# Interactive Proof: Applications to Semantics

Gerwin KLEIN
*NICTA and The University of New South Wales, Australia*

**Abstract.** Building on a previous lecture in the summer school, the introduction to interactive proof, this lecture demonstrates a specific application of interactive proof assistants: the semantics of programming languages. In particular, I show how to formalise a small imperative programming language in the theorem prover Isabelle/HOL, how to define its semantics in different variations, and how to prove properties about the language in the theorem prover.

The emphasis of the lecture is not on formalising a complex language deeply, but to teach formalisation techniques and proof strategies using simple examples. To this purpose, we cover big- and small step semantics, typing and type safety, as well as a small machine language with compiler and compiler correctness proof.

## Introduction

The purpose of these lecture notes is to demonstrate a specific application of interactive proof assistants: the semantics of programming languages. The topics cover how to formalise a small imperative programming language, IMP, in the theorem prover Isabelle/HOL, how to define its semantics in different variations, and how to prove properties about the language in the theorem prover. In particular, the material includes big- and small step semantics, typing and type safety, as well as a small machine language with compiler and compiler correctness proof.

The emphasis of the lecture is not on formalising a complex language deeply, but to teach formalisation techniques and proof strategies using simple examples.

The notation is standard Isabelle notation as introduced in the previous lecture of the same summer school on interactive proof. If we use new notation for concepts from the semantics of programming languages, we will introduce this notation when it first occurs, together with how it is declared and defined in the theorem prover.

Formulas in text are typeset as follows: `x = y`. They are produced by Isabelle LaTeX antiquotations such as `@{term "x = y"}`. A typical definition is typeset as

```
definition successor :: "nat ⇒ nat" where "successor n ≡ n+1"
```

and a typical lemma as follows:

```
lemma successor_is_Suc: "successor n = Suc n"
```

In the remainder of this document, we begin the exposition by choosing a small programming language, by formalising its abstract syntax, and by defining the semantics of its arithmetic and boolean expressions in section 1.

In the following section 2 we define the big-step semantics of the language in Isabelle/HOL, use it to prove that the language is deterministic, and show how to prove semantic equivalence of small program fragments. On this basis, we further introduce an alternative small-step semantics of the language that allows us to more easily talk about termination and that in a more elaborate setting could be used to express properties about concurrent execution. We show how to prove equivalence to the big-step semantics.

Based on the small step semantics, section 3 introduces a formalisation of types and typing rules for the language in Isabelle. We formulate a type soundness theorem and prove that the language is type safe. As an additional application, we present a security type system that is designed to prevent unwanted information flows within a program and show its correctness by proving that is enforces a high-level non-interference property.

The final part, section 4, changes abstraction levels and introduces the formalisation of a small, simplified machine language, including its execution semantics. Again, the purpose is not to accurately model an existing machine, but to introduce a number of basic concepts. This formalisation then allows us to define a very simple, non-optimising compiler for our previous language and prove its correctness.

## 1. The Language

This section shows how to define the abstract syntax of the language IMP and defines the semantics of expressions.

The constructs we need in IMP for a Turing complete imperative language are variable assignment, sequential composition (semicolon), conditionals (if-then-else), and loops (while-do). We will also include the skip command that does nothing. The right-hand side of variable assignments will have some form of arithmetic expressions, and the conditions in if-then-else and while loops will take boolean expressions. A program is simply one, possibly complex, command in this language.

Again, to keep things simple, the values our language IMP computes on will be mathematical integers and booleans only.

With this in mind, we can now formalise IMP in Isabelle/HOL. We are seeking to define the set of all possible abstract syntax trees. To do this, we use a data type definition in Isabelle with one constructor for each syntactic alternative. We will use the types *com* for commands, *aexp* for arithmetic expressions, and *bexp* for boolean expressions.

The section below shows the corresponding Isabelle definitions.

Both, arithmetic and boolean expressions, mention variable names (as does the assignment statement of com) for which we pick the type *string*. Any other type would work as well.

Further to variable names, we will want arithmetic expressions to be able to mention literal values such as numbers. For a more complex language we would invent a datatype with the different value types the language supports. Here, we again keep things simple and pick *int* as the only kind of value arithmetic expressions can return.

The first two type definitions below are the corresponding Isabelle commands for variable names and values.

## *1.1. Syntax of Expressions and Commands*

```
type_synonym  val  = int
type_synonym  name = string

datatype aexp = N val | V name | Plus aexp aexp

datatype bexp =  B bool | Not bexp | And bexp bexp | Less aexp aexp

datatype
  com = SKIP
      | Assign name aexp      ("_ ::= _" [1000, 61] 61)
      | Semi   com  com       ("_; _"  [60, 61] 60)
      | If     bexp com com   ("IF _ THEN _ ELSE _"  [0, 0, 61] 61)
      | While  bexp com       ("WHILE _ DO _"  [0, 61] 61)
```

Next to introducing the abstract syntax, the definition above also adds concrete infix syntax for Isabelle to parse and print such that we can write terms of the language more conveniently. The term "*If b* $c_1$ $c_2$" for instance will be printed as *IF b THEN* $c_1$ *ELSE* $c_2$.

## *1.2. Expression Semantics*

With the structure of commands and expressions defined, we can make our first definitions of semantic concepts: the meaning of expressions. Looking over the structure of arithmetic expressions in IMP, the meaning of an expression is simply an integer — the value of the expression. There is only one kind of expression that we cannot determine directly: variables. The value of a variable depends on the current state of the program. In IMP these are the only thing that we need to keep track of in the program state.

We begin by defining an Isabelle type for this state, a function from variable `name` to `val`. In more complex languages, this state space may grow to include a distinction between local variables, global variables, and a memory heap, for instance.

```
type_synonym state = "name ⇒ val"
```

We will write a denotational style semantics for expressions: a function that associates one semantic object with each syntactic entity. In our example, this means taking an arithmetic expression and returning a function from program state to value.

Going through the abstract syntax of arithmetic expressions, this function is easy to write recursively: numbers `N n` are mapped to the function that always returns `n`, variable lookup `V x` applies the variable name to the state, the addition syntax `Plus e1 e2` maps to the function that adds the semantics of `e1` applied to the state to the semantics of `e2` in the same state. The other operators follow the same pattern.

The definition in Isabelle is the following.

```
primrec aval :: "aexp ⇒ state ⇒ val" where
"aval (N n) _           = n" |
"aval (V x) st          = st x" |
"aval (Plus e1 e2) st  = aval e1 st + aval e2 st"
```

When writing such definitions, it is not uncommon to get them slightly wrong on the first go. It pays off to play with them a little before proceeding to any serious work.

The Isabelle command `value` is a great tool for evaluating sample terms. Below, we evaluate the expression that might be written as `3 + x` in concrete syntax on an example state that maps alls variables to 0, apart from *x* which will be `7`.

```
value "aval (Plus (V ''x'') (N 5)) (%x. if x = ''x'' then 7 else 0)"
```

For large states, we introduce special notation and write terms such as `[''x'' → 7, ''y'' → 5]` to mean that variable *x* maps to `7`, *y* to `5` and all others to `0`. Our evaluation example now becomes shorter and more readable.

```
value "aval (Plus (V ''x'') (N 5)) [''y'' → 7]"
```

Isabelle evaluates this to the expected `5`.

After this syntax side tour, we return to defining the semantics of expressions. The definition for boolean expressions is largely analogous to the arithmetic expression case. The IMP language does not have boolean-valued variables, so the variable lookup equation is missing. Instead we have the comparison operator *Less* with a different syntactic category, *aexp*, as parameters. We simply use our arithmetic expression semantics from above to evaluate and combine these.

```
primrec bval :: "bexp ⇒ state ⇒ bool" where
"bval (B b) _          = b" |
"bval (Not b) st       = (¬ bval b st)" |
"bval (And b1 b2) st   = (bval b1 st ∧ bval b2 st)" |
"bval (Less a₁ a₂) st = (aval a₁ st < aval a₂ st)"
```

As before, we briefly try out our definition using the `value` command in the same state and get the expected result *True*.

```
value "bval (Less (V ''y'') (Plus (N 3) (V ''x''))) [''x''→1,''y''→2]"
```

## *1.3. Optimisation: Simplifying Expressions*

Even though we have not even defined the semantics for the whole language yet, we can already prove some first facts about expressions.

As an example, consider the the optimisation *constant folding* that many compiler perform before translating a program to simplify expressions. The idea is to evaluate the constant part of expressions as far as possible at compile time instead of at runtime. The justification is that the values of constant expressions do not depend on the program state. It therefore does not matter when they are evaluated.

We can define constant folding on arithmetic expressions by recursing over the abstract syntax. The interesting cases are binary operators such as `Plus`. For these, we recursively perform constant folding on their operands and then make a case distinction over the result: if both sides fold into constants, we return their sum as a constant result, otherwise, we return a new `Plus` with their partially folded operands. In Isabelle:

```
primrec simp_const :: "aexp ⇒ aexp" where
"simp_const (N n)          = N n" |
"simp_const (V x)          = V x" |
"simp_const (Plus e1 e2)  = (case (simp_const e1, simp_const e2) of
    (N n1, N n2) ⇒ N(n1+n2) | (e1',e2') ⇒ Plus e1' e2')"
```

The correctness criterion for this function is that the folded/simplified expression should have the same semantics as the original expression. The point-wise formulation of this statement, both sides applied to the same arbitrary state, is proved by Isabelle with a single line of input.

```
theorem aval_simp_const[simp]: "aval (simp_const a) st = aval a st"
```

The proof follows a general pattern for statements about primitive recursive functions. We start with induction of the parameter that the function recurses over. This case is particularly nice, because both functions `simp_const` and `aval` follow the same recursion pattern. The `auto` proof method in the second part performs a combination of term rewriting and classical logic reasoning on all subgoals that are left by the induction (one for each constructor). The parameter `split: aexp.split` tells `auto` to automatically perform case distinction on case constructs of type `aexp`. We add it, because the `simp_const` contains such case constructs and the proof will have to go through these cases to make progress. Isabelle comes with enough proof setup about arithmetic to complete the rest of the proof automatically.

By declaring the lemma `[simp]`, we directly set up automation for further proofs involving the term pattern `aval (simp_const a) st`, telling the Isabelle simplifier to replace it with the simpler right-hand side of the equation.

A good automation setup is essential for keeping proofs short and manageable. Theorems that are obviously useful as simplification rules should be declared as such. Overusing this feature can quickly lead to frustrating non-terminating automatic proof methods, though, so care is needed.

In a larger development it pays off to come back to the basic definitions and design appropriate normal forms for combinations of functions, and to design sets of simplification rules for reaching these normal forms automatically.

This concludes the section on the syntax of IMP and the semantics of expressions. As is usual in the formal treatment of programming languages, we have focussed on the abstract syntax of the language instead of the concrete syntax. We have used the expression semantics for evaluating sample expressions and for proving the correctness of a simple constant folding algorithm for expressions.

We also definitions of recursive functions together with a common proof schema for simple properties about them. The proofs so far have turned out very simple and automatic. This was because the properties we were interested aligned nicely with the corresponding definitions and induction principles.

All of the definitions so far have been executable operations on executable data types. In principle, we could therefore automatically generate SML, OCaml, or Haskell code from them if we intended to use or validate the semantics in an external tool.

## 2. Semantics

This section shows two ways to define the semantics of commands in IMP: *big-step* and *small-step* operational semantics. Big-step semantics is also called *natural semantics* and small-step may occur under the name *structural operational semantics* (SOS).

Big-step semantics is simpler to define, with fewer rules and cases to consider in proofs, but it makes it hard to distinguish issues such as termination from undefined execution. Small-step semantics will result in more rules and higher proof effort for some properties, but its finer-grained structure allows us to talk about termination more naturally, and to define the interleaved execution of programs in a concurrent setup.

In the following, we first define the big-step semantics for IMP, use it to prove small language properties such as deterministic execution, define an explicit termination relation, define the small-step semantics for IMP, and finally prove that the two semantics definitions are equivalent. This means, for further properties, we are free to chose whichever representation is more convenient for the proof at hand.

### *2.1. Big-Step Semantics of Commands*

#### *2.1.1. The Definition*

In an operational semantics setting, the semantics of a program is captured as a relation. In big-step semantics, the relation is between program, initial state, and final state. Intermediate states during the execution of the program are not visible in the formal construct, the whole program is executed in one big step.

To formalise this concept in the theorem prover, we recall that a relation is simply a set of tuples, and to say that a command `c` executes in state `s` to some state `s'` is to say that the tripe `(c, s, s')` is an element of this set that defines the semantics of IMP.

Textbook-style semantics formalisations usually introduce concrete syntax to write such judgements in a more intuitive form. We use Isabelle's syntax mechanism to do the same here and will write `(c,s) ⇒ s'` for `(c, s, s')` is in the `big_step` relation.

We will use an inductive definition to define which triples this relation is made up of. For each syntactic construct we will give a number of rules that enumerate the possible executions for the construct in pattern matching style.

```
inductive
  big_step :: "com × state ⇒ state ⇒ bool" (infix "⇒" 55)
where
  Skip:    "(SKIP,s) ⇒ s"
| Assign:  "(x ::= a,s) ⇒ s(x := aval a s)"

| Semi:    "(c1,s1) ⇒ s2  ⟹  (c2,s2) ⇒ s3  ⟹
            (c1;c2, s1) ⇒ s3"
```

```
| IfTrue:  "bval b s  ⟹  (c1,s) ⇒ s'  ⟹
            (IF b THEN c1 ELSE c2, s) ⇒ s'"
| IfFalse: "¬bval b s  ⟹  (c2,s) ⇒ s'  ⟹
            (IF b THEN c1 ELSE c2, s) ⇒ s'"

| WhileFalse: "¬bval b s ⟹ (WHILE b DO c,s) ⇒ s"
| WhileTrue:  "bval b s1  ⟹  (c,s1) ⇒ s2  ⟹  (WHILE b DO c, s2) ⇒ s3
               ⟹ (WHILE b DO c, s1) ⇒ s3"
```

This is the full definition of the big-step semantics for IMP.

The rules given in the definition are the introduction rules for the inductive definition `big_step`. Note that instead of a set, we defined a predicate instead which is isomorphic to the intended set, but slightly more convenient to use. We use a tuple as the left side of our `(_,_) ⇒ _` syntax and only declare the arrow ⇒ as new.

Going through each of the named rules, they admit the following executions in IMP.

- If the command is `SKIP`, the initial and final state must be the same.
- If the command is a variable assignment `x ::= a`, and the initial state is `s`, then the final state is the same state `s` where the value of variable `x` is replaced by the evaluation of the expression `a` in state `s`.
- If the command is a sequential composition, rule `Semi` says the combined command `c1; c2` started in `s1` executes to `s3` if the the first command executes in `s1` to some intermediate state `s2` and `c2` takes this `s2` to `s3`.
- The conditional is the first command that has two rules, depending on the value of its boolean expression in the current state `s`. If that value is `True`, then the `IfTrue` rule says that the execution ends in the same state `s'` that the command `c1` results in if started in `s`. The `IfFalse` rule does the same for the command `c2` in the `False` case.
- While loops are slightly more interesting. If the condition evaluates to false, the whole loop is skipped, which is expressed in rule `WhileFalse`. If the condition evaluations in state `s1` to `True`, however, and the body `c` of the loop takes this state `s1` to some intermediate state `s3`, *and* if the same while loop, started in `s2` ends in `s3`, then the entire loop also terminates in `s3` if started in `s1`.

The rules of the inductive definition above can be directly used as introduction rules in proofs. However, a better way to execute the big-step rules is to use Isabelle's code generator. The following command tells it to generate code for the predicate `op ⇒` and make thus make the predicate available in the `values` command.

```
code_pred big_step .
```

Functions in general cannot easily be printed, so we need to convert the interesting part of the state to something printable such as a list to make use of the code generator here:

```
values "{map t [''x'', ''y'']|t. (''x'' ::= N 2, λ_. 0) ⇒ t}"
```

This has the result `{[2,0]}`.

In semantics text books, inductive definitions are often presented in graphical proof rule form. If the correct package is included, Isabelle antiquotations can produce the same layout in LaTeX. For example, the antiquotation `@{thm [mode=Rule] Semi}` produces the following layout:

$$\frac{\textit{(c1, s1)} \Rightarrow \textit{s2} \qquad \textit{(c2, s2)} \Rightarrow \textit{s3}}{\textit{(c1; c2, s1)} \Rightarrow \textit{s3}}$$

Finding or designing the right set of introduction rules for a language is not necessarily hard. The idea is to have at least one rule per syntactic construct and to add further rules when case distinctions become necessary. For each single rule, you start with the conclusion, for instance *(c1; c2, s) ⇒ s'*, and then construct the assumptions of the rule by thinking about which conditions have to be true about *s*, *s'*, and the parameters of the abstract syntax constructor, in the example *c1* and *c2*, for the conclusion to be true. If the assumptions collapse to an equation about *s'* as in the *SKIP* and *x ::= a* case, *s'* can be replaced directly.

Introduction rules are often written for readability and easy intuition, not necessarily for the most efficient automatic proof setup. It often pays off to derive further rules that aid automation in the rest of the formalisation. We do this for the big step rules of IMP below.

## *2.1.2. Proof and Automation Setup*

The most important proof tool for inductive definition is rule induction. After splitting pairs into their components, we get the folllowing rule for our definition:

```
⟦(c, s) ⇒ s'; ⋀s. P SKIP s s; ⋀x a s. P (x ::= a) s (s(x → aval a s));
 ⋀c1 s1 s2 c2 s3.
    ⟦(c1, s1) ⇒ s2; P c1 s1 s2; (c2, s2) ⇒ s3; P c2 s2 s3⟧
    ⟹ P (c1; c2) s1 s3;
 ⋀b s c1 s' c2.
    ⟦bval b s; (c1, s) ⇒ s'; P c1 s s'⟧
    ⟹ P (IF b THEN c1 ELSE c2) s s';
 ⋀b s c2 s' c1.
    ⟦¬ bval b s; (c2, s) ⇒ s'; P c2 s s'⟧
    ⟹ P (IF b THEN c1 ELSE c2) s s';
 ⋀b s c. ¬ bval b s ⟹ P (WHILE b DO c) s s;
 ⋀b s1 c s2 s3.
    ⟦bval b s1; (c, s1) ⇒ s2; P c s1 s2; (WHILE b DO c, s2) ⇒ s3;
     P (WHILE b DO c) s2 s3⟧
    ⟹ P (WHILE b DO c) s1 s3⟧
⟹ P c s s'
```

It reads: if we know *(c, s) ⇒ s'*, then proving any property *P* about *c*, *s*, and *s'*, can be reduced to proving the same property about each inductive case. The rule *Skip* and *Assign* provide the base cases, the step cases are allowed to assume *P* for their inductive preconditions.

The second common proof principle for inductive definitions are the introduction rules themselves. We tell Isabelle to use all of them as logic introduction rules. For the recursive rules this does risk non-termination and may require backtracking, so we declare them as unsafe using `[intro]` instead of the more insistent `[intro!]`.

**declare** *big_step.intros [intro]*

The rules of `big_step` are syntax directed, i.e. each of the syntactic categories is covered by its own distinct set of rules. This means we can easily use the rules in both directions. This technique is called rule inversion. We can tell Isabelle to automatically generate inversion rules for each particular conclusion pattern using the `inductive_cases` command. All of these rules are safe elimination rules for the classical logic reasoner, so we declare them as such and enable it to perform rule inversion for these patterns automatically.

```
inductive_cases skipE [elim!]:   "(SKIP,s) ⇒ s'"
inductive_cases assignE [elim!]: "(x ::= a,s) ⇒ s'"
inductive_cases semiE [elim!]:   "(c0; c1, s) ⇒ s'"
inductive_cases ifE [elim!]:     "(IF b THEN c0 ELSE c1, s) ⇒ s'"
inductive_cases whileE [elim]:   "(WHILE b DO c,s) ⇒ s'"
```

The rule for assignment for instance says that if the pattern `(x ::= a, s) ⇒ s'` occurs in an assumption, then by rule inversion `s'` must be `s(x → aval a s)`:

```
⟦(x ::= a, s) ⇒ s'; s' = s(x → aval a s) ⟹ P⟧ ⟹ P
```

The rule `semiE` shows a recursive case:

```
⟦(c0; c1, s) ⇒ s'; ⋀s2. ⟦(c0, s) ⇒ s2; (c1, s2) ⇒ s'⟧ ⟹ P⟧ ⟹ P
```

Using these elimation rules, we can additionally prove equations for use with Isabelle's simplifier can use. All of these proofs are already automatic, but they make essential use of both the elimination rules and also the potentially unsafe introduction rules. What we have achieved is that the simplifier can now do these steps on its own when just invoked with `simp`. This is gives more fine-grained control and is therefore preferable to the `auto` method that often tries too hard and to does too much.

This concludes our automation setup of the big step semantics. We have tuned the induction principle for the expected application, we have provided automated introduction rules, elimination rules, and finally simplification rules for common cases.

### *2.1.3. Equivalence of statements*

In this section we apply our setup to first define semantic equivalence of commands and then show a number of simple instances of equivalent statements.

We call two statements `c` and `c'` equivalent wrt. the big-step semantics when `c` *started in* `s` *terminates in* `s'` *iff* `c'` *started in the same* `s` *also terminates in the same* `s'`. Formally:

```
definition
  equiv_c :: "com ⇒ com ⇒ bool" (infix "∼" 50) where
  "c ∼ c' = (∀s s'. (c, s) ⇒ s' ⟷ (c', s) ⇒ s')"
```

Again, we provide proof rules for reasoning about equivalence automatically. The introduction rule is safe, so we declare it as such, the destruction rules depend on which direction to take, so we will supply them manually when needed.

```
lemma equivI [intro!]: "(⋀s s'. (c, s) ⇒ s' = (c', s) ⇒ s') ⟹ c ∼ c'"
lemma equivD1: "c ∼ c' ⟹ (c, s) ⇒ s' ⟹ (c', s) ⇒ s'"
lemma equivD2: "c ∼ c' ⟹ (c', s) ⇒ s' ⟹ (c, s) ⇒ s'"
```

Experimenting with this concept, we see that Isabelle manages to prove many simple equivalences automatically. Such rules could be used for instance to transform source-level programs in a compiler optimisation phase.

```
lemma while_unfold:
  "(WHILE b DO c) ∼ (IF b THEN c; WHILE b DO c ELSE SKIP)"
lemma triv_if: "(IF b THEN c ELSE c) ∼ c"
lemma commute_if:
  "(IF b2 THEN (IF b1 THEN c11 ELSE c2) ELSE (IF b1 THEN c12 ELSE c2))
   ∼
   (IF b1 THEN (IF b2 THEN c11 ELSE c12) ELSE c2)"
```

Note that we have neglected to provide an analogous equivalence definition for expressions. If we intended to reason more deeply about program equivalence, we would need to provide these and we would allow rewriting in expressions as well.

In IMP, loops that are trivially non-terminating, because their loop condition is always true, are indistinguishable in the big step semantics:

```
lemma while_never: "(c, s) ⇒ u ⟹ c ≠ WHILE (B True) DO c1"
lemma equiv_while_True: "(WHILE (B True) DO c1) ∼ (WHILE (B True) DO c2)"
```

This property is the first strong indication that big-step semantics, while nice to write and easy to reason about, may not be the right tool for all language properties. The main reason for this is that it does not talk about intermediate states or any intermediate input/output of the program. Many event-based programs have a loop of the form `WHILE B True DO c`, and of course it matters what precisely `c` is. Yet our lemma above says `c` can be replaced arbitrarily without effect. The effect of the replacement is not observable in the final state (which does not exist), but it is observable in intermediate states. If such properties are of interest, the more fine-grained small-step semantics is a better formal analysis tool.

### *2.1.4. Execution is deterministic*

In this section we investigate properties of the language itself. Having defined the semantics of the language as a relation, it is not immediately obvious for instance if execution in this language is deterministic or not. None of the `big_step` introduction rules in isolation would preclude multiple nondeterministic execution. On the other hand, we have not provided any particular rule that would introduce nondeterminism into IMP.

Formally, the language is deterministic if for any two executions of the same statement and we will always arrive in the same terminal state if we start in the same initial state.

With our automation setup for `big_step` this is a one-line proof – almost too easy for Isabelle:

```
theorem deterministic: "⟦ (c,s) ⇒ t; (c,s) ⇒ u ⟧ ⟹ u = t"
```

Note that the automation in this proof is not completely obvious. Using `auto` for instance leads to non-termination, but the backtracking capabilities of `blast` manage to solve each subgoal. To discover that `blast+` works, one would try each case separately and if `blast` works for the interesting cases, attempt `blast+` to cover all.

In lectures about semantics of programming languages the property that IMP is deterministic is often taken as a first simple proof to introduce rule induction and language properties. Solving this proof fully automatically is counterproductive for such a demonstration. Isabelle's Isar language allows us to provide level of readable detail similar to what would be presented in a lecture, but also allows us to hide boring cases by using automation.

The following proof demonstrates this technique. It is not necessary to step through the proof in detail to explain it in a lecture, but often using the tool in a live demonstration makes the material more accessible.

```
theorem
  "(c,s) ⇒ t  ⟹  (c,s) ⇒ u  ⟹  u = t"
proof (induct arbitrary: u rule: big_step.induct)
  — the simple SKIP case for demonstration:
  fix s u assume "(SKIP,s) ⇒ u"
  thus "u = s" by blast
next
  — and the only really interesting case, WHILE:
  fix b c s s1 s2 u
  assume "IH_c": "⋀u. (c,s) ⇒ u ⟹ u = s2"
  assume "IH_w": "⋀u. (WHILE b DO c,s2) ⇒ u ⟹ u = s1"

  assume "bval b s" and "(WHILE b DO c,s) ⇒ u"
  then obtain s' where
      c: "(c,s) ⇒ s'" and
      w: "(WHILE b DO c,s') ⇒ u"
    by auto

  from c "IH_c" have "s' = s2" by blast
  with w "IH_w" show "u = s1" by blast
qed blast+ — prove the rest automatically
```

This concludes the section about deterministic execution and thereby the section on big-step semantics.

In this section, we have defined the big-step semantics of IMPinductively, we have set up Isabelle's automation capabilities for it, and we have explored some first proofs about properties of the language.

## 2.2. *Small-Step Semantics of Commands*

### 2.2.1. *The transition relation*

This section introduces the small-step semantics of IMP. The big-step semantics gave us the completed execution of a program from its initial state. Short of inspecting the derivation tree of big-step introduction rules, it did not allow us to explicitly observe intermediate execution states. This also made it hard for us to talk about termination.

If we want to observe partial executions, for instance if we would like to talk about the interleaved execution of concurrent programs, we can make them explicit using small-step semantics. The main idea for representing a partial execution is to introduce a concept of how far execution has progressed in the program. Traditionally, for a high-level language like IMP, we modify the type of the big-step judgement from `com × state ⇒ state ⇒ bool` to something like `com × state ⇒ com × state ⇒ bool`. The second `com × state` component of the judgement is the result state of one small, atomic execution step together with a modified program statement that represents what still has to be executed.

The idea is easiest to understand by looking at the set of rules. They define one atomic execution step. The execution of a program is a sequence of such steps.

```
inductive
  small_step :: "com * state ⇒ com * state ⇒ bool" (infix "→" 55)
where
Assign:  "(x ::= a, s) → (SKIP, s(x := aval a s))" |

Semi1:   "(SKIP;c₂,s) → (c₂,s)" |
Semi2:   "(c₁,s) → (c₁',s') ⟹ (c₁;c₂,s) → (c₁';c₂,s')" |

IfTrue:  "bval b s ⟹ (IF b THEN c₁ ELSE c₂,s) → (c₁,s)" |
IfFalse: "¬bval b s ⟹ (IF b THEN c₁ ELSE c₂,s) → (c₂,s)" |

While:   "(WHILE b DO c,s) → (IF b THEN c; WHILE b DO c ELSE SKIP,s)"
```

Going through the rules above we see that variable assignment is an atomic step. We represent the terminated program by `SKIP`. There are two rule for the semicolon: either the first part is fully executed already (signified by `SKIP`), then we can just continue with the second part, or the first part still can be executed further, in which case we perform that execution step and return its result. An if-then-else reduces either to the if-branch or the else-branch, depending on the value of the condition. The final rule is the while loop: we just unroll the loop once. The subsequent execution steps will take care of testing the condition and possibly execution the body.

Had we wanted to observe partial execution of arithmetic or boolean expressions, we could have introduced a small-step semantics for these as well and made the corresponding small-step rules for assignment, if, and while non-atomic in the same style as the semicolon rules.

Before we can define program execution as a sequence of small-step semantics steps, we make a small detour to define the transitive, reflexive closure of binary predicates. Isabelle already comes with such a definition for relations. We briefly re-do it for predicates below to be able to keep big-step and small-step judgements analogous.

```
inductive
  star :: "('a ⇒ 'a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool"
for r where
refl:  "star r x x" |
step:  "r x y ⟹ star r y z ⟹ star r x z"
```

The main property we will need is transitivity which is easily shown by rule induction:

```
lemma star_trans: "star r x y ⟹ star r y z ⟹ star r x z"
```

A common case is concluding that if two elements are in the base predicate, they are in the closure. Again this rule is useful for proof search and for simplification. It may require backtracking—not every time we see *star r x y* it is enough to prove *r x y*.

```
lemma step1[simp, intro]: "r x y ⟹ star r x y"
```

Returning from our excursion back to theory *Small_Step*, we can now define the execution of a program as the transitive, reflexive closure of the *small_step* judgement:

```
abbreviation
  small_steps :: "com * state ⇒ com * state ⇒ bool" (infix "→*" 55)
  where "x →* y == star small_step x y"
```

## 2.2.2. Executability

As with the other relations we have defined, we make the rules available to the code generator for looking at examples. This time, we will get multiple elements in the set returned by the `values` command. They correspond to all partial executions of the program.

```
values "{(c',map t [''x'',''y'',''z'']) |c' t.
   (''x'' ::= V ''z''; ''y'' ::= V ''x'',
    [''x'' → 3, ''y'' → 7, ''z'' → 5]) →* (c',t)}"
```

The result Isabelle displays contains four steps, starting with the original program in the initial state, going through partial execution of the the two assignments, and ending with the final state of the final program *SKIP*: *{(''x'' ::= V ''z''; ''y'' ::= V ''x'', [3, 7, 5]), (SKIP; ''y'' ::= V ''x'', [5, 7, 5]), (''y'' ::= V ''x'', [5, 7, 5]), (SKIP, [5, 5, 5])}*

## 2.2.3. Proof infrastructure

As always, we start the proof part in theory Small_Step with setting up automation and infrastructure.

We make the introduction rules available for the simplifier and for proof search. For the rules with assumptions this is not entirely safe and might lead to non-termination in proof methods, but we take that risk to gain more automation when it works. After including the usual setup for rule inversion using the command `inductive_cases`, we are already done. As a test for the setup, we prove that, as in the big-step case, the rules give us a deterministic language.

```
lemma deterministic:
  "cs → cs' ⟹ cs → cs'' ⟹ cs'' = cs'"
```

The proof is as automatic as the big-step case.

## 2.2.4. Equivalence with big-step semantics

Having defined and set up an alternative semantics for the same language, the first interesting question is of course if our definitions are equivalent and describe the same executions.

The game plan for this proof is to show both directions separately: if there is a big-step execution, there is an equivalent small-step execution and vice versa.

The proof of the first direction is by rule induction on the big-step judgement. For each case we then construct the equivalent small-step sequence. In the semicolon case, we will need to conclude from the two separate small-step executions of *c1* and *c2* that *c1; c2* will terminate in the same result. We extract this into a separate lemma which needs another small helper lemma that allows us to lift an execution of *(c1, s) →* (c1′, s′)* into a semicolon context.

```
lemma star_semi2: "(c1,s) →* (c1′,s′) ⟹ (c1;c2,s) →* (c1′;c2,s′)"
lemma semi_comp: "⟦ (c1,s1) →* (SKIP,s2); (c2,s2) →* (SKIP,s3) ⟧
   ⟹ (c1;c2, s1) →* (SKIP,s3)"
```

The following proof is now the rule induction on the big-step semantics. The precise lemma statement is that if *(c, s) ⇒ t* then *(c, s) →* (SKIP, t)*, that is, there is a sequence of small steps that terminates successfully in *SKIP* with the same state *t*. The proof corresponds to one written on the black board where one would construct chains of → and →* steps.

```
lemma big_to_small:
  "(c,s) ⇒ t ⟹ (c,s) →* (SKIP,t)"
proof (induct rule: big_step_induct)
  fix s show "(SKIP,s) →* (SKIP,s)" by simp
next
  fix x a s show "(x ::= a,s) →* (SKIP, s(x := aval a s))" by auto
next
  fix c1 c2 s1 s2 s3
  assume "(c1,s1) →* (SKIP,s2)" and "(c2,s2) →* (SKIP,s3)"
  thus "(c1;c2, s1) →* (SKIP,s3)" by (rule semi_comp)
next
  fix s::state and b c0 c1 t
  assume "bval b s"
  hence "(IF b THEN c0 ELSE c1,s) → (c0,s)" by simp
  also assume "(c0,s) →* (SKIP,t)"
  finally show "(IF b THEN c0 ELSE c1,s) →* (SKIP,t)" . — = by assumption
next
  fix s::state and b c0 c1 t
  assume "¬bval b s"
  hence "(IF b THEN c0 ELSE c1,s) → (c1,s)" by simp
  also assume "(c1,s) →* (SKIP,t)"
  finally show "(IF b THEN c0 ELSE c1,s) →* (SKIP,t)" .
next
  fix b c and s::state
  assume b: "¬bval b s"
  let ?if = "IF b THEN c; WHILE b DO c ELSE SKIP"
  have "(WHILE b DO c,s) → (?if, s)" by blast
  also have "(?if,s) → (SKIP, s)" by (simp add: b)
  finally show "(WHILE b DO c,s) →* (SKIP,s)" by auto
next
  fix b c s s′ t
  let ?w  = "WHILE b DO c"
  let ?if = "IF b THEN c; ?w ELSE SKIP"
  assume w: "(?w,s′) →* (SKIP,t)"
  assume c: "(c,s) →* (SKIP,s′)"
  assume b: "bval b s"
  have "(?w,s) → (?if, s)" by blast
  also have "(?if, s) → (c; ?w, s)" by (simp add: b)
  also have "(c; ?w,s) →* (SKIP,t)" by(rule semi_comp[OF c w])
  finally show "(WHILE b DO c,s) →* (SKIP,t)" by auto
qed
```

Each case of the induction could also be proved automatically.

The other direction of the proof is even shorter. It cannot necessarily be called the easier direction, though, because the proof idea is less obvious. The main statement we want is `(c, s) →* (SKIP, t) ⟹ (c, s) ⇒ t`. Our first attempt would be rule induction on the derivation of the transitive closure. It quickly becomes clear that the lemma statement is too specialised for this. If we only consider steps that terminate in `SKIP`, we cannot chain them in the induction. The trick, as always, is to suitably generalise the lemma statement.

In this case, if we generalise `SKIP` to an arbitrary `c'`, the statement does not make sense any more, because the big-step semantics does not have any concept for an intermediate `c'`. The key observation is that the big-step semantics always executes the program fully and the state `(c', s')` is just an intermediate state in this execution. That means, executing the 'rest' `(c', s')` and executing the original `(c, s)` should give us precisely the same result in the big-step semantics. Formally: `⟦(c, s) →* (c', s'); (c', s') ⇒ t⟧ ⟹ (c, s) ⇒ t`. If we substitute `SKIP` for `c'`, we get that `s'` must be `t` and we are back to what we where out to show originally.

This new statement can now be proved by induction on the transitive closure. We extract the step case into its own lemma which Isabelle proves automatically after rule induction. It is the same statement, only for a single step in the small-step semantics.

```
lemma small1_big_continue:
  "(c,s) → (c',s') ⟹ (c',s') ⇒ t ⟹ (c,s) ⇒ t"
lemma small_big_continue:
  "(c,s) →* (c',s') ⟹ (c',s') ⇒ t ⟹ (c,s) ⇒ t"
lemma small_to_big: "(c,s) →* (SKIP,t) ⟹ (c,s) ⇒ t"
```

Both directions together let us conclude the equivalence we were aiming at.

```
theorem big_iff_small:  "(c,s) ⇒ t = (c,s) →* (SKIP,t)"
```

## 3. Types

### *3.1. A Typed Language*

#### *3.1.1. A New Language*

In this section we show a very basic static type system as a typical application the formal definition of language semantics.

The idea is to define the type system formally and to use the semantics for stating and proving its soundness.

The IMP language we have used so far is not well-suited for this proof, because it has only one type of values. This is not enough for even a simple type system. To make things at least slightly non-trivial, we invent a new language that computes on reals as well as integers. This means our value type now has two alternatives:

```
datatype val = Iv int | Rv real
```

Variable names and state are as before:

```
type_synonym name = string
type_synonym state = "name ⇒ val"
```

and arithmetic expressions now have two kinds of constants: `int` and `real`.

```
datatype aexp =  Ic int | Rc real | V name | Plus aexp aexp
```

In contrast to vanilla IMP, we can now write arithmetic expressions that make no sense and have no semantics. The expression `Plus (Ic 1) (Rc 3)` for example is trying to add an integer to a real number. Assuming for a moment that these are fundamentally incompatible types that cannot possibly be reconciled in an addition, we would like to express in our semantics that this is not a defined program. One alternative would be a denotational style as before that returns `val option` with `None` for the undefined cases, but that would mean we would have to explicitly consider all undefined cases. Instead, we write an operational semantics judgement for arithmetic expressions. Since we are not interested in intermediate executions at this point, we are happy with a big-step style.

The judgement relates an expression, the state it is evaluated in, and the value it is evaluated to. We refrain from introducing additional syntax and call this judgement `taval` for *typed arithmetic value* of an expression.

```
inductive taval :: "aexp ⇒ state ⇒ val ⇒ bool" where
"taval (Ic i) s (Iv i)" |
"taval (Rc r) s (Rv r)" |
"taval (V x) s (s x)" |
"taval a1 s (Iv i1) ⟹ taval a2 s (Iv i2) ⟹
  taval (Plus a1 a2) s (Iv(i1+i2))" |
"taval a₁ s (Rv r1) ⟹ taval a2 s (Rv r2) ⟹
  taval (Plus a1 a2) s (Rv(r1+r2))"
```

The definition is straight forward. The interesting cases are the ones that are not there. For instance, there is no rule to add a `real` to an `int`. We only needed to provide rules for the cases that make sense and we have implicitly defined what the error cases are.

The syntax for boolean expressions remains unchanged wrt. IMP. The evaluation, however, of boolean expressions is different to IMP. If we want to make use of the operational semantics for arithmetic expressions that we just defined, we need to use the same style for boolean expressions. Even though the case distinction for boolean expressions is complete, i.e. the rules cover everything and there are no error cases left, we have implicitly correctly propagated the evaluation errors from arithmetic expressions: if there is no execution for an arithmetic expression in state *s*, there is no execution for a boolean expression that mentions it in the same state.

```
inductive tbval :: "bexp ⇒ state ⇒ bool ⇒ bool" where
```

```
"tbval (B bv) s bv" |
"tbval b s bv ⟹ tbval (Not b) s (¬ bv)" |
"tbval b₁ s bv1 ⟹ tbval b2 s bv2 ⟹ tbval (And b1 b2) s (bv1 & bv2)" |
"taval a₁ s (Iv i1) ⟹ taval a2 s (Iv i2) ⟹
  tbval (Less a1 a2) s (i1 < i2)" |
"taval a₁ s (Rv r1) ⟹ taval a2 s (Rv r2) ⟹
  tbval (Less a1 a2) s (r1 < r2)"
```

The syntax for commands is again unchanged. The definition of the small-step semantics for commands is almost unchanged as well. It merely refers to the new judgements for arithmetic and boolean expressions.

Small-step semantics are better suited for type soundness proofs, because error states are explicitly visible in intermediate states: if there is an error, the semantics gets stuck in a non-terminal program configuration with no further progress possible.

```
inductive
  small_step :: "(com × state) ⇒ (com × state) ⇒ bool" (infix "→" 55)
where
Assign:  "taval a s v ⟹ (x ::= a, s) → (SKIP, s(x := v))" |

Semi1:   "(SKIP;c,s) → (c,s)" |
Semi2:   "(c1,s) → (c1',s') ⟹ (c1;c2,s) → (c1';c2,s')" |

IfTrue:  "tbval b s True ⟹ (IF b THEN c1 ELSE c2,s) → (c1,s)" |
IfFalse: "tbval b s False ⟹ (IF b THEN c1 ELSE c2,s) → (c2,s)" |

While:   "(WHILE b DO c,s) → (IF b THEN c; WHILE b DO c ELSE SKIP,s)"
```

As before, the execution of a program is a sequence of small steps.

```
abbreviation
  small_steps :: "com * state ⇒ com * state ⇒ bool" (infix "→*" 55)
  where "x →* y == star small_step x y"
```

### *3.1.2. The Type System*

Having defined the new language and its semantics above in a way such that execution can fail or get stuck, we are now defining a type system that statically determines safe programs, i.e. programs that are guaranteed not to get into stuck or error states.

The type system we use for this is very rudimentary, it has only two types: int and real, written as the constructors `Ity` and `Rty`.

```
datatype ty = Ity | Rty
```

The purpose of the type system is to keep track of the type of each variable and to allow only compatible combinations in expressions. For this purpose, we define a typing environment as a function from variable name to type.

```
type_synonym tyenv = "name ⇒ ty"
```

With this, we can give typing rules for arithmetic expressions. The idea is simple: constants have fixed type, variables have the type the environment Γ prescribes, and `Plus` can be typed with type $\tau$ if both operands have the same type $\tau$. The formalisation is straight forward.

```
inductive atyping :: "tyenv ⇒ aexp ⇒ ty ⇒ bool"
  ("(1_/ ⊢/ (_ :/ _))" [50,0,50] 50)
where
Ic_ty:   "Γ ⊢ Ic i : Ity" |
Rc_ty:   "Γ ⊢ Rc r : Rty" |
V_ty:    "Γ ⊢ V x : Γ x" |
Plus_ty: "Γ ⊢ a1 : τ ⟹ Γ ⊢ a2 : τ ⟹ Γ ⊢ Plus a1 a2 : τ"
```

The typing rules for booleans are even simpler. We do not need a result type, because it will always be bool. For the most part, we just need to capture that boolean expressions are well-typed if their subexpression are well-typed. The interesting case is the connection to arithmetic expressions in `Less`. Here we demand that both operands have the same type $\tau$.

```
inductive btyping :: "tyenv ⇒ bexp ⇒ bool" (infix "⊢" 50)
where
B_ty: "Γ ⊢ B bv" |
Not_ty: "Γ ⊢ b ⟹ Γ ⊢ Not b" |
And_ty: "Γ ⊢ b1 ⟹ Γ ⊢ b2 ⟹ Γ ⊢ And b1 b2" |
Less_ty: "Γ ⊢ a1 : τ ⟹ Γ ⊢ a2 : τ ⟹ Γ ⊢ Less a1 a2"
```

Similarly, commands are well-typed if their subexpressions are well typed. The only non-regular case here is assignment: we demand that the arithmetic expression has the same type as the variable it is assigned to.

```
inductive ctyping :: "tyenv ⇒ com ⇒ bool" (infix "⊢" 50) where
Skip_ty: "Γ ⊢ SKIP" |
Assign_ty: "Γ ⊢ a : Γ(x) ⟹ Γ ⊢ x ::= a" |
Semi_ty: "Γ ⊢ c1 ⟹ Γ ⊢ c2 ⟹ Γ ⊢ c1;c2" |
If_ty: "Γ ⊢ b ⟹ Γ ⊢ c1 ⟹ Γ ⊢ c2 ⟹ Γ ⊢ IF b THEN c1 ELSE c2" |
While_ty: "Γ ⊢ b ⟹ Γ ⊢ c ⟹ Γ ⊢ WHILE b DO c"
```

This concludes the definition of the type system itself. Type systems can be arbitrarily complex. The one here is intentionally simple to show the structure of a type soundness proof without getting side tracked in interesting type system details.

### *3.1.3. Well-typed Programs Do Not Get Stuck*

In this section we prove that the type system defined above is sound. Robert Milner coined the phrase that "well-typed programs do not go wrong", i.e. that well-typed programs will not exhibit any runtime errors such as segmentation faults or undefined execution. In a small-step semantics, this can be expressed nicely as the formal statement that well-typed programs never get stuck. They either terminate successfully, or they make further well-defined progress.

The main new concept we need to show type safety is that of a well-typed state. We so far have the state assigning values to variables and we have the type system statically assigning types to variables in the program. We connect these two concepts by defining a judgement that determines if a runtime state is compatible with a typing environment for variables. We call such states well-typed, and the judgement `styping` below. We also say that a state `s` *conforms* to a typing environment Γ.

With this judgement, the type soundness property naturally decomposes into two parts: preservation and progress. Preservation means that well-typed states stay well-typed during execution. Progress means that in a well-typed state, the program either terminates successfully or can make one more step of execution progress.

We start the formalisation by defining a function from values to types. In the IMP world, this is very simple. In more sophisticated type systems, there may be multiple types that can be assigned to a value and we may need a compatibility or sub-type relation between types to define the `styping` judgement. In our case, we merely have to map `Iv` values to `Ity` types and `Rv` values to `Rty` types.

```
fun type :: "val ⇒ ty" where
"type (Iv i) = Ity" |
"type (Rv r) = Rty"
```

The following two equations allow us to infer the value constructor from the result of the `type` function.

```
lemma type_Ity [simp]: "type v = Ity ⟷ (∃i. v = Iv i)"
lemma type_Rty [simp]: "type v = Rty ⟷ (∃r. v = Rv r)"
```

Our `styping` judgement for well-typed states is now very simple: for all variables, the type of the runtime value must be exactly the type predicted in the typing environment.

```
definition styping :: "tyenv ⇒ state ⇒ bool" (infix "⊢" 50)
where "Γ ⊢ s  ⟷  (∀x. type (s x) = Γ x)"
```

We now have everything defined to start the soundness proof. The plan is to prove progress and preservation for statements, and to conclude from that the final type soundness statement that an execution started in a well-typed state will never get stuck. To prove progress and preservation for statements, we will need the same properties for arithmetic and boolean expressions.

Preservation for arithmetic expressions means the following: If expression `a` has type `τ` under environment Γ, if `a` evaluates to `v` in state `s`, and if `s` conforms to Γ, then the type of the result `v` must be `τ`. The proof is by rule induction on the type derivation `Γ ⊢ a : τ`. Since we have declared rule inversion on `taval` to be used automatically, Isabelle manages the rest automatically if we unfold the definition of `styping`.

```
lemma apreservation:
  "Γ ⊢ a : τ ⟹ taval a s v ⟹ Γ ⊢ s ⟹ type v = τ"
```

The progress lemma is more verbose. It is almost the only place where something interesting is concluded in the soundness proof: there is the potential of something going wrong. If the operands of a `Plus` were of incompatible type, there would be no value `v` the expression evaluates to. Of course, the type system excludes precisely this case.

The progress statement is as standard as the preservation statement for arithmetic expressions: given that `a` has type `τ` under environment Γ, and given a conforming state `s`, there must exist a result value `v` such that `a` evaluates to `v` in `s`.

The proof is again by rule induction on the typing derivation. The interesting case is *Plus*. The induction hypothesis gives us two values *v1* and *v2* for the subexpressions *a1* and *a2*. If *v1* is an integer, then, by preservation, the type of *a1* must have been *Ity*. The typing rule says the type of *a2* must be the same, so must, again by preservation, the type of *v2*. If the type of *v2* is *Ity* then *v2* must be an *Iv* value and we can conclude using the *taval* introduction rule for *Plus* that the execution has a result. Isabelle completes this reasoning chain automatically if we carefully provide it with the right facts and rules. The case for reals is analogous, and the other typing cases are easily solved automatically.

```
lemma aprogress: "Γ ⊢ a : τ ⟹ Γ ⊢ s ⟹ ∃v. taval a s v"
```

For boolean expressions, there is no preservation lemma, because the way the rules are written they obviously can only return boolean values. The progress statement makes sense, though, and follows the standard progress statement schema. As always, the proof is by rule induction on the typing derivation. The interesting case is where something could go wrong, namely where we execute arithmetic expressions in *Less*. The proof is very similar to the one for *Plus*: we obtain the values of the subexpressions; we perform a case distinction on one of them to reason about its type; we infer the other has the same type by typing rules and by preservation on arithmetic expressions; and we conclude that execution can therefore progress. Again this case is automatic if written carefully, the other cases are trivial.

```
lemma bprogress: "Γ ⊢ b ⟹ Γ ⊢ s ⟹ ∃v. tbval b s v"
```

For commands, there are two preservation statements, because our small-step semantics has two return values: program and state. We first show that the program remains well-typed and then that the state does. Both are by induction on the small-step semantics. They could be proved by induction on the typing derivation as well, but that would require us to declare additional rule inversion lemmas for the small-step rules which we have omitted above. Often it is preferable to try induction on the typing derivation first, because the type system typically has fewer cases. On the other hand, depending on the complexity of the language, the more fine grained information that is available in operational semantics might make the more numerous cases easier to prove in the other induction alternative. In both cases it pays off to design the structure of the rules in both systems such that they technically fit together nicely, for instance such that the decompose along the same syntactic lines.

The preservation of program typing is fully automatic in this simple language. The only mildly interesting case where we are not just transforming the program into a subexpression is the while loop. Here we just need to apply the typing rules for if-then-else and sequential composition and are done.

```
theorem ctyping_preservation:
  "(c,s) → (c',s') ⟹ Γ ⊢ c ⟹ Γ ⊢ c'"
```

The preservation of state well-typedness again composes easily through the induction. The place where something interesting happens is the substitution in the assignment operation. We update the state with a new value. Type preservation on expressions gives us that the new value has the same type, and unfolding the *styping* judgement shows that it is unaffected by substitutions that are type preserving. In more complex languages, there are likely to be a number of such substitution cases and the corresponding substitution lemma is a central piece of type soundness proofs.

```
theorem styping_preservation:
  "(c,s) → (c',s') ⟹ Γ ⊢ c ⟹ Γ ⊢ s ⟹ Γ ⊢ s'"
```

The progress lemma for commands is more verbose again. Here, we need to take into account that the program might have fully terminated. If it has not, and we have a well-typed program in a well-typed state, we demand that we can make at least one step.

This time the only induction alternative is on the typing derivation again. The cases with arithmetic and boolean expressions make use of the corresponding progress lemmas to generate the values the small-step rules demand. For *If*, we additionally perform a case distinction for picking the corresponding introduction rule. In the other cases, *SKIP* is trivial, *Semi* just applies the induction hypothesis and makes a case distinction if *c1* is *SKIP* or not, and *While* always trivially makes progress in the small-step semantics.

```
theorem progress:
  "Γ ⊢ c ⟹ Γ ⊢ s ⟹ c ≠ SKIP ⟹ ∃cs'. (c,s) → cs'"
```

All that remains is to assemble the pieces into the final type soundness statement: given any execution of a well-typed program started in a well-typed state, we are not stuck; we have either terminated successfully, or the program can perform another step.

The proof lifts the one-step preservation and progress results to a sequence of steps by induction on the transitive closure.

```
theorem type_sound:
  "(c,s) →* (c',s') ⟹ Γ ⊢ c ⟹ Γ ⊢ s ⟹
   c' = SKIP ∨ (∃cs''. (c',s') → cs'')"
```

This concludes the section on typing. We have seen at the example of a very simple type system what a type soundness statement entails, how it interacts with the small-step semantics, and how it is proved. While the proof itself will grow in complexity for more interesting language, the general schema of progress and preservation with a potential substitution lemma usually remains.

For the type soundness theorem to be meaningful, it is important that the failures the type system is supposed to prevent are observable in the semantics so that their absence can be shown. In a framework like the above, the precise definition of the small-step semantics carries the main meaning and strength of the type soundness statement.

## *3.2. Security Type Systems*

In the previous section we have seen a simple static type system with soundness proof. However, type systems can be used for more than the traditional concepts of integers, reals, etc. In theory, type systems can be arbitrarily complex logical systems used to statically predict properties of programs. In the following, we will look at a type system that aims to enforce a security property: the absence of information flows from private data into public channels.

Ensuring such properties based on a programming language analysis such as a type system is also called *language-based security*. An commonly used alternative to language-based security is the use of cryptography to ensure the secrecy of private data. Cryptography only admits probabilistic arguments (one could always guess the key), whereas language-based security allows a more absolute statement.

Note that these absolute statements are always with respect to assumptions on the execution environment. For instance, our proof below will have the implicit assumption that the machine actually behaves as our semantics predicts. There are practical ways in which these assumptions can be broken or circumvented: intentionally introducing hardware-based errors into the computation to deduce private data, direct physical observation of memory contents, deduction of private data by analysis of execution time, and more. These attacks make use of details that are not visible on the abstraction level of the semantic model our proof is based on — they are *covert channels* of information flow.

The field of language-based security is substantial [3], the type system and the soundness statement in the sections below go back to Volpano and Smith [4]. While language-based security had been investigated before Volpano and Smith, they were the first to give a security type system with a soundness proof that expresses the enforced security property in terms of the standard semantics of the language. Formalise their type system and proof in Isabelle, we will see that such non-trivial properties are comfortably within the reach of machine-checked interactive proof.

### *3.2.1. Security Levels and Expressions*

We begin by defining security levels. The idea is that each variable will have an associated security level. The type system will then enforce the policy that variables of 'higher' security level will not leak information to variables of 'lower' security levels.

In the literature, levels are often reduced to just two: high and low. We keep things slightly more general making levels natural numbers. We can then compare security levels by just writing $<$ and we can compute the maximal or minimal security level of two different variables by taking the maximum or minimum respectively. The term $l < l'$ in this system would mean that $l$ is less private or confidential than $l'$, so level $0::'a$ could be equated with 'public'.

It would be easily possible to generalise further and just assume a lattice of security levels with $<$, join, and meet operations.

```
type_synonym level = nat
```

For the type system and security proof below it would be sufficient to merely assume the existence of a HOL constant that maps variables to security levels. This would express that we assume each variable to possess a security level and that this level remains the same during execution of the program.

For the sake of showing examples — the general theory does not rely on it! —, we arbitrarily choose a specific function for this mapping: a variable of length $n$ has security level $n$:

```
definition sec :: "name ⇒ level" where  "sec n = size n"
```

The kinds of information flows we would like to avoid are exemplified by the following two:

- explicit: `low ::= high`
- implicit: `IF high1 < high2 THEN low ::= 0 ELSE low ::= 1`

The property we are after is called *noninterference*: a variation in the value of high variables should not interfere with the computation or values of low variables. 'High should not interfere with low.'

More formally, a program c guarantees noninterference iff for all states `s1` and `s2`: If `s1` and `s2` agree on low variables (but may differ on high variables!), then the states resulting from executing `(c, s1)` and `(c, s2)` must also agree on low variables.

As opposed to our previous type soundness statement, this definition compares the outcome of two executions of the same program in different, but related initial states. It requires again potentially different, but equally related final states.

With this in mind, we can now proceed to define the type system that will enforce this property. We begin by computing the security level of arithmetic and boolean expressions. We are interested in flows from higher to lower variables, so we compute the security level of an expression as the highest level of any variable that occurs in it:

```
fun sec_aexp :: "aexp ⇒ level" where
  "sec_aexp (N n) = 0" |
  "sec_aexp (V x) = sec x" |
  "sec_aexp (Plus a₁ a₂) = max (sec_aexp a₁) (sec_aexp a₂)"

fun sec_bexp :: "bexp ⇒ level" where
  "sec_bexp (B bv) = 0" |
  "sec_bexp (Not b) = sec_bexp b" |
  "sec_bexp (And b₁ b₂) = max (sec_bexp b₁) (sec_bexp b₂)" |
  "sec_bexp (Less a₁ a₂) = max (sec_aexp a₁) (sec_aexp a₂)"
```

A first lemma indicating that we are moving into the right direction will be that if we change the value of only variables with a higher level than `sec_aex a`, the value of `a` should be the same.

To express this property, we introduce notation for two states agreeing on the value of all variables below a certain security level. The concept is light-weight enough that a syntactic abbreviation is sufficient and avoids us having to go through the motions of setting up additional proof infrastructure.

We will need both ≤ and the strict < later on, so we define both here:

```
abbreviation eq_le :: "state ⇒ state ⇒ level ⇒ bool"
  ("(_ = _ '(≤ _'))" [51,51,0] 50) where
  "s = s' (≤ l) == (∀ x. sec x ≤ l ⟶ s x = s' x)"

abbreviation eq_less :: "state ⇒ state ⇒ level ⇒ bool"
  ("(_ = _ '(< _'))" [51,51,0] 50) where
  "s = s' (< l) == (∀ x. sec x < l ⟶ s x = s' x)"
```

With this, the proof of our first two security properties is simple and automatic:

```
lemma aval_eq_if_eq_le:
  "⟦ s₁ = s₂ (≤ l);  sec_aexp a ≤ l ⟧ ⟹ aval a s₁ = aval a s₂"
lemma bval_eq_if_eq_le:
  "⟦ s₁ = s₂ (≤ l);  sec_bexp b ≤ l ⟧ ⟹ bval b s₁ = bval b s₂"
```

### 3.2.2. *Syntax Directed Typing*

As usual in IMP, the typing for expressions was simple. We now define the typing rules for commands. We deviate a little from the standard presentation of Volpano and Smith and instead give an equivalent, syntax-directed form of the rules. This makes the rules directly executable and allows us to run examples. The main idea of the type system is to track the security level of variables that that decisions are made on, and make sure that their level is lower or equal to variables assigned to in that context.

We write `l ⊢ c` to say command `c` contains only safe flows to variables higher or equal to `l`.

```
inductive sec_type :: "nat ⇒ com ⇒ bool" ("(_/ ⊢ _)" [0,0] 50) where
Skip:   "l ⊢ SKIP" |
Assign: "⟦ sec x ≥ sec_aexp a;  sec x ≥ l ⟧ ⟹ l ⊢ x ::= a" |
Semi:   "⟦ l ⊢ c₁;  l ⊢ c₂ ⟧ ⟹ l ⊢ c₁;c₂" |
If:     "⟦ max (sec_bexp b) l ⊢ c₁;  max (sec_bexp b) l ⊢ c₂ ⟧ ⟹
           l ⊢ IF b THEN c₁ ELSE c₂" |
While:  "max (sec_bexp b) l ⊢ c ⟹ l ⊢ WHILE b DO c"
```

Going through the rules in detail, we have defined `SKIP` to be safe at any level. We have have defined assignment to be safe if the level of `x` is higher than or equal to the level of the information source `a`, but lower than or equal to `l`. For semicolon to conform to a level `l`, we just recursively demand that both parts conform to the same level `l`. As previously shown in the motivating example, the `IF` statement could admit indirect flows. We prevent these by demanding that for if-then-else to conform to `l`, both `c1` and `c2` have to conform to level `l` or the level of the decision expression, whichever is higher. We can conveniently express this with the maximum operator `max`. The While case is similar to an If: the body must have at least the level of `b` and of the whole statement.

Using the `max` function makes the type system executable if we tell Isabelle to treat the level and the program as input to the predicate.

Testing our intuition about what we have just defined, we execute four examples for various security levels.

```
value "0 ⊢ IF Less (V ''x1'') (V ''x'') THEN ''x1'' ::= N 0 ELSE SKIP"
value "1 ⊢ IF Less (V ''x1'') (V ''x'') THEN ''x''  ::= N 0 ELSE SKIP"
value "2 ⊢ IF Less (V ''x1'') (V ''x'') THEN ''x1'' ::= N 0 ELSE SKIP"
value "3 ⊢ IF Less (V ''x1'') (V ''x'') THEN ''x1'' ::= N 0 ELSE SKIP"
```

We expect the first evaluation to yield `True`: the condition has level 2, the assignment sets a level 2 variable, and the context is 0. The next line has to yield `False`: we are in a level 1 context overall, but since the condition has level 2, we are not allowed to assign to a level 1 variable. The next line is `True` again, whereas the last line is `False`: while the if-statement itself is fine, the context says we can at most assign to level 3 variables.

As we can already see from these simple examples, the type system is not complete: it will reject some safe programs as unsafe. For instance, if the value of `x` in the second statement was already 0 in the beginning, the context would not have mattered, we only would have overwritten 0 with 0. As with any type system that can be checked automatically we should not expect otherwise. The best we can hope for is a safe approximation such that the false alarms are hopefully programs that rarely occur in practise or that can be rewritten easily.

It is the case that the Volpano-Smith security type system in its simple form has been criticised as too restrictive. It excludes too many safe programs. This can be addressed by making the type system more refined, more flexible, and more context aware.

For demonstrating the type system and its soundness proof in these notes, we will stick to its simplest form.

### *3.2.3. Soundness*

We introduced the correctness statement for this type system as noninterference: two executions of the same program started in related states end up in related states. The relation in our case is that the values of variables below security level *l* are the same. Formally, this is the following statement
`⟦(c, s) ⇒ s'; (c, t) ⇒ t'; 0 ⊢ c; s = t (≤ l)⟧ ⟹ s' = t' (≤ l)`

The key lemma in the argument for this proof is *confinement*: an execution that is typed safe in context *l* can only change variables of level *l* and above, or conversely, all variables below *l* will remain unchanged.

An important property in that lemma is the so-called anti-monotonicity of the type system: a command that is typeable in *l* is also typeable in any level smaller than *l*. In fact, anti-monotonicity is directly part of the type system as a typing rule in the standard presentation of Volpano-Smith. In exchange, the standard presentation does not need an explicit *max* calculation. Here, we can derive anti-monotinicity as a lemma by rule induction on the type system:

```
lemma anti_mono: "⟦ l ⊢ c;  l' ≤ l ⟧ ⟹ l' ⊢ c"
```

In this proof, we used the sledgehammer tool extensively after the main proof idea. The automated provers found an answer for each of the subgoals.

We now come to the confinement property mentioned above: if we have an execution of *c* from *s* to *t*, and *c* is safe in context *l*, then the states *s* and *t* agree on all variables below level *l*. The first instinct may be to try rule induction on the type system again, but the while-case will only give us an induction hypothesis about the body when we will have to show our goal for the whole loop. Rule induction on the big-step execution on the other hand does get us through. In the if- and while-cases, we make use of anti-monotinicity to instantiate the induction hypothesis. In the if-true-case, for instance, the hypothesis is `l ⊢ c1 ⟹ s = t (< l)`, but from the type system we only know `max (sec_bexp b) l ⊢ c1`. Anti-monotonicity allows us to reason from *max* to *l*.

```
lemma confinement: "⟦ (c,s) ⇒ t;  l ⊢ c ⟧ ⟹ s = t (< l)"
```

With these two lemmas, we can start the main noninterference proof.

Similarly to above, we will not get through with induction on the typing derivation, but we do manage with induction on the big-step execution. This time, the proof is a little more involved.

```
theorem noninterference:
  "⟦ (c,s) ⇒ s'; (c,t) ⇒ t';  0 ⊢ c;  s = t (≤ l) ⟧ ⟹ s' = t' (≤ l)"
```

The base case of the proof is automatic, as it should be.

The assignment case is already somewhat interesting. First, we note that `s'` is the usual state update `s(x → aval a s)` in the first big-step execution. We perform rule inversion for the second execution to get the same update for `t`. We also perform rule inversion on the typing statement to get the relationship between security levels of `x` and `a`. Now we show that the two updated states `s'` and `t'` still agree on all variables below `l`. Isabelle's auto method tells us that it is sufficient to show that the states agree on the new value if the level of `x` is below `l`, and that all other variables `y` below `l` still agree as before. In the first case, looking at `x`, we know from above that `sec_aexp a ≤ sec x`. Hence, by transitivity, we have that the `a` has a level no more than `l`. This is enough for our noninterference result on expressions to apply, given that we also know `s = t (≤ l)` from the premises. The case for all other variables below `l` is simple and follows directly from `s = t (≤ l)`.

In the semicolon case, we merely need to compose the induction hypotheses. This is solved automatically.

If-then-else has two symmetric cases as usual. We will look only at if-true in more detail. We begin the case by noting via rule inversion that both branches are safe to level `sec_bexp b`, since the maximum with 0 is the identity. Then we perform a case distinction: either the level of `b` is `≤ l` or it is not. If `sec_bexp b ≤ l`, i.e. the if-decision is on a more public level than `l`, then `s` and `t` which agree below `l` also agree below `sec_bexp b`. That in turn means by our noninterference lemma for expressions that they evaluate to the same result, so `bval b s = True` and `bval b t = True`. We already noted `sec_bexp b ⊢ c1` by rule inversion, and with anti-monotonicity, we get the necessary `0 ⊢ c1` to apply the induction hypothesis and conclude the case. In the other case, if `l < sec_bexp b`, i.e. a condition on a more confidential level than `l` we do not now that both if-statements will take the same branch. However, we do know that the whole statement is a high-confidentiality computation. It must be safe to level `sec_bexp b`, since the parts are safe to the maximum of this level and `l`, and we now can apply confinement: everything below `sec_bexp b` will be preserved — in particular the state of variables up to `l`. This is true for `t` to `t'` as well as `s` to `s'`. Together with the initial `s = t (≤ l)`, we can conclude `s' = t' (≤ l)` which closes the whole if-true case.

The if-false and while-false cases are analogous. Either the conditions evaluate to the same value and we can apply the induction hypothesis, or the security level is high enough such that we can apply confinement.

Even the while-true case is very similar. Here, we have to work slightly harder to apply the induction hypothesis, once for the body and once for the rest of the loop, but the confinement side of the argument stays the same.

### *3.2.4. The Standard Typing System*

The judgement `l ⊢ c` presented above is nicely intuitive and executable. As mentioned, however, the standard formulation is slightly different, replacing the maximum computation by the anti-monotonicity rule. We introduce the standard system now and show equivalence with our formulation.

```
inductive sec_type' :: "nat ⇒ com ⇒ bool" ("(_/ ⊢'' _)" [0,0] 50) where
Skip':    "l ⊢' SKIP" |
Assign':  "⟦ sec x ≥ sec_aexp a; sec x ≥ l ⟧ ⟹ l ⊢' x ::= a" |
Semi':    "⟦ l ⊢' c₁;  l ⊢' c₂ ⟧ ⟹ l ⊢' c₁;c₂" |
If':      "⟦ sec_bexp b ≤ l;  l ⊢' c₁;  l ⊢' c₂ ⟧ ⟹
             l ⊢' IF b THEN c₁ ELSE c₂" |
While':  "⟦ sec_bexp b ≤ l;  l ⊢' c ⟧ ⟹ l ⊢' WHILE b DO c" |
anti_mono': "⟦ l ⊢' c;  l' ≤ l ⟧ ⟹ l' ⊢' c"
```

The equivalence proof goes by rule induction on the respective type system in each direction separately. We use sledgehammer to automatically find a proof for each subgoal.

```
lemma sec_type_sec_type': "l ⊢ c ⟹ l ⊢' c"
lemma sec_type'_sec_type: "l ⊢' c ⟹ l ⊢ c"
```

## 4. Compiler

This section presents another application of programming language semantics: proving compiler correctness.

To this end, we will define a small machine language, based on a simple stack machine as it would be found for instance in the Java Virtual Machine. We then write a compiler from IMP and prove that the compiled program has the same semantics as the source program. The compiler will contain a very simple, standard optimisation for boolean expressions, but is otherwise non-optimising.

As in the other sections, the emphasis here is on showing the structure and main setup of such a proof. Our compiler proof shows the core of the argument, but compared to real compilers we make drastic simplifications: our target language is comparatively high-level, we do not consider optimisations, we ignore the compiler front-end, and our source language does not contain any concepts that are particularly hard to translate into machine language.

Two compiler correctness proofs in the literature are for the Java-like language Jinja [1] in a style that is similar to the one presented here, and more recently, the fully realistic, optimising C-compiler CompCert by Leroy et al [2] which compiles to PowerPC, x86, and ARM architectures. Both proofs are naturally more complex than the one presented here, both working with the concept of intermediate languages and multiple compilation stages. This is done to simplify the argument and to concentrate on specific issues on each level.

We begin by defining the instruction set architecture and semantics of our stack machine. Working with proofs on the machine language, we found it convenient for the program counter to admit negative values, i.e. to be of type `int` instead of the initially more intuitive `nat`. The effect of this choice is that various decomposition lemmas about machine executions have nicer algebraic properties and less preconditions than their `nat` counterparts. Such effects are usually discovered during the proof, not a priory.

Because of this choice we make a small detour before we write down the machine semantics.

### 4.1. List setup

Our machine language will model programs as lists of instructions and our `int` program counter will need to index into these lists. Isabelle comes with a predefined list index operator `nth`, but it works on `nat`. Instead of constantly converting between `int` and `nat` and dealing with the arising side-conditions in proofs, we define our own `int` version of `nth`. The size of lists is similar, but here the conversion does not present us with side conditions, because we only convert into the easy direction from `nat` to `int`.

```
abbreviation "isize xs == int (length xs)"

primrec
  inth :: "'a list => int => 'a" (infixl "!!" 100) where
  inth_Cons: "(x # xs) !! n = (if n = 0 then x else xs !! (n - 1))"
```

The only lemma we need about the new `inth` is indexing over append:

```
lemma inth_append [simp]:
  "0 ≤ n ⟹ (xs @ ys) !! n =
             (if n < isize xs then xs !! n else ys !! (n - isize xs))"
```

## *4.2. Instructions and Stack Machine*

We are now ready to define the machine itself. To keep things simple, we reuse the concepts of values and variable names from the source language. Values pose no problems, but variable names as strings are a little strange in a low-level machine, usually one would have to map such names to address locations first. We skip this complication here. It could be added easily in a separate additional compilation step for instance.

The instructions in our machine are the following:

```
datatype instr =  LOADI val | LOAD name    | ADD | STORE name |
                  JMP int   | JMPLESS int | JMPGE int
```

In detail, they are: load immediate value onto the stack, load value of a variable, add the two topmost stack values, store the top of stack into a variable, jump by a relative value, compare the two topmost stack elements and jump if the top is less, compare and jump if the top is greater or equal.

These are enough to compile IMP programs. A real machine would have significantly more arithmetic and comparison operators, different addressing modes that are useful to implement procedure stacks and pointers, potentially a number of primitive datatypes that the machine understands, and a number of instructions to deal with hardware features such as the memory management subsystem that we ignore completely in this formalisation.

As with the source language, after defining program instructions or statements, we proceed by defining the state such programs run on followed by the definition of the actual semantics.

Our state or program configuration consists of an `int` program counter, a variable assignment for which we re-use the type `state` from the source language, and a stack which we model as a list of values.

```
type_synonym stack = "val list"
type_synonym config = "int×state×stack"
```

We now define the semantics of machine execution in multiple levels. First, we define what effect a single instruction has on a configuration, then we define how an instruction is selected from the program, and finally we take the transitive closure to get full machine program executions.

In the first level, the effect of single instructions, we will need to talk about the second element of a list. The following notations makes this slightly nicer to read:

```
abbreviation "hd2 xs == hd(tl xs)"
abbreviation "tl2 xs == tl(tl xs)"
```

The formal definition for single instructions is a similar judgement to the small-step semantics of the source language, and we introduce similar syntax for it. Each introduction rule implements the effect of the corresponding instruction as described above. The program counter is `i`, usually incremented by `1`, apart from the jump instructions. Variables are loaded from and stored into the variable state `s` with function application and function update. The stack `stk` uses standard list constructs `#`, `hd`, `tl`, as well as our new `hd2` and `tl2`.

```
inductive iexec1 :: "instr ⇒ config ⇒ config ⇒ bool"
    ("(_/ ⊢i (_ →/ _))" [50,0,0] 50)
where
"LOADI n ⊢i (i,s,stk) → (i+1,s, n#stk)" |
"LOAD x  ⊢i (i,s,stk) → (i+1,s, s x # stk)" |
"ADD     ⊢i (i,s,stk) → (i+1,s, (hd2 stk + hd stk) # tl2 stk)" |
"STORE n ⊢i (i,s,stk) → (i+1,s(n := hd stk),tl stk)" |
"JMP n   ⊢i (i,s,stk) → (i+1+n,s,stk)" |
"JMPLESS n ⊢i (i,s,stk) →
   (if hd2 stk < hd stk then i+1+n else i+1,s,tl2 stk)" |
"JMPGE n ⊢i (i,s,stk) →
   (if hd2 stk >= hd stk then i+1+n else i+1,s,tl2 stk)"
```

The next level up, a single execution step of a program, which we model as an abstract list of instructions, selects the instruction the program counter points to and uses the `iexec1` judgment to execute it. For execution to be well-defined, we additionally check if the pc does actually point to a valid location in the list.

Modelling the program as a list of instructions is another abstraction we introduce to a real low-level machine. It corresponds roughly to the level of abstraction of the JVM. A real CPU would implement a von-Neumann machine, which adds fetching and decoding of instructions to the execution cycle. The main difference is that our model would not admit self-modifying programs which is not necessary for IMP.

```
inductive
  exec1 :: "instr list ⇒ config ⇒ config ⇒ bool"
    ("(_/ ⊢ (_ →/ _))" [50,0,0] 50) where
 "⟦ P!!i ⊢i (i,s,stk) → c'; 0 ≤ i; i < isize P ⟧ ⟹ P ⊢ (i,s,stk) → c'"
```

The last level is the lifting from single step execution to multiple steps using the standard reflexive, transitive closure definition that we already used for the small-step semantics of the source language.

```
inductive exec :: "instr list ⇒ config ⇒ config ⇒ bool"
```

```
  ("_/ ⊢ (_ →*/ _)" 50) where
refl: "P ⊢ c →* c" |
step: "P ⊢ c → c' ⟹ P ⊢ c' →* c'' ⟹ P ⊢ c →* c''"
```

This concludes our definition of the machine and its semantics. We can now try out a simple example:

```
values
  "{(i,map t [''x'',''y''],stk) | i t stk.
    [LOAD ''y'', STORE ''x''] ⊢
    (0, [''x'' → 3, ''y'' → 4], []) →* (i,t,stk)}"
```

The result is the following sequence of machine configurations: `{(0, [3, 4], []), (1, [3, 4], [4]), (2, [4, 4], [])}`.

## *4.3. Verification infrastructure*

The compiler proof is more involved than the short proofs we have seen so far. We will need a lot more infrastructure setup and small technical lemmas before we get to the actual problem itself. We have seen enough concepts at this stage that we can present the whole machine semantics in one go in this section instead of splitting it and its setup over multiple sections as we did for the source language.

When constructing such a proof, one usually starts with a reasonably small infrastructure setup and then grows the infrastructure as needed during the proof.

We begin with the constant `iexec1` and will work our way upwards through the 3 semantic layers of the machine. The goal is to be able to execute machine programs symbolically as far as possible using Isabelle's proof tools. We will then use this ability in the compiler proof to assemble the various program parts.

The rules of `iexec1` can easily be transformed into equations. In fact, we could have defined the judgement using `fun` or `definition` instead of an inductive predicate. The reason for using `inductive` was better readability and the fact that it fits more naturally into the general schema we used for other judgment definitions. To derive the equational form, we first set up automatic rule inversion, and then use it to prove the rest automatically.

Large parts of the compiler correctness argument are about the execution of code that is embedded in larger programs. For this purpose we show that execution results are preserved if we append additional code to the left or right of a program.

Appending at the right side is easy to state and solved automatically after rule induction:

```
lemma exec_appendR: "P ⊢ c →* c' ⟹ P@P' ⊢ c →* c'"
```

Appending additional code on the left side requires a moment of thought: the lemma statement will need to update the program counter and shift the execution by `size P'` to the right if we append `P'` on the left.

For this to go through, we need the same statement on the single-instruction level. Formulating this statement we discover that single-instruction execution at `iexec1` is actually independent of the program counter:

```
lemma iexec_shift [simp]:
  "(X ⊢i (n+i,s,stk) → (n+i',s',stk')) =
   (X ⊢i (i,s,stk) → (i',s',stk'))"
```

The result is then easily lifted to the program level, and afterwards by induction to the execution of multiple steps.

```
lemma exec1_appendL:
  "P ⊢ (i,s,stk) → (i',s',stk') ⟹
   P' @ P ⊢ (isize(P')+i,s,stk) → (isize(P')+i',s',stk')"
lemma exec_appendL:
 "P ⊢ (i,s,stk) →* (i',s',stk')  ⟹
  P' @ P ⊢ (isize(P')+i,s,stk) →* (isize(P')+i',s',stk')"
```

We then specialise the above lemmas to enable automatic proofs of `P ⊢ c →* c'` where `P` is a mixture of concrete instructions and pieces of code of which we already know how they execute (by induction), combined by `@` and `#`. Backward jumps are not supported. For details we refer to the proof scripts.

## *4.4. Compilation*

We are now ready to define the actual compiler. We begin with arithmetic expressions which are very easy to compile into a stack machine. A constant maps to `LOADI`, a variable maps to `LOAD`, and addition maps to `ADD`.

```
fun acomp :: "aexp ⇒ instr list" where
"acomp (N n) = [LOADI n]" |
"acomp (V x) = [LOAD x]" |
"acomp (Plus a1 a2) = acomp a1 @ acomp a2 @ [ADD]"
```

The idea is that the compiled code leaves the result of the computation on the top of the stack and otherwise does not change anything. This is already one direction of the correctness statement for arithmetic expressions and easily written down formally. The proof is by induction on the structure of expressions.

```
lemma acomp_correct[intro]:
  "acomp a ⊢ (0,s,stk) →* (isize(acomp a),s,aval a s#stk)"
```

The form of the lemma is suitable for an introduction rule to use automatically in the following proofs.

Compiling boolean operations has a different correctness statement. These expressions are used in if-then-else and while-statements, and what we expect the compiled code to do is to give us one exit program counter for the `True` case and another for the `False` case and apart from that leave the state and stack unchanged. We chose the pc after the expression code as the first exit and give the expression compiler an additional parameter `n` for the distance to the jump target that is our the second exit. We add another parameter `c` that tells us which exit to use in the `True` and `False` case. If `c` is `False`, then the `True` case will be the first exit.

This enables us to perform a small bit of optimisation: boolean constants do not really need to execute any code, they either compile to nothing or to a jump to the second exit, depending on the value of *c*. The *Not* case simply inverts *c* to swap exits. The *And* case performs shortcut evaluation: if *b1* evaluates to *False* we exit the whole expression taking the exit *c* demands. Otherwise we continue evaluation with *b2*. The *Less* operator uses the *acomp* compiler for *a1* and *a2* and then selects the jmp/compare instruction according to *c*.

```
fun bcomp :: "bexp ⇒ bool ⇒ int ⇒ instr list" where
"bcomp (B bv) c n = (if bv=c then [JMP n] else [])" |
"bcomp (Not b) c n = bcomp b (¬c) n" |
"bcomp (And b1 b2) c n =
 (let cb2 = bcomp b2 c n;
        m = (if c then isize cb2 else isize cb2+n);
      cb1 = bcomp b1 False m
  in cb1 @ cb2)" |
"bcomp (Less a1 a2) c n =
 acomp a1 @ acomp a2 @ (if c then [JMPLESS n] else [JMPGE n])"
```

This optimisation basically entails a form of primitive constant folding in boolean expressions. The expression *And (B True) (B True)* for instance with *c = False* compiles to the empty list.

A more usual case will translate to a series of load and jump instructions. A test case in Isabelle:

```
value "bcomp (And (Less (V ''x'') (V ''y''))
                  (Not(Less (V ''u'') (V ''v''))))
             False 3"
```

The result of this evaluation is *[LOAD ''x'', LOAD ''y'', JMPGE 6, LOAD ''u'', LOAD ''v'', JMPLESS 3]*.

The correctness statement for boolean expressions encapsulates the intention we had when writing *bcomp*: stack and state remain unchanged and the program counter indicates if the expression evaluated to *True* or *False*. If *c = False* then we end at *isize (bcomp b c n)* in the *True* case and *isize (bcomp b c n) + n* in the *False* case. If *c = True* it is the other way around. This statement only really makes sense for forward jumps, hence the precondition on *n*. We could generalise slightly and only exclude exits that lead back into the code of *bcomp b c n*, but this form is slightly easier to work with. As for the arithmetic case, we tell Isabelle to use this correctness statement automatically as an introduction rule.

```
lemma bcomp_correct[intro]:
"0 ≤ n ⟹ bcomp b c n ⊢ (0, s, stk) →*
          (isize(bcomp b c n) + (if c = bval b s then n else 0), s, stk)"
```

The proof of this lemma is by structural induction on the expression *b*. The constant and *Less* cases are solved automatically. For *Not* Isabelle needs some hand-holding by suitably instantiating the induction hypothesis; similarly in the *And* case.

With both kinds of expressions done, we can now proceed to compiling programs. The idea is to generate a program that will perform the same state transformation on the state *s* as the source, that will result in nothing new on the stack (but may push and consume intermediate values for expressions), and that will always end with program counter `size (ccomp c)`.

Relying on this condition recursively, this is surprisingly straightforward to achieve: *SKIP* compiles to the empty list and assignment compiles the expression, then stores the result into the state. The if-then-else statement compiles into the expression part and jumps to the corresponding branch. Linearising the tree structure of the abstract source syntax, we jump over the compiled *c2* to the end of the code when the *True* branch *c1* has finished executing. While loops compile into the condition followed by the body and a backwards jump to the beginning of the condition. The exit branch of the condition jumps beyond the backward branch directly to the end of the code.

The formal definition in Isabelle follows.

```
fun ccomp :: "com ⇒ instr list" where
"ccomp SKIP = []" |
"ccomp (x ::= a) = acomp a @ [STORE x]" |
"ccomp (c₁;c₂) = ccomp c₁ @ ccomp c₂" |
"ccomp (IF b THEN c₁ ELSE c₂) =
  (let cc₁ = ccomp c₁;
       cc₂ = ccomp c₂;
       cb = bcomp b False (isize cc₁ + 1)
   in cb @ cc₁ @ JMP (isize cc₂) # cc₂)" |
"ccomp (WHILE b DO c) =
 (let cc = ccomp c; cb = bcomp b False (isize cc + 1)
  in cb @ cc @ [JMP (- (isize cb + isize cc + 1))])"
```

Since everything in this definition is within the executable fragment of Isabelle/HOL, we can inspect the results of compiler runs:

```
value "ccomp (WHILE Less (V ''u'') (N 1)
              DO (''u'' ::= Plus (V ''u'') (N 1)))"
```

evaluates to `[LOAD ''u'', LOADI 1, JMPGE 5, LOAD ''u'', LOADI 1, ADD, STORE ''u'', JMP -8]`.

### 4.5. *Preservation of semantics*

Since we had already completed our infrastructure setup before we started the definition of the compiler, we are now ready to state the first direction of the correctness statement and prove it.

The idea of the compiler was to achieve the same state change as the source program, so we formulate that if `(c, s) ⇒ t` we can execute the compiled *c* starting in *0* to its completion at `isize (ccomp c)` and achieve the same state change from *s* to *t*, leaving any existing stack unchanged.

Since we conveniently have a statement about a big-step execution in the assumptions, we can proceed with rule induction on this statement. We will go into the cases into more detail below.

```
lemma ccomp_bigstep:
  "(c,s) ⇒ t ⟹ ccomp c ⊢ (0,s,stk) →* (isize(ccomp c),t,stk)"
```

As mentioned, the main structure of the proof is a rule induction on the big-step judgment. With the correctness statements for expressions used implicitly and our specifically designed composition lemmas, the proof is in large parts automatic. The *SKIP*, while-false, and if-then-else cases are solved fully automatically. The assignment case only needs a little help to align the two function updates that Isabelle manages to derive for both sides.

In the semicolon case, we instantiate the two induction hypotheses for *c1* and *c2* appropriately and put both of them into the full *ccomp c1 @ ccomp c2* context. Transitivity then gives us the execution of the whole statement.

In the while-true case, we begin the proof by observing that if the condition is true, and by induction hypothesis the body *c* executes from state *s1* to *s2*, the whole compiled program will execute from pc *0* in *s1* to the position of the backwards jump instruction and state *s2*. Moreover, we observe that the jump instruction will transport us back to the beginning of the loop in state *s2*. By induction hypothesis again, we know that the loop started in *s2* and pc *0* will terminate correctly in *s3*. Putting these three steps together by transitivity, we can conclude the case and thereby the whole proof.

This direction of the correctness proof is so convenient, because rule induction on the big-step semantics is such a powerful proof principle and it is easy in our machine to compose smaller executions into larger ones.

The real correctness statement we are after is the if-and-only-if

$$\begin{gathered}(c,\ s) \Rightarrow t \longleftrightarrow \\ ccomp\ c \vdash (0,\ s,\ stk) \rightarrow\!*\ (isize\ (ccomp\ c),\ t,\ stk)\end{gathered}$$

and the other direction is much less convenient. So much so that it may be worth thinking about if this direction is really necessary at all. Do we not already know enough? For any execution of the source program we know that the machine will produce the same execution. Unfortunately if we want to transport program correctness statements from the source to the machine level, this is precisely the wrong direction:

If we have proved a standard partial program correctness property of the form $\forall s\ t.\ (c,\ s) \Rightarrow t \longrightarrow P\ s \longrightarrow Q\ t$ as it would come out of a Hoare triple *{P} c {Q}*, and we would like a similar Hoare-triple to be true about the compiled program, then we will have to assume a machine execution and show the existence of a corresponding source execution to satisfy the first premise of the implication. This is precisely direction two of our proof!

It is worth pointing out that in a deterministic language like IMP, the second direction reduces to preserving termination: if the machine program terminates, so must the source program. If that was the case, we could conclude that started in *s* the source must terminate in some *t'*. Using the first direction of the compiler proof and by determinism of the machine language, we could then conclude that *t' = t* and that therefore the source execution was already the right one. Unfortunately, showing that machine-level termination implies source-level termination is not much easier than showing the second direction of our compiler proof directly, and so we will not take that path here. Instead, we will prove the second direction directly.

## *4.6. Compiler Correctness, 2nd direction*

In this section, we will prove the second direction of compiler correctness: *ccomp c ⊢ (0, s, stk) →* (isize (ccomp c), t, stk') ⟹ (c, s) ⇒ t*.

As argued above, this direction is more technically involved than the first one. The main reason for this is the lack of a nice structural induction principle on the machine side. Since rule induction is not applicable, we have only two further induction principles left in the arsenal we have seen so far: structural induction on the command *c* or induction on the length of the →* execution. Neither is strong enough on its own. Trying structural induction, we get into the usual problem for the while-case that our induction hypothesis only talks about the execution of the body *c*, but not the rest of the loop. Induction on the length of the execution is more promising at first, but does not work either. Consider the semicolon case: for *ccomp c1 @ ccomp2 ⊢ (0, s, stk) →* (isize (ccomp (c1; c2)), t, stk')* the first command *c1* could be *SKIP* and *ccomp SKIP* therefore the empty list. This means the statement collapses to *ccomp2 ⊢ (0, s, stk) →* (isize (ccomp c2), t, stk')* and we have no shorter execution that an induction hypothesis could apply to.

The solution is to do both: an outside structural induction on *c* which will take care of all cases but *WHILE*, and then a nested, inner induction on the length of the execution for the while-case. This takes care of the general proof structure. We will now merely be left with the problem of decomposing larger machine executions into smaller ones.

Intuitively, this is an easy problem. Consider *cs1 @ cs2 ⊢ s →* s'*. It seems obvious that there must be some *s''* such that *cs1 ⊢ s →* s''* and *cs2 ⊢ s'' →* s'*, but this is by far not true for all possible *cs1* and *cs2*, or even *s* and *s''*. For instance, execution may be jumping between *cs1* and *cs2* continuously, and neither may make sense in isolation.

A large part of the formalisation below will be concerned with identifying the useful cases where this decomposition is possible and instantiating it suitably to be applied in the main induction that is the compiler proof. The key idea in this part is to identify the possible exit program counters for a given list of instructions. In general, instruction lists may jump anywhere, but it turns out that code produced by the compiler is particularly well behaved. Code compiled from a statement *c* will always run to *pc = isize (ccomp c)* and any jump targets will be only inside the generated code, not outside.

The final new concept we will need for the proof is a machine execution of explicit length *n*. This will enable us to do induction on this *n*, showing that a property *P* holds for any *n* if it holds for any *m < n*. This saves us from having to splice off single execution steps at a time and gives us more flexibility in decomposing the large execution in the nested induction for the while-case.

In the following sections, we first define the new concepts outlined above, establish their basic properties and main lemmas, and finally put them together in the nested induction over IMP statements and the length of machine executions.

### *4.6.1. Definitions*

An execution of length *n* is easily defined using primitive recursion. It's role is to replace the transitive closure of *exec1* that we used before.

```
primrec
  exec_n :: "instr list ⇒ config ⇒ nat ⇒ config ⇒ bool"
  ("_/ ⊢ (_ →^_/ _)" [65,0,55,55] 55)
where
  "P ⊢ c →^0 c' = (c'=c)" |
  "P ⊢ c →^(Suc n) c'' = (∃c'. (P ⊢ c → c') ∧ P ⊢ c' →^n c'')"
```

The next concept is the definition of the possible pc exits of a given instruction sequence. We build up this definition in three steps: first, we define `isuccs`, the possible successor pcs of a given instruction at position `n`, then we lift this to `succs P n` which gives us the successor pcs of an instruction sequence `P` which itself may be embedded in a larger program at position `n`, and finally we subtract the set `{0..isize P}` from `succs P 0` to arrive at the possible exits of `P`.

The set of possible successors of an instruction at position `n` is usually just the singleton `{n + 1}`, only for jump instructions we get a different successor, and we get two possibilities for conditional jumps.

```
definition
  "isuccs i n ≡ case i of
     JMP j ⇒ {n + 1 + j}
   | JMPLESS j ⇒ {n + 1 + j, n + 1}
   | JMPGE j ⇒ {n + 1 + j, n + 1}
   | _ ⇒ {n +1}"
```

The possible successors pcs of an instruction list is the union of all instruction successors. The set comprehension below prefers a closed term over a recursion and uses the list indices to provide the position of the instruction to `isuccs`.

```
definition
  "succs P n = {s. ∃i. 0 ≤ i ∧ i < isize P ∧ s ∈ isuccs (P!!i) (n+i)}"
```

With the possible successors defined, the set of possible exits is straightforward.

```
definition "exits P = succs P 0 - {0..< isize P}"
```

### *4.6.2. Basic properties of* `exec_n`

We begin the property sections by relating our new `exec_n` with the existing judgement `exec` and by setting up some automation for `exec_n`.

The equivalence is proved by induction on `n` in one direction and rule induction in the other.

```
lemma exec_eq_exec_n: "(P ⊢ c →* c') = (∃n. P ⊢ c →^n c')"
```

As in the easy direction of the compiler proof, we design a set of lemmas that we can use to symbolically execute a mixture of concrete instructions and unknown parts of the instruction sequence.

Previously, we designed a set of introduction rules, because we needed to compose smaller executions into larger ones in the conclusion of the goal. Here, we will need to decompose larger executions into smaller ones, reasoning forward from known assumptions. We therefore provide conditional rewrite rules instead, which can be used in both, conclusion and assumptions.

### *4.6.3. Basic properties of* `succs`

With the basic facts about execution set up, we can turn our attention the successor and exit functions. Our goal is to derive lemmas about the possible exits for the results of *acomp*, *bcomp*, and *ccomp*. The main proofs for these results will be by induction on the expression or statement to be compiled. The key lemmas are about the constructor and append case of *succs*.

```
lemma succs_Cons:
  "succs (x#xs) n = isuccs x n ∪ succs xs (1+n)"
lemma succs_append [simp]:
  "succs (xs @ ys) n = succs xs n ∪ succs ys (n + isize xs)"
```

The first of the *succs* lemmas for *acomp*, *bcomp*, and *ccomp* is pleasantly well behaved, because *acomp* does not contain any jumps.

```
lemma acomp_succs [simp]:
  "succs (acomp a) n = {n + 1 .. n + isize (acomp a)}"
```

The *exits* for *acomp* follow easily.

```
lemma exits_acomp [simp]: "exits (acomp a) = {isize (acomp a)}"
```

Compilation for boolean expressions is less well behaved, but still almost automatic. We need to put some care into formulating the statement: Firstly, to exclude trivial spurious cases, the jump target parameter of *bcomp* should lead to a forward jump. As for the compilation, we could try for a more general statement, but we are not going to need it. Secondly, we are not able to provide a nice, short equational form for *bcomp*. The main idea is that *bcomp* has two possible *exits*, one for the true-case, one for the false-case, and therefore corresponding successor pcs. However, the result of *bcomp* may be the empty list, for instance, and therefore statically have no successors (the compilation will always just fall through into the true-case). More generally, the partial constant folding optimisation that *bcomp* performs may statically exclude one or both of the possible exits. Instead of trying to precisely describe each of these cases for stating an equation, we settle for a less precise statement and just provide an upper bound for the possible successors of *bcomp*.

The proof is mostly automatic. Isabelle merely needs some hand holding in the *And*-case where we tell it how to apply the induction hypothesis.

```
lemma bcomp_succs:
  "0 ≤ i ⟹
  succs (bcomp b c i) n ⊆ {n .. n + isize (bcomp b c i)}
                          ∪ {n + i + isize (bcomp b c i)}"
```

Again, we can derive the *exits* directly from the *succs* lemma.

```
lemma bcomp_exits:
```

```
"0 ≤ i ⟹
exits (bcomp b c i) ⊆ {isize (bcomp b c i), i + isize (bcomp b c i)}"
```

The remaining compilation function is *ccomp*. Our intuition is that the exits are *{isize (ccomp c)}*. Since we can arrive at empty compilations, though, for instance because of *SKIP*, and since we only have an upper bound for boolean expressions, we again only formulate an upper bound for *succs*.

The proof is still largely automatic for each case of the induction, we mainly give some help in how to apply the induction hypothesis.

```
lemma ccomp_succs: "succs (ccomp c) n ⊆ {n..n + isize (ccomp c)}"
```

From *succs*, the *exits* again derive easily.

```
lemma ccomp_exits: "exits (ccomp c) ⊆ {isize (ccomp c)}"
```

### *4.6.4. Splitting up machine executions*

The main purpose of this section is to derive lemmas for decomposing larger executions into smaller ones. There are two main results: The first fully executes a sub-sequence of machine instructions and lifts it out of its execution context. The second result can deal with a partial execution of a sub-sequence, but only drops its left context under the condition that there are no jumps into the left context.

For the first result, we need a connection between our definition of exits and successors and the semantics of machine executions, a correctness statement for *succs* if you want. The idea is that each execution step must yield a new pc that is within the set predicted by *succs*.

```
lemma succs_iexec1:
  "P!!i ⊢i (i,s,stk) → c' ⟹ 0 ≤ i ⟹ i < isize P ⟹
  fst c' ∈ succs P 0"
```

For the first decomposition lemma, we assume an execution in the program *P @ c @ P'* where the initial program counter is within *c* and the final program counter *j* is not within *c*. It must then be the case that there exists an execution of only *c* without additional context, started in the same state, ending in an intermediate state *s''*, and ending with a pc in the exits of *c*. This execution must be able to conclude to the same final state from *s''* in the rest of the program, possibly including *c* again. The lengths of these two executions must add up to the length of the original execution.

The proof is by induction on *n*.

```
lemma exec_n_split:
  shows "⟦ P @ c @ P' ⊢ (isize P + i, s) →^n (j, s');
           0 ≤ i; i < isize c; j ∉ {isize P ..< isize P + isize c} ⟧ ⟹
         ∃s'' i' k m.
                  c ⊢ (i, s) →^k (i', s'') ∧
                  i' ∈ exits c ∧
                  P @ c @ P' ⊢ (isize P + i', s'') →^m (j, s') ∧
                  n = k + m"
```

The base case is simple. The step case executes one execution step, and lifts it to the only-*c* context with an additional lemma. This is possible because we know the program counter to be within *c*. We then perform a case distinction: either the execution of *c* is not finished yet (*j0 ∈ {0..<isize c}*), then we apply the induction hypothesis; or the execution of *c* has finished, and therefore the pc is within the exits of *c* and the rest of the execution from the step case will bring us to the final state.

We can instantiate the general form of this lemma to just drop a right context. This situation occurs frequently in the main proof.

The second decomposition result mentioned above can be used to drop a left context. We can relax the condition from a complete to a partial execution of the right side, but we are adding a restriction on the exits of the executing part that prevents backward jumps out of the code sequence. This tells us that the left context cannot interfere with the execution of the right context. The proof is again by induction on the length of execution.

```
lemma exec_n_drop_left:
  "⟦ P @ P' ⊢ (i, s, stk) →^k (n, s', stk');
     isize P ≤ i; exits P' ⊆ {0..} ⟧ ⟹
     P' ⊢ (i - isize P, s, stk) →^k (n - isize P, s', stk')"
```

The base case is trivial. In the step case, we again start from a single step and the rest of the execution. We lift the single step, compute the new program counter, and using the *exits* assumption conclude that the program counter is still within *P2*, which means we can apply the induction hypothesis to get the rest of the executio and conclude the lemma.

From this general lemma, we can easily derive the Cons case where we drop an individual instruction from the head of the list, which will again be a frequent case in the compiler proof.

Both lemmas taken together, dropping left and right contexts, can be used to fully decompose an append of two instruction sequences *P @ P'* into isolated executions, one for *P* and one for *P'*. This case is useful for arithmetic expressions, and the semicolon case of *ccomp*, but it only works if the execution of the left side seamlessly leads over into execution of the right, i.e. if the exits of *P* are just *{isize P}*. This is true for the results *acomp* and *ccomp*, and we call such instruction sequences *closed*:

```
definition "closed P ⟷ exits P ⊆ {isize P}"

lemma ccomp_closed [simp, intro!]: "closed (ccomp c)"
lemma acomp_closed [simp, intro!]: "closed (acomp c)"
lemma exec_n_split_full:
  assumes exec: "P @ P' ⊢ (0,s,stk) →^k (j, s', stk')"
  assumes P: "isize P ≤ j"
  assumes closed: "closed P"
  assumes exits: "exits P' ⊆ {0..}"
  shows "∃k1 k2 s'' stk''.
            P ⊢ (0,s,stk) →^k1 (isize P, s'', stk'') ∧
            P' ⊢ (0,s'',stk'') →^k2 (j - isize P, s', stk')"
```

### *4.6.5. Correctness theorem*

In this section, we prove the main theorem of this part, the second direction of the compiler proof. As before, we work our way up from *acomp* over *bcomp* to *ccomp*.

The correctness statement for *acomp* says that any complete execution of an arithmetic expression will leave the result value on the otherwise unchanged stack, and will not change the state of variables.

The proof is by induction on the expression, and most cases are solved automatically using our lemma set for symbolic execution. In the *Plus*-case, we decompose the execution manually, apply the induction hypothesis for the parts, and combine the results symbolically executing the *ADD* instruction.

```
lemma acomp_exec_n [dest!]:
  "acomp a ⊢ (0,s,stk) →^n (isize (acomp a),s',stk') ⟹
  s' = s ∧ stk' = aval a s#stk"
```

The correctness statement for *bcomp* is more verbose than the *acomp* version, because we need to take the two different exits into account. We demand that the expression has been fully executed, i.e. that the exit program counter *i* is not within the compiled expression any more. With the usual exclusion of backward jumps in *bcomp* we can then conclude that the exit program counter corresponds precisely with the value of the evaluation of the source expression. Stack and state remain unchanged.

As in the arithmetic case, we prove the lemma by induction on the expression. The *And* case is the only interesting one. We first split the execution into one for the left operand *b1* and one for the right operand *b1* with our splitting lemmas above. We then determine by induction hypothesis that stack and state did not change for *b1* and that the program counter will either exit directly, in which case we are done, or it will point to the instruction sequence for *b1*, in which case we apply the second induction hypothesis to conclude the case and the lemma.

```
lemma bcomp_exec_n [dest]:
  "⟦ bcomp b c j ⊢ (0, s, stk) →^n (i, s', stk');
     isize (bcomp b c j) ≤ i; 0 ≤ j ⟧ ⟹
  i = isize(bcomp b c j) + (if c = bval b s then j else 0) ∧
  s' = s ∧ stk' = stk"
```

We are now ready to tackle the main lemma. Its statement is what we set out for initially, merely phrased for *exec_n* instead pf *exec*. As was the plan, the main proof is an outer induction on *c*, and a nested induction on *n* for the while-case.

```
lemma ccomp_exec_n:
  "ccomp c ⊢ (0,s,stk) →^n (isize(ccomp c),t,stk')
   ⟹ (c,s) ⇒ t ∧ stk'=stk"
```

The first three cases of the structural induction are automatic with the lemmas that we have built up so far. The if-then-else case is more verbose, but not much more complex. It consists mainly of splitting of the evaluation of the boolean expression, case distinction and applying the induction hypothesis for the statement on *c1* and *c2*.

The while-case unsurprisingly is the most involved. We gain the induction hypothesis that *ccomp c* will execute correctly, but we need the induction on *n* to reason about the rest of the loop. We start with a case distinction on the loop condition. If this is false, we have jumped to the end of the loop and know that state and stack did not change. We can construct the same execution on the source level with the While-False rule.

The case where the loop condition is true is the interesting one. Similarly to if-then-else, we start by splitting of the execution of `bcomp b` so we can apply the `bcomp` correctness lemma. Since we already now the condition evaluates to true, the exit program counter must be the start of `ccomp c`. Now we would like split off the execution of `ccomp c` to apply the outer induction hypothesis and gain information about the execution of the source-level `c`. This is only possible if `ccomp c` is non-empty, so we first perform a case distinction. If it is empty, then evaluation will be trivial (SKIP), and we can symbolically execute the jump instruction to the beginning of the loop, and by induction hypothesis on the now at least one step smaller execution of the rest of the while loop conclude the case. Note that this case is actually vacuous: the loop will not terminate and therefore our main assumption is false. Establishing that contradiction is more difficult than just applying the induction hypothesis, though. The second case, where `ccomp c ≠ []` goes the more normal route: split off the execution of `ccomp c`, apply the outer induction hypothesis to get the source-level execution of `c`, symbolically execute the backwards jump and, because the rest of the execution must be less steps than the whole, get the rest of the source-level while execution by induction hypothesis. Together with the fact that the loop condition was true, we can use the While-True rule from the big-step semantics to put these parts together and conclude the final correctness statement.

Converting `exec_n` back into `exec` we get what we set out to prove:

```
theorem ccomp_exec:
  "ccomp c ⊢ (0,s,stk) →* (isize(ccomp c),t,stk') ⟹ (c,s) ⇒ t"
```

As a corollary, we get final soundness statement of our compiler.

```
corollary ccomp_sound:
  "ccomp c ⊢ (0,s,stk) →* (isize(ccomp c),t,stk)  ⟷  (c,s) ⇒ t"
```

## 5. Summary

These lecture notes have presented the formalisation of a simple programming language in Isabelle/HOL together with a number of increasingly complex applications.

The aim of the presentation was to show that using an interactive theorem prover for programming language semantics brings clear benefits in tool assistance, experimentation, executability, and even readability for presentation in publications.

## References

[1] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.*, 28(4):619–695, 2006.

[2] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54. ACM, 2006.

[3] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[4] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2/3):167–188, 1996.