortener.

In the above we focused on the things essential to a human reader: the goal, the rules and their resulting sequents. The remaining lines and keywords are for the benefit of Isabelle/HOL: they fit our derivations into the Isar syntax [18] giving us all the verification benefits of Isabelle/HOL. If a rule is applied wrong, the editor tells us!

Our calculus is designed such that this boilerplate is predictable. With the exception of two rules that require clarification when more than one variable is in play, we have yet to encounter a rule application that cannot be justified by Isabelle/HOL's simplifier. This predictability means that we can write down only the essential parts of the derivation and then generate the boilerplate with the SeCaV Unshortener. Figure 3 contains an example of this, namely the same derivation as Figure 2. It starts with the goal formula on line 1, then the first rule application on line 3, the resulting sequent spans lines 4–5 and finally line 6 finishes the derivation. In fact, the SeCaV Unshortener produced the Isabelle/HOL code in Figure 2 automatically from this representation. For brevity, we will generally favor the short representation.

3.1 Instantiating Quantifiers

Consider the additional example in Figure 4, which contains a derivation of:

$$(\forall x. \forall y. p(x,y)) \rightarrow p(a,a)$$

```
1  Imp (Uni (Uni (p[1, 0]))) p[a, a]
2
3  AlphaImp
4    Neg (Uni (Uni p[1, 0]))
5    p[a, a]
6  GammaUni[a]
7    Neg (Uni p[a, 0])
8    p[a, a]
9  GammaUni
10   Neg p[a, a]
11   p[a, a]
12 Ext
13   p[a, a]
14   Neg p[a, a]
15 Basic
```

Figure 4: SeCaV Unshortener example with instantiation of quantifiers.

We write the formulas using de Bruijn indices to match the Isabelle/HOL formalization. In the example, the variable 1 is bound by the outermost quantifier and 0 by the innermost. The rule application on line 3 is propositional and straightforward: the implication holds if either the antecedent does not or the consequent does. Consider instead line 6 where several things occur. First, the *GammaUni* rule allows us to derive a negated, universally quantified formula from an example. Applied backwards, we can insert any term for the bound variable while eliminating the quantifier. We do so, replacing the bound variable with the term `a`. Second, the notation `[a]` becomes a hint to Isabelle/HOL that `a` is the term used to replace the bound variable. This ensures that the simplifier can verify the correctness and is necessary when more than one variable occurs in the term (we omit it on line 9). Line 12 applies the *Ext* rule that rearranges the sequent such that the *Basic* rule applies on line 15. We insist that the entire sequent is written down after each rule application so it is possible to read each application without referring back to previous ones.

3.2 Branching Derivations

As a final example consider the longer Figure 5, which includes branching rules. Lines 1–24 proceed using rules similar to those we have already seen. We cover them in detail in Section 4. Line 25 applies the *BetaImp* rule, which relies on two sub-derivations. The first sequent that needs to be derived is given on lines 26–28 and the second sequent on lines 30–32, with a plus symbol (`+`) separating the two. The application of *Basic* on line 33 closes the first branch and the rest of the derivation concerns only the second one.

The order of the two branches is not important for the Isabelle/HOL verification and the subsequent rules can be applied to either of the branches or even both at the same time. For the sake of human-readability, however, we suggest working on the first branch.

Figure 5 displays another feature of our calculus that is worth pointing out. On line 37, the *Ext* rule is used not just to rearrange the sequent, but also to drop a formula that was only necessary on one branch. As such, it can be used to tidy up sequents during a derivation.

```
1  Imp (Uni (Imp p[0] q[0])) (Imp (Exi p[0]) (Exi q[0]))
2
3  AlphaImp
4    Neg (Uni (Imp p[0] q[0]))
5    Imp (Exi p[0]) (Exi q[0])
6  Ext
7    Imp (Exi p[0]) (Exi q[0])
8    Neg (Uni (Imp p[0] q[0]))
9  AlphaImp
10   Neg (Exi p[0])
11   Exi q[0]
12   Neg (Uni (Imp p[0] q[0]))
13 DeltaExi
14   Neg p[a]
15   Exi q[0]
16   Neg (Uni (Imp p[0] q[0]))
17 Ext
18   Neg (Uni (Imp p[0] q[0]))
19   Neg p[a]
20   Exi q[0]
21 GammaUni
22   Neg (Imp p[a] q[a])
23   Neg p[a]
24   Exi q[0]
25 BetaImp
26   p[a]
27   Neg p[a]
28   Exi q[0]
29 +
30   Neg q[a]
31   Neg p[a]
32   Exi q[0]
33 Basic
34   Neg q[a]
35   Neg p[a]
36   Exi q[0]
37 Ext
38   Exi q[0]
39   Neg q[a]
40 GammaExi
41   q[a]
42   Neg q[a]
43 Basic
```

Figure 5: SeCaV Unshortener example with a branching derivation.

```
lemma ‹⊩ [ Imp (Pre 0 []) (Pre 0 []) ] ›
proof -
  from AlphaImp have ?thesis if ‹⊩ [ Neg (Pre 0 []), Pre 0 [] ]›
    using that by simp
  with Ext have ?thesis if ‹⊩ [ Pre 0 [], Neg (Pre 0 []) ]›
    using that by simp
  with Basic show ?thesis
    by simp
qed
```

Proof state Auto update Update Search: 100%

```
proof (chain)
picking this:
  ⊩ Neg ?p # ?q # ?z ⟹ ⊩ Imp ?p ?q # ?z
```

Figure 6: A compressed SeCaV derivation in the Isabelle/jEdit editor. The bottom panel shows the rule under the cursor. The system automatically instantiates each variable (*?p, ?q, ?z*) appropriately.

4 SeCaV

Figure 6 shows a SeCaV derivation as it appears in the Isabelle/jEdit editor. For brevity, we have removed a number of line breaks. The SeCaV Unshortener allows the same proof to be written in a much more compact syntax, cf. Figure 1.

In this section we formally describe the SeCaV system: its syntax and semantics, proof rules, metatheory and Isabelle/HOL integration.

4.1 Syntax and Semantics

The syntax of terms and formulas in SeCaV is formally defined by the following two Isabelle/HOL datatype declarations:

datatype *tm* = *Fun nat* ⟨ *tm list* ⟩ | *Var nat*
datatype *fm* = *Pre nat* ⟨ *tm list* ⟩ | *Imp fm fm* | *Dis fm fm* | *Con fm fm* | *Exi fm* | *Uni fm* | *Neg fm*

Terms are either functions identified by a natural number and applied to a list of terms, or de Bruijn indices. Formulas are either predicates, also identified by a natural number and applied to a list of terms, a connective applied to an appropriate number of formulas, or a quantifier. The embedding of SeCaV into Isabelle/HOL means that we do not need to write a parser for this syntax. As seen in Figure 6 we can write down a formula immediately. We use a simple programming-like syntax here, but it is also possible to define a more regular infix syntax with various precedences and associativity.

To formalize metatheory, like the soundness and completeness of our proof system, it is essential to assign a meaning to our formulas. We can do this because of our deep embedding of the syntax as a datatype. While we could also use Isabelle as the generic proof assistant it is, define our logic in that style and still have it check our proofs, doing so would prevent us from formalizing our metatheory, and we would not even be able to prove soundness.

The following functions interpret terms and formulas into Isabelle/HOL's higher-order logic, given a variable assignment e, a function denotation f and a predicate denotation g:

```
primrec 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›
```

```
primrec semantics where
 ‹semantics e f g (Pre i l) = g i (semantics-list e f l)› |
 ‹semantics e f g (Imp 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 (Con p q) = (semantics e f g p ∧ semantics e f g q)› |
 ‹semantics e f g (Exi p) = (∃x. semantics (shift e 0 x) f g p)› |
 ‹semantics e f g (Uni p) = (∀x. semantics (shift e 0 x) f g p)› |
 ‹semantics e f g (Neg p) = (¬ semantics e f g p)›
```

The two quantifier clauses make the interpretation of de Bruijn indices explicit. When interpreting a quantifier, the function *shift* adjusts the variable assignment e so index 0 points at the newly quantified variable. Its general definition is:

```
definition ‹shift e v x ≡ λn. if n < v then e n else if n = v then x else e (n − 1)›
```

This use matches the intuition that variable 0 is bound by the "nearest" quantifier. Similarly, the existing indices are shifted by one since they appear one scope further out.

4.2 Substitution

While on the topic of de Bruijn indices we now cover how substitution is formalized, as we need it to specify our proof rules. We include the definitions of such helper functions to make our presentation self-contained. Our substitution function on formulas, *sub*, is designed to be used whenever we instantiate a quantifier. The application *sub v s p* substitutes variable v for the term s in formula p. During this substitution, we ensure that no variable in s gets bound by a quantifier in p. We define the function by structural recursion:

```
primrec sub where
 ‹sub v s (Pre i l) = Pre i (sub-list v s l)› |
 ‹sub v s (Imp p q) = Imp (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 (Con p q) = Con (sub v s p) (sub v s q)› |
 ‹sub v s (Exi p) = Exi (sub (v + 1) (inc-term s) p)› |
 ‹sub v s (Uni p) = Uni (sub (v + 1) (inc-term s) p)› |
 ‹sub v s (Neg p) = Neg (sub v s p)›
```

Only the predicate and quantifier cases are interesting; the rest simply apply the substitution to the sub-formulas. In the predicate case we use the function *sub-list* to apply the substitution across the list of argument terms. It is defined mutually with *sub-term*:

```
primrec 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›
```

```
              inductive sequent-calculus (‹⊩ -› 0) where
Basic         ‹⊩ p # z› if ‹member (Neg p) z› |
AlphaDis      ‹⊩ Dis p q # z› if ‹⊩ p # q # z› |
AlphaImp      ‹⊩ Imp p q # z› if ‹⊩ Neg p # q # z› |
AlphaCon      ‹⊩ Neg (Con p q) # z› if ‹⊩ Neg p # Neg q # z› |
BetaCon       ‹⊩ Con p q # z› if ‹⊩ p # z› and ‹⊩ q # z› |
BetaImp       ‹⊩ Neg (Imp p q) # z› if ‹⊩ p # z› and ‹⊩ Neg q # z› |
BetaDis       ‹⊩ Neg (Dis p q) # z› if ‹⊩ Neg p # z› and ‹⊩ Neg q # z› |
GammaExi      ‹⊩ Exi p # z› if ‹⊩ sub 0 t p # z› |
GammaUni      ‹⊩ Neg (Uni p) # z› if ‹⊩ Neg (sub 0 t p) # z› |
DeltaUni      ‹⊩ Uni p # z› if ‹⊩ sub 0 (Fun i []) p # z› and ‹news i (p # z)› |
DeltaExi      ‹⊩ Neg (Exi p) # z› if ‹⊩ Neg (sub 0 (Fun i []) p) # z› and ‹news i (p # z)› |
NegNeg        ‹⊩ Neg (Neg p) # z› if ‹⊩ p # z› |
Ext           ‹⊩ y› if ‹⊩ z› and ‹ext y z›
```

Figure 7: SeCaV proof rules in Isabelle/HOL with associated names inserted manually to the left.

There are two cases for *sub-term*. At variables we leave smaller variables alone, substitute those matching the target index *v*, and decrement larger variables to account for the instantiated quantifier whose scope is now gone. At function symbols, we simply apply the substitution across the arguments.

Returning to *sub*, in the quantifier cases we increment *v* to account for the quantifier whose scope we are now under. For the same reason, we use the function *inc-term* to increment the variables in *s*. It is defined mutually with *inc-list*:

```
primrec 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›
```

4.3 Proof System

Our sequent calculus is one-sided system like System G by Ben-Ari [1], which inspired it. While two-sided systems have a certain elegance, a one-sided system has a couple of advantages. First, it can be explained and understood as simply meta-notation for a disjunction between formulas. Second, it can be formalized as a single list of formulas, in turn reducing the syntactic burden of writing down a sequent. Consider the SeCaV Unshortener syntax in e.g. Figure 4. Rule applications and sequents alternate throughout the derivation, with no need for a special symbol to distinguish a left- and right-hand side of the sequent.

Figure 7 contains our proof rules. We use Smullyan's uniform notation [15] for the names, designating whether they branch (β) or not (α) and whether the quantifiers can be built from any term (γ) or only a fresh witness (δ). Notice that each rule is actually a schema: the symbols *p* and *q* etc. are not concrete formulas but metavariables that can be instantiated with any type-correct value.

4.3.1 Proof Rules

As mentioned our sequents are lists of formulas, which means that they are ordered. In Isabelle/HOL, the symbol # separates the head and tail of a list. All our rules except *Ext* use this notation to replace the head of the list.

Figure 7 begins with the only axiom, *Basic*, which states that a sequent with some formula *p* at the head and *Neg p* somewhere in the tail can be derived. That is, we can derive the sequent ⊩ *p* # *z*, **if** we can demonstrate that *Neg p* is a member of *z*, i.e. *member* (*Neg p*) *z*. The function *member* is defined in SeCaV as a simple primitive recursive function on lists for users to inspect (or even run):

primrec *member* **where**
⟨*member p* [] = *False*⟩ |
⟨*member p* (*q* # *z*) = (*if p* = *q then True else member p z*)⟩

Since a sequent is understood as a disjunction, those of the *Basic* shape are clearly valid. To derive a sequent that contains both some *p* and a corresponding *Neg p* but not necessarily in the order dictated by *Basic*, the final rule in Figure 7, *Ext*, can be used. As we have seen in Section 3, it allows one to rearrange the formulas of a sequent or to drop formulas. It should be read as follows: if we can derive a sequent *z* and the sequent *y* is an *ext*ension of *z*, then we are allowed to derive *y* itself. The function *ext* builds on *member* to check that the formulas in *y* constitute a superset of those in *z*:

primrec *ext* **where**
⟨*ext y* [] = *True*⟩ |
⟨*ext y* (*p* # *z*) = (*if member p y then ext y z else False*)⟩

Alpha Rules After *Basic* follow three α-rules that rely on just one sub-derivation. The *AlphaDis* rule moves the connective from the object language into the metalanguage, simply removing the connective and adding the two disjuncts to the sequent. To show that an implication *Imp p q* holds, we can either falsify the antecedent, *Neg p*, or show the conclusion, *q*, so the rule *AlphaImp* replaces an implication with exactly those formulas. Finally, *AlphaCon* states that to show *Neg* (*Con p q*) we can falsify either *p* or *q*. The pen-ultimate rule *NegNeg* also relies on just one sub-derivation, so we include it here. It introduces a double negation.

Beta Rules After the first α-rules follow three β-rules that make the derivation branch. A conjunction only holds if both conjuncts do, so the *BetaCon* rule adds each to separate sub-derivations. The *BetaImp* rule works on a negated implication, *Neg* (*Imp p q*), and states that we must both prove *p* and falsify *q*. Finally, *BetaDis* replaces *Neg* (*Dis p q*) with both *Neg p* and *Neg q* on separate branches, as both *p* and *q* must be falsified for their disjunction to be falsified.

Gamma Rules The γ-rules apply to formulas that are effectively existentially quantified. Such formulas can be built from any witnessing term. The next rule exemplifies this: *GammaExi* derives the sequent *Exi p* # *z* from *sub 0 t p* # *z*. In the sub-derivation, we have the formula *p* with its outermost variable instantiated with the term *t* using the *sub* function. This term, *t*, witnesses the existence so in the conclusion we quantify over the variable, giving *Exi p*, instead of substituting it.

The *GammaUni* rule applies when the head of the sequent is *Neg* (*Uni p*) for some formula *p*. In the sub-derivation, the head is replaced by *Neg* (*sub 0 t p*) since falsifying *p* instantiated with any term *t* is enough to falsify *Uni p*.

Delta Rules The two δ-rules apply to formulas that are effectively universally quantified. To prove a universal quantifier, we cannot abstract over just any term like with δ-rules. Instead, the term must be arbitrary, i.e. *new* to the sequent as formalized below, so that any other term could stand in its place. Sometimes this is called *fresh* rather than *new*.

The *DeltaUni* rule allows the derivation of *Uni p # z* if we can derive *sub 0 (Fun i []) p # z* where the name *i* does not occur in either *p* or *z*, as checked by *news i (p # z)*. The *DeltaExi* rule is similar but applies when the head of the sequent is *Neg (Exi p)*. We define newness similarly to the other side conditions. The function *new* defines what it means for a function symbol *c* to be new to a formula:

primrec *new* **where**
⟨*new c (Pre i l) = new-list c l*⟩ |
⟨*new c (Imp 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 (Con p q) = (if new c p then new c q else False)*⟩ |
⟨*new c (Exi p) = new c p*⟩ |
⟨*new c (Uni p) = new c p*⟩ |
⟨*new c (Neg p) = new c p*⟩

Only the predicate case is interesting; the rest simply consider sub-formulas. The function *new-list* checks whether *c* is new to a list of terms. It is defined mutually with *new-term*:

primrec *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)*⟩

If the term is a variable then the function symbol *c* is obviously new. Otherwise the term is a function application and we check whether the two function symbols coincide. If they do, *c* is not new, but even if they do not, *c* still has to be new to the arguments of the function, which we check with *new-list*.

The entry point to these functions is *news*, which checks whether the function symbol *c* is new to the given sequent:

primrec *news* **where**
⟨*news c [] = True*⟩ |
⟨*news c (p # z) = (if new c p then news c z else False)*⟩

4.3.2 Rule Design

After seeing how the proof rules work, we want to point out several choices in their design.

While sequents are often unordered (cf. Ben-Ari [1], Nipkow and Michaelis [11]) ours do have an order. Where Ben-Ari underlines the formula in a sequent that the next rule applies to, our rules always work on the first one. This simplification has several benefits: (i) it makes the formalization simpler to state and the success of the verification easier to predict, (ii) it reduces the notational burden in the SeCaV Unshortener syntax and (iii) it provides a straight-forward recipe for new users to get started: "simply look at the first formula and see if any rules apply." Of course, the recipe in (iii) may result in derivations that are longer than necessary and because of our γ-rules the recipe may even be insufficient for more advanced formulas, but such formulas are also out of reach if the user starts out overwhelmed and never gets going. The simplification forces us to include a structural rule like *Ext*. This is the price of separating concerns, but as we have seen, *Ext* can also, for instance, be used to drop formulas on branches that do not need them.

Another point is that our rules do not just add to the sequent, but always replace the head of it in the sub-derivation(s). Even the γ-rules do this, even though we may want to instantiate such formulas with several different terms (cf. Ben-Ari [1]). Again we separate concerns: to apply a γ-rule twice, first duplicate the formula with *Ext* and then apply the rule. This makes such duplication deliberate instead of an arbitrary feature of γ-rules that is sometimes useful and sometimes not.

```
lemma ⟨⊩
  [
   GOAL
  ]
  ⟩
proof −
 from RULE_1 have ?thesis if ⟨⊩
   [
    SUBGOAL_1
         ⋮
    SUBGOAL_N
   ]
   ⟩
   using that by simp
              ⋮
 with Basic show ?thesis
   by simp
qed
```

Figure 8: SeCaV derivation template.

Our last point relates to the benefit of specifying our system in Isabelle/HOL: every operation and side condition is explicitly spelled out and *computational*. It may seem obvious what membership in a sequent entails or what it means for a constant name to be *new*, but something like substitution is notoriously tricky, no matter the representation. In our system, these things are implemented by simple functional programs, accessible directly in the system. They are not opaque pieces of natural language or hidden away in an implementation, but can be inspected by the user and even run on simple examples.

It speaks to the complexity of substitution that only rules that involve this operation can be hard to verify: *GammaUni* and *GammaExi*. Otherwise, our definitions, like those of *member* and *ext*, play on the strengths of the Isabelle/HOL simplifier: they are simple functional programs that can be checked by rewriting. Similarly, by letting our rules work on the first formula in the sequent, it becomes a simple problem to unify it with the current goal, solving the meta-variables to check if they match the stated sub-derivation. These design choices make the system predictable to work with.

4.4 Writing Proofs

As alluded to in Section 3, derivations in SeCaV follow a common template, which we have sketched in Figure 8. Users of the system only need to worry about filling in the GOAL and a number of RULE applications with corresponding SUBGOALs. Those curious about Isabelle/HOL can investigate the meaning of the remaining keywords if they want to. Since derivations are entirely textual, it is easy to copy this template, or parts of it, from given examples or previous derivations.

In Figure 9 we see the error message obtained when *AlphaDis* is replaced by *AlphaImp* in Figure 2. Isabelle/HOL will highlight the **by** keyword following the rule application, notifying the user that something is wrong. The error is then displayed in the output panel when placing the cursor over the highlight. Line 1 in Figure 9 contains the rule being applied, lines 2–3 the stated subgoal and lines 4–5 the goal itself. By inspection we see that since *Imp* and *Dis* do not match, the rule does not apply to produce the

```
Failed to finish proof:
goal (1 subgoal):
```

1 *1.* $(\bigwedge p\ q\ z.\ \Vdash \mathit{Neg}\ p\ \#\ q\ \#\ z \Longrightarrow \Vdash \mathit{Imp}\ p\ q\ \#\ z) \Longrightarrow$
2 $\Vdash [\mathit{Pre}\ 0\ [\mathit{Fun}\ 0\ [], \mathit{Fun}\ (\mathit{Suc}\ 0)\ []],$
3 $\mathit{Neg}\ (\mathit{Pre}\ 0\ [\mathit{Fun}\ 0\ [], \mathit{Fun}\ (\mathit{Suc}\ 0)\ []])] \Longrightarrow$
4 $\Vdash [\mathit{Dis}\ (\mathit{Pre}\ 0\ [\mathit{Fun}\ 0\ [], \mathit{Fun}\ (\mathit{Suc}\ 0)\ []])$
5 $(\mathit{Neg}\ (\mathit{Pre}\ 0\ [\mathit{Fun}\ 0\ [], \mathit{Fun}\ (\mathit{Suc}\ 0)\ []]))]$

Figure 9: Error message when *AlphaDis* is replaced by *AlphaImp* in Figure 2.

goal (the subgoal does not match either).

We obtain this error message completely for free by leveraging Isabelle/HOL as the platform for specifying our system.

We also inherit the interactive features of Isabelle/HOL. Users can click a name and be taken to its definition, e.g. that of *sub* if in doubt about substitution. Or when following the template, they can put their cursor on an applied rule and see its definition in the output panel as in Figure 6. “This derivation uses *AlphaImp*, how does that look again? Oh, right: $\Vdash \mathit{Neg}\ ?p\ \#\ ?q\ \#\ ?z \Longrightarrow \Vdash \mathit{Imp}\ ?p\ ?q\ \#\ ?z$.”

4.5 Soundness and Completeness

Having specified SeCaV in a proof assistant enables us to give certain guarantees about not just our calculus but its implementation as well. The first and most obvious is soundness of the rules. If we can derive a sequent, then for any interpretation, some formula in the sequent is satisfied:

theorem *sound*: $\langle \Vdash z \Longrightarrow \exists p \in \mathit{set}\ z.\ \mathit{semantics}\ e\ f\ g\ p \rangle$

See the formalization for the proof which works by induction over the rules and using a substitution lemma. We immediately obtain that if a derivable sequent contains just one formula then that formula must be valid:

corollary $\langle \Vdash [p] \Longrightarrow \mathit{semantics}\ e\ f\ g\ p \rangle$

We build the completeness proof on existing work in the Archive of Formal Proofs, namely the entry “A Sequent Calculus for First-Order Logic” [9]. In less than a hundred lines of Isabelle/HOL, we relate our syntax, semantics, side conditions and operations to an existing sequent calculus formalization and show that derivations in that one ($\vdash$) can be translated into ours ($\Vdash$) (cf. the formalization):

lemma *sim*: $\langle (\vdash x) \Longrightarrow (\Vdash (\mathit{map}\ \mathit{to\text{-}fm}\ x)) \rangle$

From these components, completeness follows straightforwardly:

theorem *complete-sound*: $\langle \gg p \Longrightarrow \Vdash [p] \rangle \langle \Vdash [q] \Longrightarrow \mathit{semantics}\ e\ f\ g\ q \rangle$

The symbol $\gg$ abbreviates validity in the universe of Herbrand terms. Validity in just this universe is enough to show the existence of a derivation (a slightly stronger completeness result than assuming validity in all universes). The soundness result, conversely, implies validity in any universe, as e, f and g can be picked at will.

These aspects can be ignored when working with the system, but used to concretize discussions of soundness and completeness in a course.

Sequent Calculus Verifier | Help and Input Examples | 1:1 | Copy Output to Clipboard | SeCaV Unshortener 1.0

```
(* A shortened proof - with a mistake *)

Imp p p

AlphaImp
  Neg p
  p
Basic
```

```
(* Warning:
(Basic) Sequent not a tautology/Positive formula is not the first
*)
proposition ‹p ⟶ p› by metis

text ‹
  Predicate numbers
    0 = p
  ›

lemma ‹⊩
  [
    Imp (Pre 0 []) (Pre 0 [])
  ]
  ›
proof -
  from AlphaImp have ?thesis if ‹⊩
    [
      Neg (Pre 0 []),
      Pre 0 []
    ]
    ›
    using that by simp
  with Basic show ?thesis
    by simp
qed
```

Figure 10: The SeCaV Unshortener generating the example in Figure 6 — With a mistake.

4.6 Isabelle/HOL Integration

Next, we want to reiterate a few consequences of building our system on top of Isabelle/HOL.

In terms of infrastructure we are relieved from implementing a lot of work ourselves. By giving two simple datatype declarations, in a syntax resembling BNF, we get to reuse Isabelle/HOL's parser when writing formulas in our object logic. The same reusability applies to the proof system, both in its declarative specification using the **inductive** command and when writing concrete derivations. Given the declaration in Figure 7, we inherit proof checking completely for free: Isabelle/HOL verifies the correctness of derivations for us.

The use of Isabelle/HOL also means that we can reuse its mature graphical editor Isabelle/jEdit. Besides regular editor features like undo, Isabelle/jEdit continually checks the correctness of what the user enters. As seen, it produces decent errors that are displayed in the same window as the derivation and where the offending rule application is highlighted directly in the derivation. Finally, all definitions used by the system can be inspected by using the editor to look them up within the same interface.

5 SeCaV Unshortener

While the embedding of SeCaV into Isabelle/HOL makes it possible for users to get quick feedback on their proofs and provides us with an editor for free, actually writing out the proofs in the Isabelle/HOL syntax can becomes tedious for experienced users. To remedy this, we have introduced the SeCaV Unshortener, shown in Figure 1 and Figure 10. It allows proofs to be written in a much more compact syntax that resembles the style one might use when writing pen-and-paper proofs.

The SeCaV Unshortener is a web application whose main page consists of two panes. The first pane is a text area in which the user can write proofs in the compact SeCaV Unshortener syntax. The second pane contains the result of "unshortening" the proofs written in the first pane into the full SeCaV syntax, ready to be copied into Isabelle/HOL for verification. For each proof, the SeCaV Unshortener also generates a representation of the statement in usual logical syntax and a mapping from predicate and function names to the natural numbers used in the full SeCaV syntax. The first of these is useful to detect misunderstandings in the statement to be proved, while the latter is needed to relate the actual proof to

the representation in usual logical syntax, and may also be used to quickly detect typos in names. The second pane reacts to changes in the first pane in real time, and will contain an error message if a proof is written using wrong syntax. Along the top of the main page is a link to a page containing extensive help and a number of examples, an indication of the currently selected line and column in the first pane (for use with error messages), and a button that copies the unshortened proof to the clipboard.

The SeCaV Unshortener provides a canonical formatting of proofs and a lighter, more readable syntax at the expense of introducing another step in the proof procedure. The SeCaV Unshortener does not actually verify the proofs entered into it. The proof must be copied into Isabelle/HOL for verification. But in addition to unshortening proofs, the SeCaV Unshortener adds warnings when the proofs will most likely be rejected by Isabelle/HOL.

The SeCaV Unshortener is implemented in PureScript using the Concur web UI framework with a React backend. The application is compiled down to a few JavaScript, HTML, and CSS files, which can be hosted on any web server or downloaded for local use.

6 Conclusion

We have introduced SeCaV, a sequent calculus verifier built on top of Isabelle/HOL, and explained the syntax, semantics, and proof rules of the system. SeCaV is designed to be easy to learn and understand for students, and is therefore implemented as a number of simple functional programs utilizing the interactive Isabelle/jEdit editor to allow inspection of every part of the system. We have used Isabelle/HOL to prove soundness and completeness of the SeCaV calculus exactly as users work with it.

We have also introduced the SeCaV Unshortener, a web application that allows users of SeCaV to omit the boilerplate notation needed for the embedding in Isabelle/HOL. Future work includes exploring how SeCaV and the SeCaV Unshortener can be integrated further, either by embedding the SeCaV Unshortener into the Isabelle/jEdit editor or by integrating the verifier into the SeCaV Unshortener.

Acknowledgements

We want to thank Agnes Moesgård Eschen, Alexander Birch Jensen, Simon Tobias Lund, Emmanuel André Ryom and Anders Schlichtkrull for comments on a draft.

References

[1] Mordechai Ben-Ari (2012): *Mathematical Logic for Computer Science*. Springer.

[2] Joachim Breitner (2016): *Visual Theorem Proving with the Incredible Proof Machine*. In Jasmin Christian Blanchette & Stephan Merz, editors: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, *Lecture Notes in Computer Science* 9807, Springer, pp. 123–139, doi:10.1007/978-3-319-43144-4_8.

[3] Joachim Breitner & Denis Lohner (2016): *The meta theory of the Incredible Proof Machine*. *Archive of Formal Proofs*. `https://isa-afp.org/entries/Incredible_Proof_Machine.html`, Formal proof development.

[4] David M. Cerna, Rafael P. D. Kiesel & Alexandra Dzhiganskaya (2019): *A Mobile Application for Self-Guided Study of Formal Reasoning*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 8th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2019, Natal, Brazil, 25th August 2019*, *EPTCS* 313, pp. 35–53, doi:10.4204/EPTCS.313.3.

[5] David M. Cerna, Martina Seidl, Wolfgang Schreiner, Wolfgang Windsteiger & Armin Biere (2020): *Aiding an Introduction to Formal Reasoning Within a First-Year Logic Course for CS Majors Using a Mobile Self-Study App*. In Michail N. Giannakos, Guttorm Sindre, Andrew Luxton-Reilly & Monica Divitini, editors: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2020, Trondheim, Norway, June 15-19, 2020*, ACM, pp. 61–67, doi:10.1145/3341525.3387409.

[6] Arno Ehle, Norbert Hundeshagen & Martin Lange (2017): *The Sequent Calculus Trainer with Automated Reasoning - Helping Students to Find Proofs*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 August 2017*, *EPTCS* 267, pp. 19–37, doi:10.4204/EPTCS.267.2.

[7] Asta Halkjær From, Alexander Birch Jensen, Anders Schlichtkrull & Jørgen Villadsen (2019): *Teaching a Formalized Logical Calculus*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 8th International Workshop on Theorem Proving Components for Educational Software, ThEdu@CADE 2019, Natal, Brazil, 25th August 2019*, *EPTCS* 313, pp. 73–92, doi:10.4204/EPTCS.313.5.

[8] Asta Halkjær From, Jørgen Villadsen & Patrick Blackburn (2020): *Isabelle/HOL as a Meta-Language for Teaching Logic*. In Pedro Quaresma, Walther Neuper & João Marcos, editors: *Proceedings 9th International Workshop on Theorem Proving Components for Educational Software, ThEdu@IJCAR 2020, Paris, France, 29th June 2020*, *EPTCS* 328, pp. 18–34, doi:10.4204/EPTCS.328.2.

[9] Asta Halkjær From (2019): *A Sequent Calculus for First-Order Logic*. *Archive of Formal Proofs*. `https://isa-afp.org/entries/FOL_Seq_Calc1.html`, Formal proof development.

[10] Graham Leach-Krouse (2017): *Carnap: An Open Framework for Formal Reasoning in the Browser*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 6th International Workshop on Theorem proving components for Educational software, ThEdu@CADE 2017, Gothenburg, Sweden, 6 August 2017*, *EPTCS* 267, pp. 70–88, doi:10.4204/EPTCS.267.5.

[11] Julius Michaelis & Tobias Nipkow (2018): *Formalized Proof Systems for Propositional Logic*. In A. Abel, F. Nordvall Forsberg & A. Kaposi, editors: *23rd Int. Conf. Types for Proofs and Programs (TYPES 2017)*, *LIPIcs* 104, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 6:1–6:16.

[12] Tobias Nipkow (2012): *Teaching Semantics with a Proof Assistant: No More LSD Trip Proofs*. In Viktor Kuncak & Andrey Rybalchenko, editors: *Verification, Model Checking, and Abstract Interpretation*, Springer, pp. 24–38.

[13] Giselle Reis, Zan Naeem & Mohammed Hashim (2020): *Sequoia: A Playground for Logicians - (System Description)*. In Nicolas Peltier & Viorica Sofronie-Stokkermans, editors: *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, *Lecture Notes in Computer Science* 12167, Springer, pp. 480–488, doi:10.1007/978-3-030-51054-1_32.

[14] Anders Schlichtkrull, Jørgen Villadsen & Andreas Halkjær From (2018): *Students' Proof Assistant (SPA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018*, *EPTCS* 290, pp. 1–13, doi:10.4204/EPTCS.290.1.

[15] Raymond M. Smullyan (1995): *First-Order Logic*. Dover Publications.

[16] Jørgen Villadsen (2020): *Tautology Checkers in Isabelle and Haskell*. In Francesco Calimeri, Simona Perri & Ester Zumpano, editors: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020*, *CEUR Workshop Proceedings* 2710, CEUR-WS.org, pp. 327–341. Available at `http://ceur-ws.org/Vol-2710/paper21.pdf`.

[17] Jørgen Villadsen, Andreas Halkjær From & Anders Schlichtkrull (2018): *Natural Deduction Assistant (NaDeA)*. In Pedro Quaresma & Walther Neuper, editors: *Proceedings 7th International Workshop on Theorem proving components for Educational software, ThEdu@FLoC 2018, Oxford, United Kingdom, 18 July 2018*, *EPTCS* 290, pp. 14–29, doi:10.4204/EPTCS.290.2.

[18] Makarius Wenzel (2007): *Isabelle/Isar—a generic framework for human-readable proof documents*. *From Insight to Proof—Festschrift in Honour of Andrzej Trybulec* 10(23), pp. 277–298.