Proof Assistants (L81): Assignment I

Thomas Bauereiss Leo Stefanesco

Michaelmas 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 Monday, 16 November 2026, 12 noon. Please use the Moodle page for the course to submit an archive containing your theories and write-up by the deadline.

You must work on this assignment as an individual; collaboration with others is not allowed. The use of AI tools (other than the Sledgehammer tool built into Isabelle) is not permitted for writing the formalisation, proofs, or write-up. Asking an AI tool to explain concepts from the course is acceptable, but be aware that, depending on the question, it might go beyond an explanation and produce (part of) a solution, which you must not use. If in doubt, please ask us instead; we are happy to answer questions about the material and the assignment. Copying material found elsewhere counts as plagiarism. We may schedule vivas after the submission deadline, in which we discuss your submission with you in person to confirm that the work is your own and that the rules above were followed.

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 lectures. 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. Also try to avoid long apply-style proofs, and try to clean them up into structured Isar proofs. You might want to test your specifications to check that they work as intended. The main assessment criteria for these tasks are correctness and completeness of the specifications and proofs, although incomplete proofs may still

earn partial marks. Hence, if you cannot complete a task, please still submit your partial proofs, e.g. with "sorry" holes, and explain in the write-up how far you got, where you got stuck, and what is still left to do.

- 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 lectures. 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. Again, partial proofs here may still earn partial marks.

- 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

This assignment is about adding *FOR* loops to the IMP language and reasoning about them.

1 Semantics

Add a constructor for loops *FOR x FROM lo TO hi DO c* to *com* (where *x* is a *vname*, *lo* and *hi* are *aexp* expressions for the (inclusive) bounds, and *c* is a *com* command).

You can copy the syntax annotation from the following placeholder. Delete or comment out the placeholder after you have added the constructor.

definition *For* :: *vname* ⇒ *aexp* ⇒ *aexp* ⇒ *com* ⇒ *com*
(‹(*FOR* _/ *FROM* _/ *TO* _/ *DO* _)› [*0*, *0*, *0*, *61*] *61*) **where**
For x lo hi c ≡ *undefined*

The tasks below are about defining a semantics for this kind of loop, proving some properties, e.g. about unrolling one iteration, and finally defining an elaboration of these *FOR* loops into *WHILE* loops and proving that it preserves the behaviour.

For defining the semantics, there are some design decisions to be made here, e.g. whether the bounds are evaluated once at the start or re-evaluated at every loop iteration. Before defining your semantics, read through the other tasks in this assignment, and think about which design choices are required here, e.g. to align with the command equivalences in Section 3, and which can make tasks easier, e.g. the proofs about the elaboration in Section 6. For example, if your elaboration into *WHILE* loops evaluates the lower bound at the start, and re-evaluates the upper bound in each iteration, you might want to mirror that in your semantics of the *FOR* loop.

Feel free to make choices that are convenient for the tasks below. Discuss your choices and their implications in your write-up.

When you have designed your semantics, formalise it by adding rules to the big-step semantics *big_step*.

Test your semantics with a few test programs using the *values* command (which should still work due to the *code_pred* setup in the *Big_Step* theory).

Note that the code generated by *code_pred* can be sensitive to how the premises of your rules are phrased, even if the phrasings are logically equivalent. If *values* does not terminate even for simple test programs, try rephrasing the premises, e.g. comparing with an evaluated expression directly instead of first binding its value to an auxiliary variable.

As one example, consider the following program calculating Fibonacci numbers:

definition *fib_body* :: *com* **where**
fib_body ≡
(''*temp*'' ::= *Plus* (*V* ''*a*'') (*V* ''*b*'');;
''*a*'' ::= *V* ''*b*'';;
''*b*'' ::= *V* ''*temp*'')

definition *fib_com* :: *nat* ⇒ *com* **where**
fib_com n ≡
(''*a*'' ::= *N 0*;;
''*b*'' ::= *N 1*;;
FOR ''*i*'' *FROM* (*N 1*) *TO* (*N* (*int n*)) *DO fib_body*)

With a command like the following you can test what output the program produces and whether it agrees with the *fib* function from the standard library:

values {(*t* ''*a*'', *t* ''*a*'' = *int* (*fib 10*)) | *t*. (*fib_com 10*, <>) ⇒ *t*}

Add that test to the theory once your semantics is in place, and add a few small example programs of your own to test corner cases of the semantics.

2 Determinism

Repair the existing proofs in the *Big_Step* theory, in particular the determinism of the big-step semantics.

3 Command equivalences

Prove the following command equivalences.

lemma *for_empty*:
hi < *lo* ⟹ (*FOR x FROM* (*N lo*) *TO* (*N hi*) *DO c*) ∼ (*x* ::= *N lo*)

lemma *for_single*:
(*FOR x FROM* (*N n*) *TO* (*N n*) *DO c*) ∼ (*x* ::= *N n*;; *c*;; *x* ::= *N* (*n* + *1*))

lemma *for_unfold*:
(*FOR x FROM* (*N lo*) *TO* (*N hi*) *DO c*) ∼
(*IF Less* (*N hi*) (*N lo*)
THEN x ::= *N lo*
ELSE (*x* ::= *N lo*;; *c*;; *FOR x FROM* (*N* (*lo* + *1*)) *TO* (*N hi*) *DO c*))

lemma *for_cong*:
assumes ∀ *s*. *aval lo s* = *aval lo*′ *s* **and** ∀ *s*. *aval hi s* = *aval hi*′ *s*
and *c* ∼ *c*′
shows (*FOR x FROM lo TO hi DO c*) ∼ (*FOR x FROM lo*′ *TO hi*′ *DO c*′)

4 Elaboration of *FOR* loops to plain IMP

Define an elaboration function *elab_for* of type *com* ⇒ *com* that rewrites a command into one without *FOR* loops, using *WHILE* loops instead. The result should be equivalent to the original command, at least for suitable programs (see Section 6).

fun *elab_for* :: *com* ⇒ *com* **where**
  elab_for c = *undefined*

Test the function with a few example programs using the *value* command.

Then, prove that your elaboration actually removes *FOR* loops, using a function *has_for* that checks for the presence of *FOR* loops:

fun *has_for* :: *com* ⇒ *bool* **where**
  has_for c = *undefined*

lemma *has_for_elab_for*:
  ¬*has_for* (*elab_for c*)

5 Auxiliary functions

When stating and proving correctness properties of the elaboration below, you might want to talk about the variables accessed in commands and expressions. Define a function *com_writes* that determines the variables that a command writes to, and prove it correct:

fun *com_writes* :: *com* ⇒ *vname set* **where**
  com_writes c = *undefined*

lemma *non_com_writes_unchanged*:
  (*c*, *s*) ⇒ *t* ⟹ *x* ∉ *com_writes c* ⟹ *t x* = *s x*

Similarly, define a function *avars* that determines the variables in an arithmetic expression, and prove:

fun *avars* :: *aexp* ⇒ *vname set* **where**
  avars e = *undefined*

lemma *avars_update_other*:
  x ∉ *avars e* ⟹ *aval e* (*s*(*x* := *y*)) = *aval e s*

6 Elaboration soundness

Now prove a soundness result for your elaboration of suitable programs; it will likely not generally hold for all programs.

In particular, you may assume in this task that the bodies of *FOR* loops in the original program do not write to the loop variable.

You might also want to make further assumptions, e.g. on what variables may appear in the bounds.

Define a function *for_ok* that checks those constraints, using the functions *com_writes* and *avars* defined above:

fun *for_ok* :: *com* ⇒ *bool* **where**
  for_ok c = *undefined*

Be careful not to make your constraints vacuous or overly restrictive; they should still allow a wide range of programs. In particular, the Fibonacci program *fib_com* above should be supported:

lemma *fib_com_for_ok*:
  for_ok (*fib_com n*)

Restricting the bounds to constants is acceptable as a first step, but more general constraints, e.g. allowing some variables in the bounds, will earn more marks.

Explain your choices for the constraints and their implications in the write-up.

Then prove soundness of the elaboration for programs satisfying your constraints (and the Fibonacci program in particular):

lemma *elab_for_sound*:
  (*c*, *s*) ⇒ *t* ⟹ *for_ok c* ⟹ (*elab_for c*, *s*) ⇒ *t*

corollary *elab_for_fib_com_sound*:
  (*fib_com n*, *s*) ⇒ *t* ⟹ (*elab_for* (*fib_com n*), *s*) ⇒ *t*

7 Elaboration completeness (advanced)

Finally, prove the converse of *elab_for_sound*, and hence full equivalence:

lemma *elab_for_complete*:
  (*elab_for c*, *s*) ⇒ *t* ⟹ *for_ok c* ⟹ (*c*, *s*) ⇒ *t*

corollary *elab_for_equiv*:
  for_ok c ⟹ *elab_for c* ∼ *c*

corollary *elab_for_fib_com_equiv*:
  elab_for (*fib_com n*) ∼ *fib_com n*

Hint: You might want to prove a lemma about the *WHILE* loops in your elaboration, and think about how to set up the induction for *elab_for_complete*.