Proof Assistants (L81): Assignment I

Thomas Bauereiss Leo Stefanesco

Lent 2026

This is the first of two marked assignments for the course "Proof Assistants" (L81), covering the Isabelle part of the course. This assignment comprises a number of small formalisation and verification tasks related to the imperative programming language IMP, as well as a short write-up explaining your work.

The due date for this assignment is Thursday, 5 March 2026, 4pm. You must work on this assignment as an individual; collaboration with others or the use of AI tools (other than the Sledgehammer tool built into Isabelle) is not allowed. Copying material found elsewhere counts as plagiarism. Please use the Moodle page for the course to submit an archive containing your theories and write-up by the deadline.

Each of the two assignments is worth 100 marks, distributed as follows:

- 50 marks for completing basic formalisation and verification tasks assessing grasp of the material taught in the lecture. For this assignment, please use the Isabelle theory file template provided on the course website. When preparing your submission, please include your `Assignment1.thy` file as well as any other theory files that you have edited. You are allowed to use Sledgehammer to find proofs, although you might want to streamline any complex proofs it finds into something more readable. The main assessment criteria for these tasks are correctness and completeness of the specifications and proofs. You might want to test your specifications to check that they work as intended.

- 20 marks for completing designated more challenging tasks, which might require more creativity when designing specifications and proof strategies, and might require some Isabelle techniques that go beyond what was taught in the lecture. Completing these advanced tasks can help you achieve a distinction grade, but you might want to focus first on completing the main tasks before attempting the advanced tasks.

- 30 marks for a clear write-up, where 10 of these marks will be reserved for write-ups of exceptional quality, e.g. demonstrating particular insight. In general, your write-up should explain the design decisions

you made during the formalisation and the strategies you used for the proofs. It might also discuss proof attempts that failed in interesting ways and lessons you learned from them, if that happens. The maximum length of the write-up is 2,500 words, although it could be much shorter, especially if you add comments to your theory files. It is possible to use Isabelle to generate a document,[1] but you can use any tool you prefer to prepare your write-up and add a PDF to your submission.

[1]The template files for the assignment are set up to generate the handout document with this introduction and the tasks. For more information on document preparation from Isabelle, see Chapter 4 of "The Isabelle/Isar Reference Manual" and Chapter 3 of "The Isabelle System Manual".

Assignment I: The IMP language in Isabelle

The following tasks are about writing/generating and verifying programs in the IMP language.

Note that the version of the IMP language defined in the files in this assignment template uses natural numbers as variable identifiers rather than strings, in order to simplify reasoning about variable names.

Task 1 (4 marks) Prove the following lemma about the preservation of invariants, which can be useful when proving properties about programs containing WHILE loops:

lemma *While_inv*:
  assumes $(WHILE\ b\ DO\ c,\ s) \Rightarrow t$
    and $\bigwedge s\ t.\ (c,\ s) \Rightarrow t \Longrightarrow bval\ b\ s \Longrightarrow Inv\ s \Longrightarrow Inv\ t$
    and $Inv\ s$
  shows $\neg bval\ b\ t \wedge Inv\ t$

Task 2 (15 marks) The basic arithmetic expression language *aexp* in our IMP language only supports addition, but we can still write programs that perform other arithmetic operations.

Write a program that subtracts the value of variable 1 from the value of variable 0 and stores the result in variable 2:

definition *minus_com* :: *com* **where**
  $minus\_com \equiv undefined$

You may use all commands of the IMP language including WHILE and IF commands, as well as additional variables.

Prove your program correct:

lemma
  $(minus\_com,\ s) \Rightarrow t \implies t\ 2 = s\ 0 - s\ 1$

Task 3 (3 marks) Now consider the following expression language that extends *aexp* with subtraction and multiplication:

datatype *aexp2* =
    N2 int
  | *V2 vname*
  | *Plus2 aexp2 aexp2*
  | *Minus2 aexp2 aexp2*
  | *Mult2 aexp2 aexp2*

Write an evaluation function and a function determining the variables appearing in an expression:

fun *aval2* :: *aexp2* ⇒ *state* ⇒ *int* **where**
  aval2 e s = *undefined*

fun *avars2* :: *aexp2* ⇒ *vname set* **where**
  avars2 e = *undefined*

Show the following congruence lemma, stating that evaluating an *aexp2* expression in two states agreeing on the values of variables appearing in the expression gives the same result, i.e. the value of an expression does not depend on variables that don't appear in it:

lemma *aval2_cong*:
  (∀ *v*. *v* ∈ *avars2 e* ⟶ *s v* = *t v*) ⟹ *aval2 e s* = *aval2 e t*

Task 4 (10 marks) Write a compiler *com_of_aexp2* from an *aexp2* expression to a program in the original IMP language (possibly using loops and conditionals for evaluating sub-expressions not supported there, as well as additional variables for intermediate values), where the program *com_of_aexp2 res e* writes the result of evaluating *e* to variable *res*:

fun *com_of_aexp2* :: *vname* ⇒ *aexp2* ⇒ *com* **where**
  com_of_aexp2 res e = *undefined*

Test your compiler using some example expressions.

Explain informally in your write-up how, why, and under what conditions the compiler works.

Task 5 (18 marks) Prove (partial) correctness of the programs generated by *com_of_aexp2*. You may add assumptions to the lemma capturing the conditions under which your compiler works, as explained above.

lemma *com_of_aexp2_correct*:
  (*com_of_aexp2 res e*, *s*) ⇒ *t* ⟹ *t res* = *aval2 e s*

Task 6 (20 marks (advanced)) Prove the termination (and therefore total correctness) of programs generated by *com_of_aexp2* (again possibly adding assumptions as in the previous task):

lemma *com_of_aexp2_terminates*:
  ∃ *t*. (*com_of_aexp2 res e*, *s*) ⇒ *t*

lemma *com_of_aexp2_correct_total*:
  ∃ *t*. (*com_of_aexp2 res e*, *s*) ⇒ *t* ∧ *t res* = *aval2 e s*