MiniCalc Tutorial

Emmanuel André Ryom and Jørgen Villadsen

2024-02-12

1 Introduction

This tutorial explains how to use the MiniCalc web app to prove formulas in first-order logic and then formally verify the proofs in the Isabelle proof assistant. Please jump to section 2 if you would like to get started right away with a sample proof. In section 3 we explain how to formally verify proofs in Isabelle. In the present section we continue with a brief description of how to write formulas and proofs in MiniCalc.

To write formulas in MiniCalc, we combine logical connectives. All of these connectives are applied to two existing formulas except the negation operator, which is only applied to one, and the quantifiers, which are explained in section 4. A list of the connectives can be found below. We note that these connectives are more than enough to define any other connective one might want. Note also that the connectives are applied in the style of functional programming, i.e. `Imp A B` instead of `Imp(A, B)`.

	Usual syntax	MiniCalc syntax
Implication	$A \longrightarrow B$	`Imp A B`
Disjunction	$A \vee B$	`Dis A B`
Conjunction	$A \wedge B$	`Con A B`
Negation	$\neg A$	`Neg A`
Existential quantifier	$\exists A$	`Exi A`
Universal quantifier	$\forall A$	`Uni A`

The full syntax for proofs in MiniCalc is as follows.

```
⟨Proofs⟩    =  ⟨Proof⟩
            |  ⟨Proof⟩ "#" ⟨Proofs⟩
            |  ⟨Proof⟩ "-" ⟨Proofs⟩
⟨Proof⟩     =  ⟨Formula⟩ ⟨Steps⟩
⟨Steps⟩     =  ⟨Step⟩
            |  ⟨Step⟩ ⟨Steps⟩
⟨Step⟩      =  ⟨Rule⟩ ⟨Branches⟩
⟨Rule⟩      =  "Basic"
            |  "Imp_R" | "Imp_L"
            |  "Dis_R" | "Dis_L"
            |  "Con_R" | "Con_L"
            |  "Exi_R" | "Exi_L"
            |  "Uni_R" | "Uni_L"
            |  "Ext"
            |  "NegNeg"
⟨Branches⟩  =  ⟨Formulas⟩
            |  ⟨Formulas⟩ "+" ⟨Branches⟩

⟨Formulas⟩  =  ⟨Formula⟩
            |  ⟨Formula⟩ ⟨Formulas⟩
⟨Formula⟩   =  "Imp" ⟨Formula⟩ ⟨Formula⟩
            |  "Dis" ⟨Formula⟩ ⟨Formula⟩
            |  "Con" ⟨Formula⟩ ⟨Formula⟩
            |  "Neg" ⟨Formula⟩
            |  "Exi" ⟨Formula⟩
            |  "Uni" ⟨Formula⟩
            |  ⟨Atom⟩
⟨Atom⟩      =  ⟨Identifier⟩
            |  ⟨Identifier⟩ "[" ⟨Terms⟩ "]"
⟨Terms⟩     =  ⟨Term⟩
            |  ⟨Term⟩ "," ⟨Terms⟩
⟨Term⟩      =  ⟨Number⟩
            |  ⟨Identifier⟩
            |  ⟨Identifier⟩ "[" ⟨Terms⟩ "]"
```

⟨*Identifier*⟩ is a string used to identify a predicate or a function (see section 2).

⟨*Number*⟩ is a natural number $0, 1, 2, \ldots$ used to represent a variable (see section 4).

It is possible to write a one-line comment after "`#`" or "`-`" in MiniCalc proofs. These comments can be placed before, in between and after proofs and must be surrounded by quotation marks. Furthermore, it is also possible to write a multi-line comment delimited by "`(*`" and "`*)`" but such a comment does not get transferred to the Isabelle proof described in section 3.

2 Getting Started with MiniCalc

We are going to prove the following formula.

$$p(a,b) \lor \neg p(a,b) \tag{1}$$

Let us start by converting it to MiniCalc's syntax.

```
Dis p[a, b] (Neg p[a, b])
```

So let us see what rule can be applied here. Proofs in MiniCalc are driven using rules, of which an explanation can be found in section 6. Rule `Dis_R` can be applied, because the formula's outermost connective is a disjunction, and because it is not wrapped in a negation.

We can start our proof with that rule, as in the snippet below:

```
Dis p[a, b] (Neg p[a, b])

Dis_R
```

Now we need to see what the outcome is when you use the rule. For `Dis_R`, we simply unwrap the disjunction into two formulas.

```
  p[a, b]
  Neg p[a, b]
```

At this point we are very lucky to have two formulas in the sequent that complement each other. We can then finish the proof by using `Basic`.

```
Basic
```

We have thus proved formula 1. The complete proof is made of the snippets in the blue boxes put together as one string. Let us try to prove another formula.

$$(p \land q(f(a,g(b)))) \longrightarrow p \tag{2}$$

```
Imp (Con p q[f[a,g[b]]]) p

Imp_R
  Neg (Con p q[f[a,g[b]]])
  p
Con_L
  Neg p
  Neg q[f[a,g[b]]]
  p
```

Now we have a sequent with complementary formulas. We are close to finished with the proof, but we can only use `Basic` if the positive (non-negated) formula is the first in the sequent. We use `Ext` to change the order of the formulas and to remove unnecessary formulas. Then we finish the proof.

```
Ext
  p
  Neg p
Basic
```

3 Formal Verification of Proofs

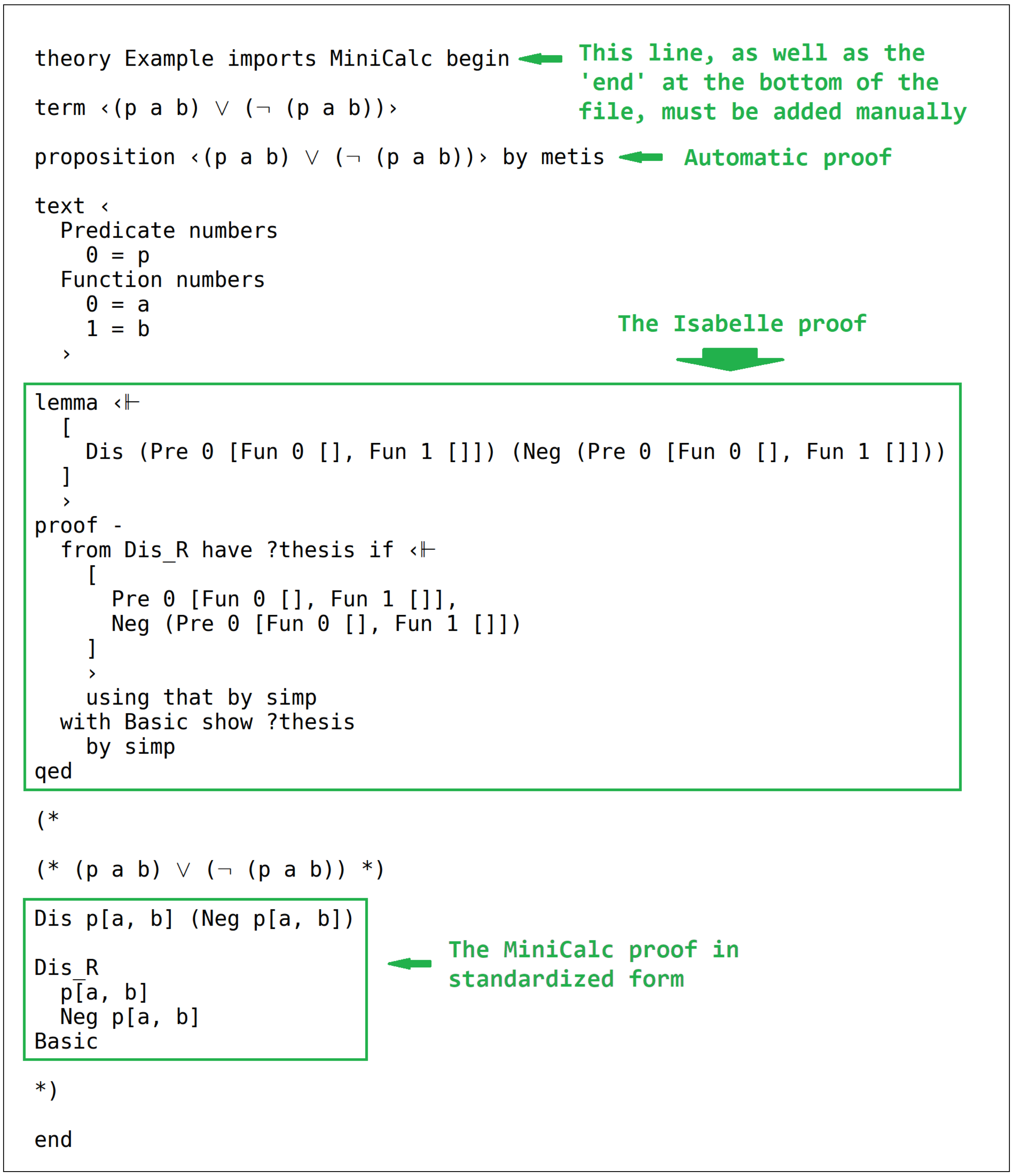
The MiniCalc web app you received in the zip folder does not guarantee that the proofs are correct. Therefore we need to use the Isabelle proof assistant, which offers formal verification. If your proof in Isabelle is without any red highlighting, then your proof is correct.

In the same directory as the `MiniCalc.thy` and `MainProof.thy` files, make a new `.thy` file. Press the "Copy Result to Clipboard" button in the web app, and paste the contents into Isabelle. Then, at the top of your file add the command "`theory Example imports MiniCalc begin`" (you should replace `Example` with the name of your file) and at the bottom add the command "`end`" (this ensures that Isabelle will check the very last proof too).

With the first example in this tutorial in mind, the text in Isabelle's IDE should be the same as on the figure below.

The result (received by clicking the button) always starts with a table of contents with every formula you tried to prove preceded by "`term`". Then for each proof you get both the Isabelle proof and a comment containing the MiniCalc proof in standardized form (which you can paste back into MiniCalc), as well as an automated proof which starts with "`proposition`" and ends with "`by metis`". This automated proof can work as a sanity check, to ensure that the formula you are trying to prove can be proven. Finally you also get a mapping for the predicate and function numbers.

Notice how the two types of proof relate to each other. MiniCalc has a much more concise way of writing the same proofs as Isabelle.


This line, as well as the 'end' at the bottom of the file, must be added manually
Automatic proof
The Isabelle proof
The MiniCalc proof in standardized form

```
theory Example imports MiniCalc begin

term ‹(p a b) ∨ (¬ (p a b))›

proposition ‹(p a b) ∨ (¬ (p a b))› by metis

text ‹
  Predicate numbers
    0 = p
  Function numbers
    0 = a
    1 = b
  ›

lemma ‹⊩
  [
    Dis (Pre 0 [Fun 0 [], Fun 1 []]) (Neg (Pre 0 [Fun 0 [], Fun 1 []]))
  ]
  ›
proof -
  from Dis_R have ?thesis if ‹⊩
    [
      Pre 0 [Fun 0 [], Fun 1 []],
      Neg (Pre 0 [Fun 0 [], Fun 1 []])
    ]
    ›
    using that by simp
  with Basic show ?thesis
    by simp
qed

(*

(* (p a b) ∨ (¬ (p a b)) *)

Dis p[a, b] (Neg p[a, b])

Dis_R
  p[a, b]
  Neg p[a, b]
Basic

*)

end
```

4 Proof of Formulas with Quantifiers

Let us bring it a step further by proving a formula with quantifiers.

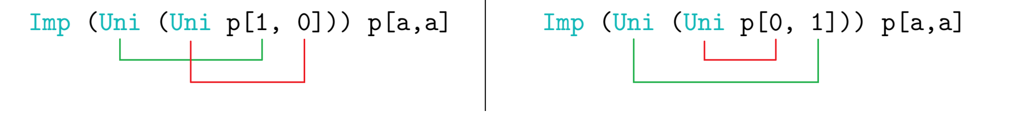
$$(\forall x.\forall y.p(x, y)) \longrightarrow p(a, a) \tag{3}$$

Again we should begin by converting it to MiniCalc's syntax, because this one has variables. In MiniCalc we have to convert the variables to de Bruijn indices.

```
Imp (Uni (Uni p[1, 0])) p[a, a]
```

This means that our two variables are represented by integers. To give some intuition on how the de Bruijn indices are determined, it is essentially the amount of quantifiers you have to 'cross' before you reach the quantifier the variable is bounded by.

Example formula $(\forall x.\forall y.p(x, y)) \longrightarrow p(a, a)$	*Alternate formula with swapped indices* $(\forall x.\forall y.p(y, x)) \longrightarrow p(a, a)$
`Imp (Uni (Uni p[1, 0])) p[a,a]`	`Imp (Uni (Uni p[0, 1])) p[a,a]`



The green and red lines indicate the relations between the variables and the quantifiers. We can see that the variable with the index 0 has its quantifier right next to it. If we swap the indices then the variables also swap quantifiers, and as such we end up with a completely different formula. Let us start the proof with the rule that can be applied here, `Imp_R`.

```
Imp (Uni (Uni p[1, 0])) p[a, a]

Imp_R
  Neg (Uni (Uni p[1, 0]))
  p[a, a]
```

We can see that we can use the `Uni_L` rule. Using this rule will substitute the variable, represented by the highest de Bruijn index, with a function of our choice.

```
Uni_L
  Neg (Uni p[a, 0])
  p[a, a]
```

Again, we can use the same rule, and this time the other variable will be substituted.

```
Uni_L
  Neg p[a, a]
  p[a, a]
```

Similarly to the previous example, we use `Ext` to change the order of the formulas, and then we can finish the proof.

```
Ext
  p[a, a]
  Neg p[a, a]
```

Now we have the formulas in the right order. We can finish the proof.

```
Basic
```

5 More on Proofs in MiniCalc

Some of the rules of MiniCalc require you to branch your proof into multiple sequents (see section 6). In that case, we use the "+" symbol to delimit the sequents from each other. Finally, let's prove an example which requires us to branch the proof. We are going to prove the following formula.

$$(\forall x.p(x) \longrightarrow q(x)) \longrightarrow ((\exists x.p(x)) \longrightarrow (\exists x.q(x))) \tag{4}$$

```
Imp (Uni (Imp p[0] q[0])) (Imp (Exi p[0]) (Exi q[0]))

Imp_R
 Neg (Uni (Imp p[0] q[0]))
 Imp (Exi p[0]) (Exi q[0])
Ext
 Imp (Exi p[0]) (Exi q[0])
 Neg (Uni (Imp p[0] q[0]))
Imp_R
 Neg (Exi p[0])
 Exi q[0]
 Neg (Uni (Imp p[0] q[0]))
Exi_L
 Neg p[a]
 Exi q[0]
 Neg (Uni (Imp p[0] q[0]))
Ext
 Neg (Uni (Imp p[0] q[0]))
 Neg p[a]
 Exi q[0]
Uni_L
 Neg (Imp p[a] q[a])
 Neg p[a]
 Exi q[0]
Imp_L
 p[a]
 Neg p[a]
 Exi q[0]
+
 Neg q[a]
 Neg p[a]
 Exi q[0]
```

At this point, we have branched the proof using `Imp_L`, and we see that we can finish part of the proof by using `Basic` as the first sequent is now tautological. The second sequent is not tautological yet, and therefore the proof continues for a bit. We can remove the tautological sequent already.

```
Basic
 Neg q[a]
 Neg p[a]
 Exi q[0]
Ext
 Exi q[0]
 Neg q[a]
Exi_R
 q[a]
 Neg q[a]
Basic
```

6 The Rules of MiniCalc

Here are the names of all the rules in MiniCalc.

Non-branching	*Branching*	*Quantifiers*	*Special*
Imp_R	Imp_L	Exi_R	Basic
Dis_R	Dis_L	Exi_L	Ext
Con_L	Con_R	Uni_R	NegNeg
		Uni_L	

The general understanding of the categories non-branching, branching, and quantifiers relies on the common patterns the rules share, meanwhile the special category has the three unique rules. The non-branching rules take the first formula in a sequent and alters it into two formulas. The branching rules alter the first formula in a sequent and makes two different sequents, hence they are branching. The quantifier rules remove a quantifier and replaces a variable with some function, on the first formula.

Finally, we have the special rules. NegNeg deals with double negations and Ext lets us shuffle, duplicate, and remove formulas in the sequent. Basic is the only rule that can finish a proof, and it does so by removing the tautological sequents.

The best way to get a thorough understanding of the rules of MiniCalc is to look into the files that make up the proof system. `MainProof.thy` has an almost comprehensive list of the rules, displayed below, as well as a formal proof of the soundness theorem (the file also defines other proof systems). In this context, the "#" symbol is meant to be interpreted as the addition of a formula to a list of formulas.

```
inductive sequent_calculus :: ‹fm list ⇒ bool› (‹⊩ _› 0) where
  Basic: ‹⊩ p # z› if ‹member (Neg p) z› |
  Imp_R: ‹⊩ Imp p q # z› if ‹⊩ Neg p # q # z› |
  Imp_L: ‹⊩ Neg (Imp p q) # z› if ‹⊩ p # z› and ‹⊩ Neg q # z› |
  Dis_R: ‹⊩ Dis p q # z› if ‹⊩ p # q # z› |
  Dis_L: ‹⊩ Neg (Dis p q) # z› if ‹⊩ Neg p # z› and ‹⊩ Neg q # z› |
  Con_R: ‹⊩ Con p q # z› if ‹⊩ p # z› and ‹⊩ q # z› |
  Con_L: ‹⊩ Neg (Con p q) # z› if ‹⊩ Neg p # Neg q # z› |
  Exi_R: ‹⊩ Exi p # z› if ‹⊩ subt t p # z› |
  Exi_L: ‹⊩ Neg (Exi p) # z› if ‹⊩ Neg (inst c p) # z› and ‹news c (p # z)› |
  Uni_R: ‹⊩ Uni p # z› if ‹⊩ inst c p # z› and ‹news c (p # z)› |
  Uni_L: ‹⊩ Neg (Uni p) # z› if ‹⊩ Neg (subt t p) # z› |
  Extra: ‹⊩ z› if ‹⊩ p # z› and ‹member p z›
```

There are a few functions in the definition of the sequent calculus. `member` which ensures that some element is a member of a list. `inst` essentially instantiates a variable into a function in some formula, and `news` ensures that the function is not previously seen anywhere else in a sequent. Then `subt` ensures that the some variable was substituted in the sequent. There is more to these functions than it seems at first, in particular to what variables are substituted, so the full definitions of these functions can be found in `MainProof.thy` (to navigate quickly to a definition, search the rule with ":" at the end).

NegNeg is a derived rule in `MainProof.thy` (we display only the rule and not the derivation of the rule).

```
theorem NegNeg: ‹⊩ Neg (Neg p) # z› if ‹⊩ p # z›
```

MiniCalc uses the derived rule Ext instead of the very restricted rule Extra. However, the derivation of Ext from the other rules is rather complicated. `MiniCalc.thy` has the details as well as a formal proof of the completeness theorem (which is the main purpose of the file).

Acknowledgements

We thank Oliver Emil Bøving, Agnes Moesgård Eschen, Frederik Krogsdal Jacobsen, Simon Tobias Lund and Anders Schlichtkrull for their very helpful comments.