

Applications of Interactive Proof to Data Flow Analysis and Security

Gerwin Klein^a, Tobias Nipkow^b

^a *NICTA & UNSW, Sydney, Australia*

^b *Technische Universität München, Germany*

Abstract. We show how to formalise a small imperative programming language in the theorem prover Isabelle/HOL, how to define its semantics, and how to prove properties about the language, its type systems, and a number of data flow analyses.

The emphasis is not on formalising a complex language deeply, but to teach a number of formalisation techniques and proof strategies using simple examples. For this purpose, we cover a basic type system with type safety proof, more complex security type systems, also with soundness proofs, and different kinds of data flow analyses, in particular definite initialisation analysis and constant propagation, again with correctness proofs.

Keywords. Semantics, Security, Data-Flow Analysis, Isabelle

1. Introduction

These notes present the formalisation of a small imperative programming language in the theorem prover Isabelle/HOL [13] together with applications to data flow analysis and security type systems.

We assume that the reader is familiar with Isabelle/HOL and its basic notation, which is close to standard mathematics and functional languages. In particular, we assume familiarity with the concepts of recursive data types, inductive and recursive definitions, and proofs by rule induction and structural induction. These are covered in a separate text [13].

2. IMP: A Simple Imperative Language

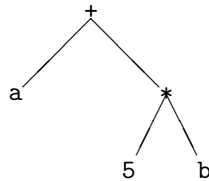
This section defines a minimalistic imperative programming language called **IMP**. We introduce the concepts of expressions and commands with their abstract syntax, and use them to illustrate two styles of defining the semantics of a programming language: big-step and small-step operational semantics. Our first larger theorem about IMP will be the equivalence of these two definitions of its semantics. As a smaller concrete example, we will apply our semantics to the concept of program equivalence.

2.1. Arithmetic expressions

We start by defining syntax and semantics for arithmetic and boolean expressions.

2.1.1. Syntax

Programming languages have both a concrete and an abstract syntax. **Concrete syntax** means strings. For example, "a + 5 * b" is an arithmetic expression given as a string. The concrete syntax of a language is usually defined by a context free grammar. The expression "a + 5 * b" can also be viewed as the following tree:



The tree immediately reveals the nested structure of the object and is the right level for analysing and manipulating expressions. Linear strings are more compact than two-dimensional trees, which is why they are used for reading and writing programs. But the first thing a compiler, or rather its parser will do is to convert the string into a tree for further processing. Now we are at the level of **abstract syntax** and these trees are **abstract syntax trees**. To regain the advantages of the linear string notation we write our abstract syntax trees as strings with parentheses to indicate the nesting (and with identifiers instead of the symbols + and *), for example like this: *Plus a (Times 5 b)*. Now we have arrived at ordinary terms like we have used them all along. More precisely, these terms are over some datatype that defines the abstract syntax of the language. Our little language of arithmetic expressions is defined by the datatype *aexp*:

```
type-synonym vname = string  
datatype aexp = N int | V vname | Plus aexp aexp
```

where *int* is the predefined type of integers and *vname* stands for variable name. Isabelle strings require two single quotes on both ends, for example *"abc"*. The intended meaning of the three constructors is as follows: *N* represents numbers, i.e. constants, *V* represents variables, and *Plus* represents addition. The following examples illustrate the intended correspondence:

Concrete	Abstract
5	<i>N 5</i>
x	<i>V "x"</i>
x + y	<i>Plus (V "x") (V "y")</i>
2 + (z + 3)	<i>Plus (N 2) (Plus (V "z") (N 3))</i>

It is important to understand that so far we have only defined syntax, not semantics! Although the binary operation is called *Plus*, this is merely a suggestive name and does not imply that it behaves like addition. For example, *Plus (N 0)*

$(N\ 0) \neq N\ 0$, although you may think of them as semantically equivalent—but syntactically they are not.

Datatype *axp* is intentionally minimal to concentrate on the essentials. Further operators can be added as desired. However, as we shall discuss below, not all operators are as well-behaved as addition.

2.1.2. Semantics

The semantics, or meaning of an expression is its value. But what is the value of $x+1$? The value of an expression with variables depends on the values of its variables. The value of all variables is recorded in the (program) **state**. The state is a function from variable names to values.

type-synonym $val = int$
type-synonym $state = vname \Rightarrow val$

In our little toy language, the only values are integers.

The value of an arithmetic expression is computed like this:

fun $aval :: axp \Rightarrow state \Rightarrow val$ where
 $aval\ (N\ n)\ s = n$ |
 $aval\ (V\ x)\ s = s\ x$ |
 $aval\ (Plus\ a_1\ a_2)\ s = aval\ a_1\ s + aval\ a_2\ s$

Function *aval* carries around a state and is defined by recursion over the form of the expression. Numbers evaluate to themselves, variables to their value in the state, and addition is evaluated recursively. Here is a simple example:

value $aval\ (Plus\ (N\ 3)\ (V\ "x"))\ (\lambda x.\ 0)$

returns 3. However, we would like to be able to write down more interesting states than $\lambda x.\ 0$ easily. This is where function update comes in.

To update the state, that is, change the value of some variable name, the generic function update notation $f(a := b)$ is used: the result is the same as f , except that it maps a to b :

$f(a := b) = (\lambda x.\ \text{if } x = a \text{ then } b \text{ else } f\ x)$

This operator allows us to write down concrete states in a readable fashion. Starting from the state that is 0 everywhere, we can update it to map certain variables to given values. For example, $((\lambda x.\ 0)\ ("x" := 7))\ ("y" := 3)$ maps $"x"$ to 7, $"y"$ to 3 and all other variable names to 0. Below we employ the following more compact notation

$\langle "x" := 7, "y" := 3 \rangle$

which works for any number of variables, even for none: $\langle \rangle$ is syntactic sugar for $\lambda x.\ 0$.

It would be easy to add subtraction and multiplication to *axp* and extend *aval* accordingly. However, not all operators are as well-behaved: division by zero raises an exception and C's `++` changes the state. Neither exceptions nor side-effects can be supported by an evaluation function of the simple type $axp \Rightarrow state \Rightarrow val$; the return type has to be more complicated.

2.2. Boolean expressions

In keeping with our minimalist philosophy, our boolean expressions contain only the bare essentials: boolean constants, negation, conjunction and comparison of arithmetic expressions for less-than:

```
datatype bexp = Bc bool | Not bexp | And bexp bexp | Less aexp aexp
```

Note that there are no boolean variables in this language. Other operators like disjunction and equality are easily expressed in terms of the basic ones.

Evaluation of boolean expressions is again by recursion over the abstract syntax. In the *Less* case, we switch to *aval*:

```
fun bval :: bexp  $\Rightarrow$  state  $\Rightarrow$  bool where  
bval (Bc v) s = v |  
bval (Not b) s = ( $\neg$  bval b s) |  
bval (And b1 b2) s = (bval b1 s  $\wedge$  bval b2 s) |  
bval (Less a1 a2) s = (aval a1 s < aval a2 s)
```

2.3. IMP Commands

Having defined expressions and their evaluation, we can now move on to the commands. Our language is a minimal Turing-complete *WHILE* language. It has assignment, sequential composition (semicolon), conditionals (*IF*), and *WHILE*. To be able to express other syntactic forms, such as an *IF* without an *ELSE* branch, we also include the *SKIP* command that does nothing. The right-hand side of variable assignments uses the arithmetic expressions that we have defined above, and similarly, the conditions in *IF* and *WHILE* will take boolean expressions. A program is then simply one, possibly complex, command in this language. The formal syntax of commands is:

```
datatype com = SKIP  
| Assign vname aexp  
| Seq com com  
| If bexp com com  
| While bexp com
```

The definition above introduces a datatype for abstract syntax. In the definitions, proofs, and examples further along in these notes, we will often want to refer to concrete program fragments. To make such fragments more readable, we also introduce concrete infix syntax in Isabelle for the four compound constructors of the *com* datatype. The term *Assign x e* for instance can be written as $x ::= e$, the term *Seq c₁ c₂* as $c_1;; c_2$, the term *If b c₁ c₂* as *IF b THEN c₁ ELSE c₂*, and the while loop *While b c* as *WHILE b DO c*. Sequential composition is denoted by “;;” to distinguish it from the “;” that separates assumptions in the $[[\dots]]$ notation. Nevertheless we still pronounce “;;” as “semicolon”.

Example 1. *The following is an example IMP program with two assignments.*

$$"x" ::= Plus (V "y") (N 1); "y" ::= N 2$$

We have not defined its meaning yet, but the intention is that it assigns the value of variable y incremented by one to the variable x , and afterwards sets y to 2. In a more conventional concrete programming language syntax, we would have written

$$x := y + 1; y := 2$$

We will occasionally use this more compact style for examples in the text, with the obvious translation into the formal form.

! Note that, formally we write concrete variable names as strings enclosed in double quotes. Examples are $V "x"$ or $"x" ::= exp$. If we write $V x$ instead, x is a logical variable for the name of the program variable. That is, in $x ::= exp$, the x stands for any concrete name $"x"$, $"y"$, and so on, the same as exp stands for any arithmetic expression.

! The associativity of semicolon in our language is to the left. That means, we have $c_1 ;; c_2 ;; c_3 = (c_1 ;; c_2) ;; c_3$. We will later prove that semantically it does not matter whether semicolon associates to the left or to the right.

The compound commands *IF* and *WHILE* bind stronger than semicolon. That means $WHILE b DO c_1 ;; c_2 = (WHILE b DO c_1) ;; c_2$.

While more convenient than writing abstract syntax trees, as we have seen in the example, even the more concrete Isabelle notation above is occasionally somewhat cumbersome to use. This is not a fundamental restriction of the theorem prover or of mechanising semantics. If one was interested in a more traditional concrete syntax for IMP, or if one were to formalise a larger, more realistic language, one could write separate parsing/printing ML code that integrates with Isabelle and implements the concrete syntax of the language. This is usually only worth the effort when the emphasis is on program verification as opposed to meta theorems about the programming language.

A larger language may also contain a so-called syntactic de-sugaring phase, where more complex constructs in the language are transformed into simple core concepts. For instance, our IMP language does not have syntax for Java style **for**-loops, or **repeat ... until** loops. For our purpose of analysing programming language semantics in general these concepts add nothing new, but for a full language formalisation they would be required. De-sugaring would take the **for**-loop and **do ... while** syntax and translate it into the standard *WHILE* loops that IMP supports. This means, definitions and theorems about the core language only need to worry about one type of loops, while still supporting the full richness of a larger language. This significantly reduces proof size and effort for the theorems that we discuss in these notes.

2.4. Big-Step Semantics

In the previous section we defined the abstract syntax of the IMP language. In this section, we show its semantics. More precisely, we will use a big-step operational semantics to give meaning to commands.

$$\begin{array}{c}
\frac{}{(SKIP, s) \Rightarrow s} \textit{Skip} \qquad \frac{}{(x ::= a, s) \Rightarrow s(x ::= \textit{aval } a \ s)} \textit{Assign} \\
\frac{(c_1, s_1) \Rightarrow s_2 \quad (c_2, s_2) \Rightarrow s_3}{(c_1;; c_2, s_1) \Rightarrow s_3} \textit{Seq} \\
\frac{\textit{bval } b \ s \quad (c_1, s) \Rightarrow t}{(IF \ b \ THEN \ c_1 \ ELSE \ c_2, s) \Rightarrow t} \textit{IfTrue} \\
\frac{\neg \textit{bval } b \ s \quad (c_1, s) \Rightarrow t}{(IF \ b \ THEN \ c_2 \ ELSE \ c_1, s) \Rightarrow t} \textit{IfFalse} \\
\frac{\neg \textit{bval } b \ s}{(WHILE \ b \ DO \ c, s) \Rightarrow s} \textit{WhileFalse} \\
\frac{\textit{bval } b \ s_1 \quad (c, s_1) \Rightarrow s_2 \quad (WHILE \ b \ DO \ c, s_2) \Rightarrow s_3}{(WHILE \ b \ DO \ c, s_1) \Rightarrow s_3} \textit{WhileTrue}
\end{array}$$

Figure 1. The big-step rules of IMP.

In an operational semantics setting, the aim is to capture the meaning of a program as a relation that describes *how* a program executes. Other styles of semantics may be concerned with assigning values or mathematical structures as meanings to programs, e.g. in the so-called denotational style, or they may be interested in capturing the meaning of programs by describing how to reason about them, e.g. in the axiomatic style of Hoare-logic.

2.4.1. Definition

In big-step operational semantics, the relation to be defined is between program, initial state, and final state. Intermediate states during the execution of the program are not visible in the relation. Although the inductive rules that define the semantics will tell us how the execution proceeds internally, the relation itself looks as if the whole program was executed in one big step.

We formalise the big-step execution relation in the theorem prover as a ternary predicate *big-step*. The intended meaning of *big-step* $c \ s \ t$ is that execution of command c starting in state s terminates in state t . To display such predicates in a more intuitive form, we use Isabelle’s syntax mechanism and the more conventional notation $(c, s) \Rightarrow t$ instead of *big-step* $c \ s \ t$.

It remains for us to define which c , s and s' this predicate is made up of. Given the recursive nature of the abstract syntax, it will not come as a surprise that our choice is an inductive definition. Figure 1 shows its rules. Predicates such as $(c, s) \Rightarrow t$ that are defined by a set of rules are often also called **judgements**, because the rules decide for which parameters the predicate is true. However, there is nothing special about them, they are merely ordinary inductively defined predicates.

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

- If the command is *SKIP*, the initial and final state must be the same.

$$\frac{\frac{(''x'' ::= N\ 5, s) \Rightarrow s(''x'' := 5)}{\quad} \quad \frac{(''y'' ::= V\ ''x'', s(''x'' := 5)) \Rightarrow s'}{\quad}}{(''x'' ::= N\ 5;; ''y'' ::= V\ ''x'', s) \Rightarrow s'}$$

where $s' = s(''x'' := 5, ''y'' := 5)$

Figure 2. Derivation tree for execution an IMP program.

- If the command is an 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 *Seq* says the combined command $c_1;; c_2$ started in s_1 executes to s_3 if the the first command executes in s_1 to some intermediate state s_2 and c_2 takes this s_2 to s_3 .
- 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 c_1 results in if started in s . The *IfFalse* rule does the same for the command c_2 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 evaluates to *True* in state s_1 , however, and the body c of the loop takes this state s_1 to some intermediate state s_2 , *and* if the same *WHILE* loop started in s_2 ends in s_3 , then the entire loop also terminates in s_3 .

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, one starts with the conclusion, for instance $(c_1;; c_2, s) \Rightarrow s'$, and then constructs 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 $c_1;; c_2$ example, the parameters are c_1 and c_2 . If the assumptions collapse to an equation about s' as in the *SKIP* and $x ::= a$ case, s' can be replaced directly.

2.4.2. Deriving IMP Executions

As [Figure 2](#) demonstrates, we can use the rules of [Figure 1](#) to construct a so-called **derivation tree** that shows a particular execution is admitted by the IMP language. [Figure 2](#) shows an example: we are executing the program $''x'' ::= N\ 5;; ''y'' ::= V\ ''x''$, starting it in an arbitrary state s . Our claim is that at the end of this execution, we get the same state s , but with both x and y set to 5. We construct the derivation tree from its root, the bottom of [Figure 2](#), by starting with the *Seq* rule, which gives us two obligations, one for each assignment. Working on $''x'' ::= N\ 5$ first, we can conclude via the *Assign* rule from [Figure 1](#) that it results in the state $s(''x'' := 5)$. We feed this intermediate state into the execution of the second assignment, and again with the assignment rule complete the derivation tree. In general, a derivation tree consists of rule

applications at each node and of applications of axioms (rules without premises) at the leafs.

We can conduct the same kind of argument in the theorem prover. The following is the example from Figure 2 in Isabelle. Instead of telling the prover what the result state is, we state the lemma with a schematic variable and let Isabelle compute its value as the proof progresses.

```

schematic_lemma ex: ("x'' ::= N 5;; "y'' ::= V "x'', s) ⇒ ?t
apply(rule Seq)
apply(rule Assign)
apply simp
apply(rule Assign)
done

```

After the proof is finished, Isabelle instantiates the lemma statement, and after simplification we get the expected

$$("x'' ::= N 5;; "y'' ::= V "x'', s) \Rightarrow s("x'' := 5, "y'' := 5)$$

We could use this style of lemma to execute IMP programs symbolically. However, a more convenient 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 \Rightarrow and thus make the predicate available in the **values** command which is similar to **value**, but works on inductive definitions and computes a set of possible results.

```

code_pred big-step .

```

We could now write

```

values {t. (SKIP, λ· 0) ⇒ t}

```

but this only shows us $\{\cdot\}$, i.e. that the result is a set containing one element. Functions cannot always easily be printed, but lists can be, so we just ask for the values of a list of variables we are interested in, using set-comprehension notation:

```

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

```

This has the result $\{\{2,0\}\}$. In the following sections, we will again omit such code generator detail, but we use it to produce examples.

This section showed us how to construct program derivations and how to execute small IMP programs according to the big-step semantics. In the next section, we instead deconstruct executions that we know have happened and analyse all possible ways we could have gotten there.

2.4.3. Rule Inversion

What can we conclude from $(SKIP, s) \Rightarrow t$? Clearly $t = s$. This is an example of rule inversion and is a consequence of the fact that an inductively defined predicate is only true if the rules force it to be true, i.e. only if there is some derivation tree for it.

Inversion of the rules for big-step semantics tells us what we can infer from $(c, s) \Rightarrow t$. For the different commands we obtain the following inverted rules:

$$\begin{aligned}
& (SKIP, s) \Rightarrow t \implies t = s \\
& (x ::= a, s) \Rightarrow t \implies t = s(x := \text{aval } a \ s) \\
& (c_1;; c_2, s_1) \Rightarrow s_3 \implies \exists s_2. (c_1, s_1) \Rightarrow s_2 \wedge (c_2, s_2) \Rightarrow s_3 \\
& (IF \ b \ THEN \ c_1 \ ELSE \ c_2, s) \Rightarrow t \implies \\
& \quad \text{bval } b \ s \wedge (c_1, s) \Rightarrow t \vee \neg \text{bval } b \ s \wedge (c_2, s) \Rightarrow t \\
& (WHILE \ b \ DO \ c, s) \Rightarrow t \implies \\
& \quad \neg \text{bval } b \ s \wedge t = s \vee \\
& \quad \text{bval } b \ s \wedge (\exists s'. (c, s) \Rightarrow s' \wedge (WHILE \ b \ DO \ c, s') \Rightarrow t)
\end{aligned}$$

As an example, we paraphrase the final implication: if $(WHILE \ b \ DO \ c, s) \Rightarrow t$ then either b is false and $t = s$, i.e. rule *WhileFalse* was used, or b is true and there is some intermediate state s' such that $(c, s) \Rightarrow s'$ and $(WHILE \ b \ DO \ c, s') \Rightarrow t$, i.e. rule *WhileTrue* was used.

These inverted rules can be proved automatically by Isabelle from the original rules. Moreover, proof methods like *auto* and *blast* can be instructed to use both the introduction and the inversion rules automatically during proof search. For details see theory *Big-Step*.

One can go one step further and combine the above inverted rules with the original rules to obtain equivalences rather than implications, for example

$$(c_1;; c_2, s_1) \Rightarrow s_3 \iff (\exists s_2. (c_1, s_1) \Rightarrow s_2 \wedge (c_2, s_2) \Rightarrow s_3)$$

Every \implies in the inverted rules can be turned into \iff because the \impliedby direction follows from the original rules.

As an example of the two proof techniques in this and the previous section consider the following lemma. It states that the syntactic associativity of semicolon has no semantic effect. We get the same result, no matter if we group semicolons to the left or to the right.

Lemma 2. $(c_1;; c_2;; c_3, s) \Rightarrow s' \iff (c_1;; (c_2;; c_3), s) \Rightarrow s'$

Proof. We show each direction separately. Consider first the execution where the semicolons are grouped to the left: $((c_1;; c_2); c_3, s) \Rightarrow s'$. By rule inversion we can decompose this execution in twice and obtain the intermediate states s_1 and s_2 such that $(c_1, s) \Rightarrow s_1$, as well as $(c_2, s_1) \Rightarrow s_2$ and $(c_3, s_2) \Rightarrow s'$. From this, we can construct a derivation for $(c_1;; (c_2;; c_3), s) \Rightarrow s'$ by first concluding $(c_2;; c_3, s_1) \Rightarrow s'$ with the *Seq* rule and then using the *Seq* rule again, this time on c_1 , to arrive at the final result. The other direction is analogous. \square

2.4.4. Equivalence of Commands

In the previous section we have applied rule inversion and introduction rules of the big-step semantics to show equivalence between two particular IMP commands. In this section, we define semantic equivalence a concept in its own right.

We call two commands c and c' **equivalent** w.r.t. the big-step semantics when c started in s terminates in s' iff c' started in s also terminates in the same s' . Formally, we define it as an abbreviation:

abbreviation

equiv-c :: $com \Rightarrow com \Rightarrow bool$ (infix \sim 50) where
 $c \sim c' \equiv (\forall s t. (c, s) \Rightarrow t = (c', s) \Rightarrow t)$



Note that the \sim symbol in this definition is not the standard tilde \sim , but the symbol `\<sim>` instead.

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. One example is the unfolding of while loops:

Lemma 3.

$WHILE\ b\ DO\ c \sim IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP$

Another example is a trivial contraction of *IF*:

Lemma 4. $IF\ b\ THEN\ c\ ELSE\ c \sim c$

Of course not all equivalence properties are trivial. For example, the congruence property

Lemma 5. $c \sim c' \implies WHILE\ b\ DO\ c \sim WHILE\ b\ DO\ c'$

is a corollary of

Lemma 6.

$\llbracket (WHILE\ b\ DO\ c, s) \Rightarrow t; c \sim c' \rrbracket \implies (WHILE\ b\ DO\ c', s) \Rightarrow t$

This lemma needs the third main proof technique for inductive definitions: rule induction. Recall that for the big-step semantics, rule induction applies to properties of the form $(c, s) \Rightarrow s' \implies P\ c\ s\ s'$. To prove statements of this kind, we are allowed to consider one case for each introduction rule, and to assume *P* as an induction hypothesis for each occurrence of the inductive relation \Rightarrow in the assumption of the respective introduction rule.

This concept of semantic equivalence is not only useful in phrasing correctness statements, it also has nice algebraic properties. For instance, it forms a so-called **equivalence relation**.

Definition 1 (Equivalence Relation). *A relation R is called an equivalence relation iff it is*

reflexive: $\forall x. R\ x\ x$,
symmetric: $\forall x\ y. R\ x\ y \longrightarrow R\ y\ x$, and
transitive: $\forall x\ y. R\ x\ y \longrightarrow R\ y\ z \longrightarrow R\ x\ z$.

Equivalence relations can be used to partition a set into sets of equivalent elements — in this case, commands that are semantically equivalent belong to the same partition. The standard equality $=$ can be seen as the most fine-grained equivalence relation for a given set.

Lemma 7. *The semantic equivalence \sim is an equivalence relation. It is reflexive: $c \sim c$, symmetric: $c \sim c' \implies c' \sim c$, and transitive: $\llbracket c \sim c'; c' \sim c'' \rrbracket \implies c \sim c''$.*

Proof. All three properties are proved automatically. □

Our relation \sim is also a so-called **congruence** on the syntax of commands: it respects the structure of commands — if all sub-commands are equivalent, so will be the compound command. This is why we called [Lemma 5](#) a congruence property. It establishes that \sim is a congruence relation w.r.t. *WHILE*. We can easily prove further such rules for semicolon and *IF*.

We have used the concept of semantic equivalence in this section as a first example of how semantics can be useful — to prove that two programs always have the same behaviour. We will use this concept in later sections to show the correctness of program transformations and optimisations.

2.4.5. Execution in IMP is deterministic

So far, we have proved properties about particular IMP commands and we have introduced the concept of semantic equivalence.

We have not yet investigated properties of the language itself. One such property is whether the language IMP is **deterministic** or not. A language is called deterministic if, for every input, there is precisely one possible result state. Conversely, a language is called **non-deterministic** if it admits multiple possible results.

Having defined the semantics of the language as a relation, it is not immediately obvious if execution in this language is deterministic or not.

Formally, the language is deterministic if we compare any two executions for the same command and will always arrive in the same final state if we start in the same initial state. The following lemma states this in Isabelle.

Lemma 8 (IMP is deterministic).

$$\llbracket (c, s) \Rightarrow t; (c, s) \Rightarrow t' \rrbracket \implies t' = t$$

Proof. The proof is by induction on the big step semantics. With our inversion and introduction rules from above, each case is then solved automatically by Isabelle. Note that the automation in this proof is not completely obvious. Merely using the proof method **auto** after the induction for instance leads to non-termination, but the backtracking capabilities of **blast** manage to solve each subgoal. Experimenting with different automated methods is encouraged if the standard ones fail. □

So far, we have defined the big-step semantics of IMP, we have explored the proof principles of derivation trees, rule inversion, and rule induction in the context of the big-step semantics, and we have explored semantic equivalence as well as determinism of the language. In the next section we will look at a different way of defining the semantics of IMP.

2.5. Small-Step Semantics

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. For that, we use small-step semantics.

Small-step semantics lets us explicitly observe partial executions and make formal statements about them, for instance if we would like to talk about the interleaved, concurrent execution of multiple programs. The main idea for representing a partial execution is to introduce the concept of how far execution has progressed in the program. There are many ways of doing this. Traditionally, for a high-level language like IMP, we modify the type of the big-step judgement from $com \times state \Rightarrow state \Rightarrow bool$ to something like $com \times state \Rightarrow com \times state \Rightarrow bool$. The second $com \times state$ component of the judgement is the result state of one small, atomic execution step together with a modified command that represents what still has to be executed. We call a $com \times state$ pair a **configuration** of the program, and use the command *SKIP* to indicate that execution has terminated.

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

$$\begin{array}{c}
 \frac{}{(x ::= a, s) \rightarrow (SKIP, s(x := aval\ a\ s))} \textit{Assign} \\
 \\
 \frac{}{(SKIP;; c_2, s) \rightarrow (c_2, s)} \textit{Seq1} \qquad \frac{(c_1, s) \rightarrow (c_1', s')}{(c_1;; c_2, s) \rightarrow (c_1'; c_2, s')} \textit{Seq2} \\
 \\
 \frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \rightarrow (c_1, s)} \textit{IfTrue} \\
 \\
 \frac{\neg\ bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \rightarrow (c_2, s)} \textit{IfFalse} \\
 \\
 \frac{}{(WHILE\ b\ DO\ c, s) \rightarrow (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP, s)} \textit{While}
 \end{array}$$

Figure 3. The small-step rules of IMP.

Going through the rules in [Figure 3](#) we see that:

- Variable assignment is an atomic step. As mentioned, we represent the terminated program by *SKIP*.
- There are two rule for semicolon: either the first part is fully executed already (signified by *SKIP*), then we just continue with the second part, or the first part can be executed further, in which case we perform the execution step and replace this first part with its result.

- An *IF* reduces either to the command in the *THEN* branch or the *ELSE* branch, depending on the value of the condition.
- The final rule is the *WHILE* loop: we define its semantics by merely unrolling the loop once. The subsequent execution steps will take care of testing the condition and possibly executing the body.

Note that we could have used the unrolling definition of *WHILE* in the big-step semantics as well. We were, after all, able to prove it as an equivalence in [Section 2.4.4](#). However, such an unfolding is less natural in the big-step case, whereas in the small-step semantics, the whole idea is to transform the command bit by bit to model execution.

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.

Transforming the program in the small-step style works elegantly, because the language follows the structured programming principle, hence also its alternative name **structural operational semantics**.

We can now define the execution of a program as the reflexive transitive closure of the *small-step* judgement \rightarrow^* :

abbreviation $op \rightarrow^* :: com \times state \Rightarrow com \times state \Rightarrow bool$ **where**
 $x \rightarrow^* y \equiv star\ small\text{-}step\ x\ y$

Example 9. *To look at an example execution of a command in the small-step semantics, we again use the **values** command. This time, we will get multiple elements in the set that it returns — all partial executions of the program. Given the command c with*

$$c = \text{"x''} ::= V \text{"z''}; \text{"y''} ::= V \text{"x''}$$

and an initial state s with

$$s = \langle \text{"x''} := 3, \text{"y''} := 7, \text{"z''} := 5 \rangle$$

we issue the following query to Isabelle

$$\mathbf{values} \{ (c', map\ t\ [\text{"x''}, \text{"y''}, \text{"z''}]) \mid c' \ t. (c, s) \rightarrow^* (c', t) \}$$

The result contains four execution steps, starting with the original program in the initial state, proceeding through partial execution of the the two assignments, and ending in the final state of the final program *SKIP*:

$$\begin{aligned} & \{ (\text{"x''} ::= V \text{"z''}; \text{"y''} ::= V \text{"x''}, [3, 7, 5]), \\ & \quad (SKIP; \text{"y''} ::= V \text{"x''}, [5, 7, 5]), \\ & \quad (\text{"y''} ::= V \text{"x''}, [5, 7, 5]), \\ & \quad (SKIP, [5, 5, 5]) \} \end{aligned}$$

As a further test whether our definition of the small-step semantics is useful, we prove that the rules still give us a deterministic language as in the big-step case.

Lemma 10. $\llbracket cs \rightarrow cs'; cs \rightarrow cs'' \rrbracket \implies cs'' = cs'$

Proof. After induction on the first premise (the small-step semantics), the proof is as automatic as the big-step case. \square

! Recall that both sides of the small-step arrow \rightarrow are configurations, that is, pairs of commands and states. If we don't need to refer to the individual components, we refer to the configuration as a whole, such as cs in the lemma above.

We could conduct further tests like this, but since we already have a semantics for IMP, we can use it to show that our new semantics defines precisely the same behaviour. The next section does this.

2.5.1. Equivalence with big-step semantics

Having defined an alternative semantics for the same language, the first interesting question is of course if our definitions are equivalent. This section shows a formal proof that this is the case.

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

The first direction is $cs \Rightarrow t \implies cs \rightarrow^* (SKIP, t)$. We will show it by rule induction on the big-step judgement, and we will use the following two lemmas. Both lemmas are about the small-step semantics. The first lifts the execution of a command into the context of a semicolon:

Lemma 11.

$$(c_1, s) \rightarrow^* (c_1', s') \implies (c_1;; c_2, s) \rightarrow^* (c_1';; c_2, s')$$

Proof. The proof is by induction on the reflexive transitive closure $star$. The base case is trivial by reflexivity on both sides. In the step case, we use the rule *Seq2* of the small-step semantics and the induction hypothesis with the step case of $star$ on the right-hand side. \square

The second lemma establishes that executing two commands independently one after the other means that we can also execute them as one compound semicolon command with the same result.

Lemma 12. $\llbracket (c_1, s_1) \rightarrow^* (SKIP, s_2); (c_2, s_2) \rightarrow^* (SKIP, s_3) \rrbracket \implies (c_1;; c_2, s_1) \rightarrow^* (SKIP, s_3)$

Proof. This proof is by case distinction on the first premise. In the reflexive case, c_1 is *SKIP* and the statement becomes trivial. In the step case, we use [Lemma 11](#) together with transitivity of $star$. \square

We are now ready to prove that big-step executions imply small-step executions.

Lemma 13. $cs \Rightarrow t \implies cs \rightarrow^* (SKIP, t)$

Proof. The proof is by induction on the big-step semantics. Each case is solved automatically, instantiating the induction hypotheses for each part of the big-step semantics and constructing the corresponding small-step execution. The semicolon case boils down to [Lemma 12](#). The theory file *Small-Step* also contains a long version of this proof that goes into more detail. \square

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 is $(c, s) \rightarrow^* (SKIP, t) \implies (c, s) \Rightarrow t$. Our first attempt would be rule induction on the derivation of the reflexive transitive closure. However, it quickly becomes clear that the statement is too specialised. If we only consider steps that terminate in *SKIP*, we cannot chain them together in the induction. The trick, as always, is to suitably generalise the 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 of an intermediate c' . The key observation is that the big-step semantics always executes the program fully and that (c', s') is just an intermediate configuration 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:

$$\llbracket (c, s) \rightarrow^* (c', s'); (c', s') \Rightarrow t \rrbracket \implies (c, s) \Rightarrow t$$

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 reflexive transitive closure. We extract the step case into its own lemma:

Lemma 14 (Step case). $\llbracket cs \rightarrow cs'; cs' \Rightarrow t \rrbracket \implies cs \Rightarrow t$

Proof. The proof is automatic after rule induction on the small-step semantics. \square

With this, we can now state the main generalised inductive lemma:

Lemma 15 (Small-step implies big-step).

$$\llbracket cs \rightarrow^* cs'; cs' \Rightarrow t \rrbracket \implies cs \Rightarrow t$$

Proof. As mentioned, the proof is by induction on the reflexive transitive closure, and the step case is solved by [Lemma 14](#). \square

Our initial second direction of the proof is now an easy corollary.

Corollary 16. $cs \rightarrow^* (SKIP, t) \implies cs \Rightarrow t$

Proof. As planned, we use [Lemma 15](#) and instantiate cs' to $(SKIP, t)$, which collapses the second premise of [Lemma 15](#) to *True*. \square

Both directions together let us conclude the equivalence we were aiming for in the first place.

Corollary 17. $(c, s) \Rightarrow t \iff (c, s) \rightarrow^* (SKIP, t)$

This concludes our proof that the small-step and big-step semantics of IMP are equivalent. Such equivalence proofs are useful whenever there are different formal descriptions of the same artefact. The reason one might want different expressions of the same thing is that they differ in what and how they can be used for. For instance, big-step semantics are relatively intuitive to define, while small-step semantics allow us to make more fine-grained formal observations. The next section shows one such kind of observation.

3. Types

This section introduces types into IMP, first a traditional programming language type system, then more sophisticated type systems for information flow analysis.

Why bother with types? Because they prevent mistakes. They are a simple, automatic way to find obvious problems in programs before these programs are ever run.

There are 3 kinds of types.

The Good Static types that *guarantee* absence of certain runtime faults.

The Bad Static types that have mostly decorative value but do not guarantee anything at runtime.

The Ugly Dynamic types that detect errors only when it can be too late.

Examples of the first kind are Java, ML and Haskell. In Java for instance, the type system enforces that there will be no memory access errors, which in other languages manifest as segmentation faults. ML and Haskell have even more powerful type systems that can be used to enforce basic higher-level program properties by type alone, for instance strict information hiding in modules or abstract data types.

Famous examples of the bad kind are C and C++. These languages have static type systems, but they can be circumvented easily. The language specification may not even allow these circumventions, but there is no way for compilers to guarantee their absence.

Examples for dynamic types are scripting languages such as Perl and Python. These languages are typed, but typing violations are discovered and reported at runtime only, which leads to runtime messages such as “*TypeError: ...*” in Python for instance.

In all of the above cases, types are useful. Even in Perl and Python, they at least are known at runtime and can be used to conveniently convert values of one type into another and to enable object-oriented features such as dynamic dispatch of method calls. They just don't provide any compile-time checking. In C and C++, the compiler can at least report some errors already at compile time and alert the programmer to obvious problems. But only static, sound type systems can enforce the absence of whole classes of runtime errors.

In fact, static type systems can be seen as proof systems, type checking as proof checking, and type inference as proof search. Every time a type checker passes a program, it in effect proves a set of small theorems about this programs.

The ideal for a static type system is to be permissive enough not to get into the programmer's way while being strong enough to achieve Robin Milner's slogan *Well-typed programs cannot go wrong* [9]. It is the most influential slogan and one of the most influential papers in programming language theory.

What could go wrong? Some examples of common runtime errors are corruption of data, null pointer exceptions, nontermination, running out of memory, and leaking secrets. There exist type systems for all of these, and more, but in practise only the first is covered in widely-used languages such as Java, C#, Haskell, or ML. We will cover this first kind in [Section 3.1](#), and information leakage in [Section 3.2](#).

As mentioned above, the ideal for a language is to be **type safe**. Type safe means that the execution of a well-typed program cannot lead to certain errors. Java and the JVM, for instance, have been *proved* to be type safe. An execution of a Java program may throw legitimate language exceptions such as `NullPointerException` or `OutOfMemory`, but it can never produce data corruption or segmentation faults other than by hardware defects or calls into native code. In the following sections we will show how to prove such theorems for IMP.

Type safety is a feature of a programming language. **Type soundness** means the same thing, but talks about the type system instead. It means that a type system is *sound* or *correct* with respect to the semantics of the language: If the type system says yes, the semantics does not lead to an error. The semantics is the primary definition of behaviour, and therefore the type system must be justified w.r.t. it.

If there is soundness, how about completeness? Remember Rice's theorem:

Nontrivial semantic properties of programs (e.g. termination)
are undecidable.

Hence there is no (decidable) type system that accepts precisely the programs that have a certain semantic property.

Automatic analysis of semantic program properties is necessarily incomplete.

This applies not only to type systems, but also to the other automatic semantic analyses that we present here.

3.1. Typed IMP

In this section we develop a very basic static type system as a typical application of programming 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 real numbers as well as integers.

To define this new language, we go through the complete exercise again, and define new arithmetic and boolean expressions, together with their values and semantics, as well as a new semantics for commands. In the theorem prover we can do this by merely copying the original definitions and tweaking them slightly. Here, we will briefly walk through the new definitions step by step.

We begin with values occurring in the language. Our introduction of a second kind of value means our value type now correspondingly has two alternatives:

datatype $val = Iv\ int \mid Rv\ real$

This definition means we tag values with their type at runtime (the constructor tells us which is which). We do this, so we can observe when things go wrong, for instance when a program is trying to add an integer to a real. This does not mean that a compiler for this language would also need to carry this information around at runtime. In fact, it is the type system that lets us avoid this overhead! Since it will only admit safe programs, the compiler can optimise and blindly apply the operation for the correct type. It can determine statically what that correct type is.

 Note that the type *real* stands for the mathematical real numbers, not floating point numbers, just as we use mathematical integers in IMP instead of finite machine words. For the purposes of the type system this difference does not matter. For formalising a real programming language, one should model values more precisely.

Continuing in the formalisation of our new type language, variable names and state stay as they are, i.e. variable names are strings and the state is a function from such names to values.

Arithmetic expressions, however, now have two kinds of constants: *int* and *real*:

datatype $aexp = Ic\ int \mid Rc\ real \mid V\ vname \mid Plus\ aexp\ aexp$

In contrast to vanilla IMP, we can now write arithmetic expressions that make no sense, or in other words 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 added, this expression makes no sense. We would like to express in our semantics that this is not an expression with well-defined behaviour. One alternative would be to continue using a functional style of semantics for expressions. In this style we would now return *val option* with the constructor *None* of the option data type to denote the undefined cases. It is quite possible to do so, but we would have to explicitly enumerate all undefined cases.

It is more elegant and concise to only write down the cases that make sense and leave everything else undefined. The operational semantics judgement already lets us do this for commands. We can use the same style for arithmetic expressions. Since we are not interested in intermediate states at this point, we choose the big-step style.

Our new judgement relates an expression and the state it is evaluated in to the value it is evaluated to. We refrain from introducing additional syntax and

$$\begin{array}{c}
\frac{}{taval (Ic\ i)\ s\ (Iv\ i)} \quad \frac{}{taval (Rc\ r)\ s\ (Rv\ r)} \quad \frac{}{taval (V\ x)\ s\ (s\ x)} \\
\\
\frac{taval\ a_1\ s\ (Iv\ i_1) \quad taval\ a_2\ s\ (Iv\ i_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Iv\ (i_1 + i_2))} \\
\\
\frac{taval\ a_1\ s\ (Rv\ r_1) \quad taval\ a_2\ s\ (Rv\ r_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Rv\ (r_1 + r_2))}
\end{array}$$

Figure 4. Inductive definition of $taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$

$$\begin{array}{c}
\frac{}{tbval\ (Bc\ v)\ s\ v} \quad \frac{tbval\ b\ s\ bv}{tbval\ (Not\ b)\ s\ (\neg\ bv)} \\
\\
\frac{tbval\ b_1\ s\ bv_1 \quad tbval\ b_2\ s\ bv_2}{tbval\ (And\ b_1\ b_2)\ s\ (bv_1 \wedge bv_2)} \quad \frac{taval\ a_1\ s\ (Iv\ i_1) \quad taval\ a_2\ s\ (Iv\ i_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (i_1 < i_2)} \\
\\
\frac{taval\ a_1\ s\ (Rv\ r_1) \quad taval\ a_2\ s\ (Rv\ r_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (r_1 < r_2)}
\end{array}$$

Figure 5. Inductive definition of $tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$

call this judgement $taval$ for *typed arithmetic value* of an expression. In Isabelle, this translates to an inductive definition with type $aexp \Rightarrow state \Rightarrow val \Rightarrow bool$. We show its introduction rules in Figure 4. The term $taval\ a\ s\ v$ means that arithmetic expression a evaluates in state s to value v .

The definition is straightforward. The first rule $taval\ (Ic\ i)\ s\ (Iv\ i)$ for instance says that an integer constant $Ic\ i$ always evaluates to the the value $Iv\ i$, no matter what the state is. The interesting cases are the rules 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 following is an example derivation for $taval$ where $s\ ''x'' = Iv\ 4$.

$$\frac{\frac{}{taval\ (Ic\ 3)\ s\ (Iv\ 3)} \quad \frac{}{taval\ (V\ ''x'')\ s\ (Iv\ 4)}}{taval\ (Plus\ (Ic\ 3)\ (V\ ''x''))\ s\ (Iv\ 4)}$$

For $s\ ''x'' = Rv\ 3$ on the other hand, there would be no execution of $taval$ that we could derive for the same term.

The syntax for boolean expressions remains unchanged. Their evaluation, however, is different. In order to use the operational semantics for arithmetic expressions that we just defined, we need to employ the same operational style for boolean expressions. Figure 5 shows the formal definition. Next to its own error conditions, e.g. for $Less\ (Ic\ n)\ (Rc\ r)$, this definition also propagates errors from the evaluation of arithmetic expressions: If there is no evaluation for a then there is also no evaluation for $Less\ a\ b$.

The syntax for commands is again unchanged. We now have a choice: do we define a big-step or a small-step semantics? The answer seems clear: it must be

$$\begin{array}{c}
\frac{}{taval\ a\ s\ v} \\
\hline
(x ::= a, s) \rightarrow (SKIP, s(x := v)) \\
\\
\frac{}{(SKIP;; c, s) \rightarrow (c, s)} \quad \frac{(c_1, s) \rightarrow (c_1', s')}{(c_1;; c_2, s) \rightarrow (c_1';; c_2, s')} \\
\\
\frac{}{tbval\ b\ s\ True} \\
\hline
(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \rightarrow (c_1, s) \\
\\
\frac{}{tbval\ b\ s\ False} \\
\hline
(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \rightarrow (c_2, s) \\
\\
\hline
(WHILE\ b\ DO\ c, s) \rightarrow (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP, s)
\end{array}$$

Figure 6. Inductive definition of $op \rightarrow :: com \times state \Rightarrow com \times state \Rightarrow bool$

small-step semantics, because only there can we observe when things are going wrong in the middle of an execution. In the small-step case, error states are explicitly visible in intermediate states: if there is an error, the semantics gets stuck in a non-final program configuration with no further progress possible. We need executions *to be able to go wrong* if we want a meaningful proof that they do not.

In fact, the big-step semantics could be adjusted as well, to perform the same function. By default, in the style we have seen so far, a big-step semantics is not suitable for this, because it conflates non-termination, which is allowed, with runtime errors or undefined execution, which are not. If we mark errors specifically and distinguish them from non-termination in the big-step semantics, we can observe errors just as well as in the small-step case.

So we still have a choice. Small-step semantics are more concise and more traditional for type soundness proofs. Therefore we will choose this one. Later, in [Section 4](#), we will show the other alternative.

After all this discussion, the definition of the small-step semantics for typed commands is almost the same as the untyped case. As shown in [Figure 6](#), it merely refers to the new judgements for arithmetic and boolean expressions, but does not add any new rules on its own.

As before, the execution of a program is a sequence of small steps, denoted by star, for example $(c, s) \rightarrow^* (c', s')$.

Example 18. *For well-behaved programs, our typed executions look as before. For instance, let s satisfy $s \Vdash y'' = Iv\ 7$. Then we get the following example execution chain.*

$$\begin{array}{l}
(x'' ::= V\ y'', y'' ::= Plus\ (V\ x'')\ (V\ y''), s) \rightarrow \\
(y'' ::= Plus\ (V\ x'')\ (V\ y''), s(x'' := Iv\ 7)) \rightarrow \\
(SKIP, s(x'' := Iv\ 7, y'' := Iv\ 14))
\end{array}$$

However, programs that contain type errors can get stuck. For example, if in the same state s , we take a slightly different program that adds a constant of the wrong type, we get:

$$\begin{array}{c}
\overline{\Gamma \vdash Ic\ i : Ity} \quad \overline{\Gamma \vdash Rc\ r : Rty} \quad \overline{\Gamma \vdash V\ x : \Gamma\ x} \\
\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau \\
\hline
\Gamma \vdash Plus\ a_1\ a_2 : \tau
\end{array}$$

Figure 7. Inductive definition of \vdash $-$ $-$ $tyenv \Rightarrow aexp \Rightarrow ty \Rightarrow bool$

$$\begin{array}{l}
(''x'' ::= V\ ''y'';; ''y'' ::= Plus\ (V\ ''x'')\ (Rc\ 3),\ s) \rightarrow \\
(''y'' ::= Plus\ (V\ ''x'')\ (Rc\ 3),\ s(''x'' := Iv\ 7))
\end{array}$$

The first assignment succeed as before, but after that there there is no further execution step possible, because we cannot find an execution for taval on the right-hand side of the second assignment.

3.1.1. The Type System

Having defined our new language above, we can now define its type system. The idea of such type systems is to predict statically which values will appear at runtime and to exclude programs in which unsafe values or value combinations might be encountered.

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*, corresponding to the two kinds of values we have introduced. In Isabelle, this is simply:

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 so-called typing environment. Where a runtime state maps variable names to values, a static typing environment maps variable names to their static types.

type-synonym *tyenv* = *vname* \Rightarrow *ty*

For example, we could have $\Gamma\ ''x'' = Ity$, telling us that variable *x* has type integer and that we should therefore not use it in an expression of type real.

With this, we can give typing rules for arithmetic expressions. The idea is simple: constants have fixed type, variables have the type the typing environment Γ prescribes, and *Plus* can be typed with type τ if both operands have the same type τ . Figure 7 shows the definition in Isabelle. We use the notation $\Gamma \vdash a : ty$ to say that expression *a* has type *ty* in context Γ .

The typing rules for booleans in Figure 8 are even simpler. We do not need a result type, because it will always be bool, so the notation is just $\Gamma \vdash b$ for expression *b* is well-typed in context Γ . For the most part, we just need to capture that boolean expressions are well-typed if their subexpressions are well-typed. The interesting case is the connection to arithmetic expressions in *Less*. Here we demand that both operands have the same type τ , i.e. either we compare two ints or two reals, but not an int to a real.

Similarly, commands are well-typed if their subexpressions are well-typed. The only non-regular case here is assignment: we demand that the arithmetic

$$\begin{array}{c}
\frac{}{\Gamma \vdash Bc\ v} \qquad \frac{\Gamma \vdash b}{\Gamma \vdash Not\ b} \\
\\
\frac{\Gamma \vdash b_1 \quad \Gamma \vdash b_2}{\Gamma \vdash And\ b_1\ b_2} \qquad \frac{\Gamma \vdash a_1 : \tau \quad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Less\ a_1\ a_2}
\end{array}$$

Figure 8. Inductive definition of $op \vdash :: tyenv \Rightarrow bexp \Rightarrow bool$

$$\begin{array}{c}
\frac{}{\Gamma \vdash SKIP} \qquad \frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x ::= a} \\
\\
\frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1;;\ c_2} \qquad \frac{\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2} \qquad \frac{\Gamma \vdash b \quad \Gamma \vdash c}{\Gamma \vdash WHILE\ b\ DO\ c}
\end{array}$$

Figure 9. Inductive definition of $op \vdash :: tyenv \Rightarrow com \Rightarrow bool$

expression has the same type as the variable it is assigned to. We re-use the syntax $\Gamma \vdash c$ for *command c is well-typed in context Γ* .

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.

Note that there is precisely one rule per syntactic construct in our definition of the type system, and the premises of each rule apply the typing judgement only to sub-terms of the conclusion. We call such rule sets **syntax directed**. Syntax directed rules are a good candidate for automatic application and for deriving an algorithm that infers the type simply by applying them backwards, at least if there are no side conditions in their assumptions. Since there is exactly one rule per construct, it is always clear which rule to pick and there is no need for back-tracking. Further, since there is always at most one rule application per syntax node in the term or expression the rules are applied to, this process must terminate. This idea can be extended to allow side conditions in the assumptions of rules, as long as these side conditions are decidable.

Given such a type system, we can now check whether a specific program c is well-typed. To do so, we merely need to construct a derivation tree for the judgment $\Gamma \vdash c$. Such a derivation tree is also called a **type derivation**. Let for instance $\Gamma''x'' = Ity$ as well as $\Gamma''y'' = Ity$. Then our previous example program is well-typed, because of the following type derivation.

$$\frac{\frac{\Gamma''y'' = Ity}{\Gamma \vdash V''y'' : Ity} \quad \frac{\frac{\Gamma''x'' = Ity}{\Gamma \vdash V''x'' : Ity} \quad \frac{\Gamma''y'' = Ity}{\Gamma \vdash V''y'' : Ity}}{\Gamma \vdash Plus(V''x'')(V''y'') : Ity}}{\Gamma \vdash ''y'' ::= Plus(V''x'')(V''y'')} \quad \frac{\Gamma \vdash ''x'' ::= V''y''}{\Gamma \vdash ''x'' ::= V''y'';; ''y'' ::= Plus(V''x'')(V''y'')}}$$

3.1.2. Well-typed Programs do Not Get Stuck

In this section we prove that the type system defined above is sound. As mentioned earlier, Robert Milner coined the phrase *well-typed programs cannot go wrong* [9], i.e. well-typed programs will not exhibit any runtime errors such as segmentation faults or undefined execution. In our small-step semantics we have defined precisely what “go wrong” means formally: a program exhibits a runtime error when the semantics gets stuck.

To prove type soundness we merely have to prove that well-typed programs never get stuck. They either terminate successfully, or they make further progress. Taken literally, the above sentence translates into the following lemma statement:

$$\llbracket (c, s) \rightarrow^* (c', s'); \Gamma \vdash c \rrbracket \implies c' = \text{SKIP} \vee (\exists cs''. (c', s') \rightarrow cs'')$$

Given an arbitrary command c , which is well-typed $\Gamma \vdash c$, any execution $(c, s) \rightarrow^* (c', s')$ either has terminated successfully with $c' = \text{SKIP}$, or can make another execution step $\exists cs''. (c', s') \rightarrow cs''$. Clearly, this statement is wrong, though: take c for instance to be a command that computes the sum of two variables: $z := x+y$. This command is well-typed, for example, if the variables are both of type `int`. However, if we start the command in a state that disagrees with this type, e.g. where x contains an `int`, but y contains a `real`, the execution gets stuck.

Of course, we want the value of a variable to be of type `int` when the typing says it should be `int`. This means we want not only the program to be well-typed, but the state to be well-typed too.

We so far have the state assigning values to variables and we have the type system statically assigning types to variables in the program. The concept of well-typed states connects these two: we define a judgement that determines if a runtime state is compatible with a typing environment for variables. We call this formal judgement *styping* below, written $\Gamma \vdash s$. We equivalently also say that a state s **conforms** to a typing environment Γ .

With this judgement, our full statement of type soundness is now

$$\llbracket (c, s) \rightarrow^* (c', s'); \Gamma \vdash c; \Gamma \vdash s; c' \neq \text{SKIP} \rrbracket \implies \exists cs''. (c', s') \rightarrow cs''$$

Given a well-typed program, started in a well-typed state, any execution that has not reached `SKIP` yet can make another step.

We will prove this property by induction on the reflexive transitive closure of execution steps, which naturally decomposes this type soundness property 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.

In the following, we formalise the soundness proof for typed IMP.

We start the formalisation by defining a function from values to types, which will then allow us to phrase what well-typed states are. 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 subtype 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 $type :: val \Rightarrow ty$ **where**
 $type (Iv\ i) = Ity$
 $type (Rv\ r) = Rty$

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 $op \vdash :: tyenv \Rightarrow state \Rightarrow bool$ **where**
 $\Gamma \vdash s \longleftrightarrow (\forall x. type (s\ x) = \Gamma\ x)$

We now have everything defined to start the soundness proof. The plan is to prove progress and preservation, and to conclude from that the final type soundness statement that an execution of a well-typed command started in a well-typed state will never get stuck. To prove progress and preservation for commands, we will first 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 :

Lemma 19 (Preservation for arithmetic expressions).

$[\Gamma \vdash a : \tau; taval\ a\ s\ v; \Gamma \vdash s] \Longrightarrow type\ v = \tau$

Proof. The proof is by rule induction on the type derivation $\Gamma \vdash a : \tau$. If we declare rule inversion on *taval* to be used automatically and unfold the definition of *styping*, Isabelle proves the rest. \square

The proof of the progress lemma is slightly 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 .

Lemma 20 (Progress for arithmetic expressions).

$[\Gamma \vdash a : \tau; \Gamma \vdash s] \Longrightarrow \exists v. taval\ a\ s\ v$

Proof. The proof is again by rule induction on the typing derivation. The interesting case is *Plus* $a_1\ a_2$. The induction hypothesis gives us two values v_1 and v_2 for the subexpressions a_1 and a_2 . If v_1 is an integer, then, by preservation, the type of a_1 must have been *Ity*. The typing rule says that the type of a_2 must be the same. This means, by preservation, the type of v_2 must be *Ity*, which in turn means then v_2 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 solved automatically. \square

For boolean expressions, there is no preservation lemma, because *tval*, by its Isabelle type, can only return boolean values. The progress statement makes sense, though, and follows the standard progress statement schema.

Lemma 21 (Progress for boolean expressions).

$$\llbracket \Gamma \vdash b; \Gamma \vdash s \rrbracket \implies \exists v. \text{tval } b \text{ } s \text{ } v$$

Proof. 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. \square

For commands, there are two preservation statements, because the configurations in our small-step semantics have two components: program and state. We first show that the program remains well-typed and then that the state does. Both proofs are by induction on the small-step semantics. They could be proved by induction on the typing derivation as well. 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 the 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 they decompose along the same syntactic lines.

Theorem 22 (Preservation: commands stay well-typed).

$$\llbracket (c, s) \rightarrow (c', s'); \Gamma \vdash c \rrbracket \implies \Gamma \vdash c'$$

Proof. 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 subcommand is the while loop. Here we just need to apply the typing rules for *IF* and sequential composition and are done. \square

Theorem 23 (Preservation: states stay well-typed).

$$\llbracket (c, s) \rightarrow (c', s'); \Gamma \vdash c; \Gamma \vdash s \rrbracket \implies \Gamma \vdash s'$$

Proof. The proof is by induction on the small-step semantics. Most cases are simple instantiations of the induction hypothesis, without further modifications to the state. In the assignment operation, we do 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. \square

The next step is the progress lemma for commands. 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.

Theorem 24 (Progress for commands).

$$\llbracket \Gamma \vdash c; \Gamma \vdash s; c \neq \text{SKIP} \rrbracket \implies \exists cs'. (c, s) \rightarrow cs'$$

Proof. 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. As for the other cases: *SKIP* is trivial, sequential composition just applies the induction hypotheses and makes a case distinction if c_1 is *SKIP* or not, and *WHILE* always trivially makes progress in the small-step semantics, because it is unfolded into an *IF/WHILE*. \square

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.

Theorem 25 (Type soundness).

$$\llbracket (c, s) \rightarrow^* (c', s'); \Gamma \vdash c; \Gamma \vdash s; c' \neq \text{SKIP} \rrbracket \implies \exists cs''. (c', s') \rightarrow cs''$$

Proof. The proof lifts the one-step preservation and progress results to a sequence of steps by induction on the reflexive transitive closure. The base case of zero steps is solved by the progress lemma, the step case needs our two preservation lemmas for commands. \square

This concludes the section on typing. We have seen, exemplified by a very simple type system, what a type soundness statement means, how it interacts with the small-step semantics, and how it is proved. While the proof itself will grow in complexity for more interesting languages, the general schema of progress and preservation 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 definition of the small-step semantics carries the main meaning and strength of the type soundness statement.

Our mantra for type systems:

Type systems have a purpose: the static analysis of programs in order to predict their runtime behaviour. The correctness of this prediction must be provable.

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 to public observers. The idea is that we want an easy and automatic way to check if programs protect private data such as passwords, bank details, or medical records.

Ensuring such **information flow control** properties based on a programming language analysis such as a type system is a part of so-called **language-based security**. Another common option for enforcing information flow control 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 also allows more absolute statements. As techniques they are not incompatible: both approaches could be mixed to enforce a particular information flow property.

Note that absolute statements in language-based security 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.

3.2.1. Security Levels and Expressions

We begin developing our security type system 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 information may only flow from variables of ‘lower’ security levels to variables of ‘higher’ levels, but never the other way around.

In the literature, levels are often reduced to just two: high and low. We keep things slightly more general by 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 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. We could then also enforce that information does not travel ‘sideways’ between two incomparable security levels. For the sake of simplicity we refrain from doing so here and merely use *nat*.

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 .

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 s_1 and s_2 : if s_1 and s_2 agree on low variables (but may differ on high variables!), then the states resulting from executing (c, s_1) and (c, s_2) 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 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 define the security level of an expression as the highest level of any variable that occurs in it. We make use of Isabelle’s overloading and call the security level of an arithmetic or boolean expression $sec\ e$.

```

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

fun sec :: bexp ⇒ level where
  sec (Bc v)         = 0
  sec (Not b)        = sec b
  sec (And b1 b2) = max (sec b1) (sec b2)
  sec (Less a1 a2) = max (sec a1) (sec 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 $sec\ e$, the value of e should stay 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 \leq , but also the strict $<$ later on, so we define both here:

$$\begin{array}{c}
\frac{}{l \vdash \text{SKIP}} \quad \frac{\text{sec } a \leq \text{sec } x \quad l \leq \text{sec } x}{l \vdash x ::= a} \quad \frac{l \vdash c_1 \quad l \vdash c_2}{l \vdash c_1;; c_2} \\
\frac{\text{max } (\text{sec } b) \ l \vdash c_1 \quad \text{max } (\text{sec } b) \ l \vdash c_2}{l \vdash \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2} \quad \frac{\text{max } (\text{sec } b) \ l \vdash c}{l \vdash \text{WHILE } b \ \text{DO } c}
\end{array}$$

Figure 10. Definition of $\text{sectype} :: \text{nat} \Rightarrow \text{com} \Rightarrow \text{bool}$

$$\begin{aligned}
s = s' (\leq l) &\equiv \forall x. \text{sec } x \leq l \longrightarrow s \ x = s' \ x \\
s = s' (< l) &\equiv \forall x. \text{sec } x < l \longrightarrow s \ x = s' \ x
\end{aligned}$$

With this, the proof of our first two security properties is simple and automatic: The evaluation of an expression e only depends on variables with level below or equal to $\text{sec } e$.

Lemma 26 (Noninterference for arithmetic expressions).
 $\llbracket s_1 = s_2 (\leq l); \text{sec } e \leq l \rrbracket \Longrightarrow \text{aval } e \ s_1 = \text{aval } e \ s_2$

Lemma 27 (Noninterference for boolean expressions).
 $\llbracket s_1 = s_2 (\leq l); \text{sec } b \leq l \rrbracket \Longrightarrow \text{bval } b \ s_1 = \text{bval } b \ s_2$

3.2.2. Syntax Directed Typing

As usual in IMP, the typing for expressions was simple. We now define a syntax directed set of security typing rules for commands. This makes the rules directly executable and allows us to run examples. Checking for explicit flows, i.e. assignments from high to low variables is easy. For implicit flows, the main idea of the type system is to track the security level of variables that decisions are made on, and to make sure that their level is lower or equal to variables assigned to in that context.

We write $l \vdash c$ to mean that command c contains no information flows to variables lower than level l , and only safe flows into variables $\geq l$.

Going through the rules of Figure 10 in detail, we have defined *SKIP* to be safe at any level. We 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* command could admit implicit flows. We prevent these by demanding that for *IF* to conform to l , both c_1 and c_2 have to conform to level l or the level of the boolean 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 command.

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.

Example 28. *Testing our intuition about what we have just defined, we look at four examples for various security levels.*

$$0 \vdash \text{IF } \text{Less } (V \text{ ''}x_1'') (V \text{ ''}x'') \ \text{THEN } \text{''}x_1'\' ::= N \ 0 \ \text{ELSE } \text{SKIP}$$

The statement claims that the command is well-typed at security level 0: flows can occur down to even a level 0 variable, but they have to be internally consistent, i.e. flows must still only be from lower to higher levels. According to our arbitrary example definition of security levels that assigns the length of the variable to the level, variable $x1$ has level 2, and variable x has level 1. This means the evaluation of this typing expression will yield *True*: the condition has level 2, and the context is 0, so according to the *IF* rule, both commands must be safe up to level 2, which is the case, because the first assignment sets a level 2 variable, and the second is just *SKIP*.

Does the same work if we assign to a lower-level variable?

$0 \vdash \text{IF Less } (V \text{ "x1"}) (V \text{ "x"}) \text{ THEN "x"} ::= N 0 \text{ ELSE SKIP}$

Clearly not. Again, we need to look at the *IF* rule which still says the assignment must be safe at level 2, i.e. we have to derive $2 \vdash \text{"x"} ::= N 0$. But x is of level 1, and the assignment rule demands that we only assign to levels higher than the context. Intuitively, the *IF* decision expression reveals information about a level 2 variable. If we assign to a level 1 variable in one of the branches we leak level 2 information to level 1.

What if we demand a higher security context from our original example?

$2 \vdash \text{IF Less } (V \text{ "x1"}) (V \text{ "x"}) \text{ THEN "x1"} ::= N 0 \text{ ELSE SKIP}$

Context of level 2 still works, because our highest level in this command is level 2, and our arguments from the first example still apply.

What if we go one level higher?

$3 \vdash \text{IF Less } (V \text{ "x1"}) (V \text{ "x"}) \text{ THEN "x1"} ::= N 0 \text{ ELSE SKIP}$

Now we get *False*, because we need to take the maximum of the context and the boolean expression for evaluating the branches. The intuition is that the context gives the minimum level to which we may reveal information.

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 command was already 0 in the beginning, the context would not have mattered, we only would have overwritten 0 with 0. As we know by now, 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 simple type system presented here, going back to Volpano, Irvine, and Smith [18], 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 here, however, 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

$$\llbracket (c, s) \Rightarrow s'; (c, t) \Rightarrow t'; 0 \vdash c; s = t (\leq l) \rrbracket \Longrightarrow s' = t' (\leq l)$$

An important property, which will be useful for this 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 . Anti-monotonicity is also often called the **subsumption rule**, to say that higher contexts subsume lower ones. Intuitively it is clear that this property should hold: we defined $l \vdash c$ to mean that there are no flows to variables $< l$. If we write $l' \vdash c$ with an $l' \leq l$, then we are only admitting more flows, i.e. we are making a weaker statement.

Lemma 29 (Anti-monotonicity). $\llbracket l \vdash c; l' \leq l \rrbracket \Longrightarrow l' \vdash c$

Proof. The formal proof is by rule induction on the type system. Each of the cases is then solved automatically. \square

The second key lemma in the argument for the soundness of our security type system is **confinement**: an execution that is type correct in context l can only change variables of level l and above, or conversely, all variables below l will remain unchanged. In other words, the effect of c is **confined** to variables of level $\geq l$.

Lemma 30 (Confinement). $\llbracket (c, s) \Rightarrow t; l \vdash c \rrbracket \Longrightarrow s = t (< l)$

Proof. 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. Therefore, we choose rule induction on the big-step execution instead. In the *IF* and *WHILE* cases, we make use of anti-monotonicity to instantiate the induction hypothesis. In the *IfTrue* case, for instance, the hypothesis is $l \vdash c_1 \Longrightarrow s = t (< l)$, but from the type system we only know $\max(\text{sec } b) l \vdash c_1$. Since $l \leq \max(\text{sec } b) l$, anti-monotonicity allows us to conclude $l \vdash c_1$. \square

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

Theorem 31 (Noninterference).

$$\llbracket (c, s) \Rightarrow s'; (c, t) \Rightarrow t'; 0 \vdash c; s = t (\leq l) \rrbracket \Longrightarrow s' = t' (\leq l)$$

Proof. The proof is again by induction on the big-step execution. The *SKIP* case is easy and automatic, as it should be.

The assignment case is already somewhat interesting. First, we note that s' is the usual state update $s(x := \text{aval } a \text{ } s)$ 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 : $\text{sec } a \leq \text{sec } x$. Now we show that the two updated states s' and t' still agree on all variables below l . For this, it is sufficient to show that the states agree on the new value if $\text{sec } x < l$, and that all other variables y with $\text{sec } y < l$ still agree as before. In the first case, looking at x , we know from

above that $\text{sec } a \leq \text{sec } x$. Hence, by transitivity, we have that $\text{sec } a \leq l$. This is enough for our noninterference result on expressions to apply, given that we also know $s = t (\leq l)$ from the premises. This means, we get $\text{aval } a \ s = \text{aval } a \ t$: the new values for x agree as required. The case for all other variables y below l follows directly from $s = t (\leq l)$.

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

IF has two symmetric cases as usual. We will look only at the *IfTrue* case in more detail. We begin the case by noting via rule inversion that both branches are type correct to level $\text{sec } b$, since the maximum with 0 is the identity, i.e. we know $\text{sec } b \vdash c_1$. Then we perform a case distinction: either the level of b is $\leq l$ or it is not. If $\text{sec } b \leq l$, i.e. the *IF* decision is on a more public level than l , then s and t , which agree below l , also agree below $\text{sec } b$. That in turn means by our noninterference lemma for expressions that they evaluate to the same result, so $\text{bval } b \ s = \text{True}$ and $\text{bval } b \ t = \text{True}$. We already noted $\text{sec } b \vdash c_1$ by rule inversion, and with anti-monotonicity, we get the necessary $0 \vdash c_1$ to apply the induction hypothesis and conclude the case. In the other case, if $l < \text{sec } b$, i.e. a condition on a more confidential level than l we do not know that both *IF* commands will take the same branch. However, we do know that the whole command is a high-confidentiality computation. We can use the typing rule for *IF* to conclude $\text{sec } b \vdash \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2$ since we know both $\text{max } (\text{sec } b) \ 0 \vdash c_1$ and $\text{max } (\text{sec } b) \ 0 \vdash c_2$. This in turn means we now can apply confinement: everything below $\text{sec } b$ 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 $s = t (\leq l)$, we can conclude $s' = t' (\leq l)$ which closes the whole *IfTrue* case.

The *IfFalse* and *WhileFalse* 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 *WhileTrue* case is similar. Here, we have to work slightly harder to apply the induction hypotheses, once for the body and once for the rest of the loop, but the confinement side of the argument stays the same. \square

3.2.4. The Standard Typing System

The judgement $l \vdash c$ presented above is nicely intuitive and executable. However, the standard formulation in the literature is slightly different, replacing the maximum computation directly with the anti-monotonicity rule. We introduce the standard system now in Figure 11 and show equivalence with our previous formulation.

The equivalence proof goes by rule induction on the respective type system in each direction separately. Isabelle proves each subgoal of the induction automatically.

Lemma 32. $l \vdash c \implies l \vdash' c$

Lemma 33. $l \vdash' c \implies l \vdash c$

$$\begin{array}{c}
\frac{}{l \vdash' \text{SKIP}} \qquad \frac{\text{sec } a \leq \text{sec } x \quad l \leq \text{sec } x}{l \vdash' x ::= a} \\
\frac{l \vdash' c_1 \quad l \vdash' c_2}{l \vdash' c_1;; c_2} \qquad \frac{\text{sec } b \leq l \quad l \vdash' c_1 \quad l \vdash' c_2}{l \vdash' \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2} \\
\frac{\text{sec } b \leq l \quad l \vdash' c}{l \vdash' \text{WHILE } b \text{ DO } c} \qquad \frac{l \vdash' c \quad l' \leq l}{l' \vdash' c}
\end{array}$$

Figure 11. Definition of $\text{sec-type}' :: \text{nat} \Rightarrow \text{com} \Rightarrow \text{bool}$

$$\begin{array}{c}
\frac{}{\vdash \text{SKIP} : l} \qquad \frac{\text{sec } a \leq \text{sec } x}{\vdash x ::= a : \text{sec } x} \qquad \frac{\vdash c_1 : l_1 \quad \vdash c_2 : l_2}{\vdash c_1;; c_2 : \min l_1 l_2} \\
\frac{\text{sec } b \leq \min l_1 l_2 \quad \vdash c_1 : l_1 \quad \vdash c_2 : l_2}{\vdash \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2 : \min l_1 l_2} \qquad \frac{\text{sec } b \leq l \quad \vdash c : l}{\vdash \text{WHILE } b \text{ DO } c : l}
\end{array}$$

Figure 12. Definition of the bottom-up security type system.

3.2.5. A Bottom-Up Typing System

The type systems presented above are top-down systems: the level l is passed from the context or the user and is checked at assignment commands. We can also give a bottom-up formulation where we compute the smallest l consistent with variable assignments and check this value at *IF* and *WHILE* commands. Instead of *max* computations, we now get *min* computations in Figure 12.

We can read the bottom-up statement $\vdash c : l$ as c has a write-effect of l , meaning that no variable below l is written to in c .

Again, we can prove equivalence. The first direction is straightforward and the proof is automatic.

Lemma 34. $\vdash c : l \implies l \vdash' c$

The second direction needs more care. The statement $l \vdash' c \implies \vdash c : l$ is not true, Isabelle's nitpick tool quickly finds a counter example:

$$0 \vdash' ''x'' ::= N 0, \text{ but } \neg \vdash ''x'' ::= N 0 : 0$$

The standard formulation admits anti-monotonicity, the computation of a minimal l does not. If we take this discrepancy into account, we get the following statement that is then again proved automatically by Isabelle.

Lemma 35. $l \vdash' c \implies \exists l' \geq l. \vdash c : l'$

3.2.6. What about termination?

In the previous section we proved the following security theorem (Theorem 31):

$$\llbracket (c, s) \Rightarrow s'; (c, t) \Rightarrow t'; 0 \vdash c; s = t (\leq l) \rrbracket \implies s' = t' (\leq l)$$

We read it as: *If our type system says yes, our data is safe: there will be no information flowing for high to low variables.*

Is this correct? The formal statement is certainly true, we proved it in Isabelle. But: it doesn't quite mean what the sentence above says. It means only precisely what the formula states: given two terminating executions started in states we can't tell apart, we won't be able to tell apart their final states.

What if we don't have two terminating executions? Consider, for example, the following typing statement.

$$0 \vdash \text{WHILE } \text{Less } (V \text{ "x"}) \text{ (N 1) DO SKIP}$$

This is a true statement, the program is type correct at context level 0. Our noninterference theorem holds. Yet, the program still leaks information: the program will terminate if and only if the higher-security variable x (of level 1) is not 0. We can infer information about the contents of a secret variable by observing termination.

This is also called a **covert channel**, that is, an information flow channel that is not part of the security theorem or even the security model, and that therefore the security theorem does not make any claims about.

In our case, termination is observable in the model, but not in the theorem, because it already assumes two terminating executions from the start. An example of a more traditional covert channel is timing. Consider the standard `strcmp` function in C that compares two strings: it goes through the strings from left to right, and terminates with false as soon as two characters are not equal. The more characters are equal in the prefix of the strings, the longer it takes to execute this function. This time can be measured and the timing difference is significant enough to be statistically discernible even over network traffic. Such timing attacks can even be effective against widely deployed cryptographic algorithms, for instance as used in SSL [2].

Covert channels and the strength of security statements are the bane of security proofs. The literature is littered with the bodies of security theorems that have been broken, either because their statement was weak, their proof was wrong, or because the model made unreasonably strong assumptions, i.e. admitted too many obvious covert channels.

Conducting security proofs in a theorem prover only helps against one of these: wrong proofs. Strong theorem statements and realistic model assumptions, or at least explicit model assumptions, are still our own responsibility.

So what can we do to fix our statement of security? For one, we could prove separately, and manually, that the specific programs we are interested in always terminate. Then the problem disappears. Or we could strengthen the type system and its security statement. The key idea is: *WHILE conditions must not depend on confidential data.* If they don't, then termination cannot leak information.

In the following, we formalise and prove this idea.

Formalising our idea means we replace the *WHILE*-rule with a new one that does not admit anything higher than level 0 in the condition:

$$\frac{\text{sec } b = 0 \quad 0 \vdash c}{0 \vdash \text{WHILE } b \text{ DO } c}$$

$$\begin{array}{c}
\frac{}{l \vdash \text{SKIP}} \quad \frac{\text{sec } a \leq \text{sec } x \quad l \leq \text{sec } x}{l \vdash x ::= a} \quad \frac{l \vdash c_1 \quad l \vdash c_2}{l \vdash c_1;; c_2} \\
\frac{\text{max } (\text{sec } b) \ l \vdash c_1 \quad \text{max } (\text{sec } b) \ l \vdash c_2}{l \vdash \text{IF } b \ \text{THEN } c_1 \ \text{ELSE } c_2} \quad \frac{\text{sec } b = 0 \quad 0 \vdash c}{0 \vdash \text{WHILE } b \ \text{DO } c}
\end{array}$$

Figure 13. Termination-sensitive security type system.

This is already it. [Figure 13](#) shows the full set of rules, putting the new one into context.

We now need to change our noninterference statement such that it takes termination into account. The interesting case was where one execution terminated and the other didn't. If both executions terminate, our previous statement already applies, if both do not terminate then there is no information leakage, because there is nothing to observe.¹ So, since our statement is symmetric, we now assume *one* terminating execution, a well-typed program of level 0 as before, and two start states that agree up to level l , also as before. We then have to show that the other execution *also* terminates and that the final states still agree up to level l .

$$\llbracket (c, s) \Rightarrow s'; 0 \vdash c; s = t (\leq l) \rrbracket \Longrightarrow \exists t'. (c, t) \Rightarrow t' \wedge s' = t' (\leq l)$$

We build up the proof of this new theorem in the same way as before. The first property is again anti-monotonicity, which still holds.

Lemma 36 (Anti-monotonicity).

$$\llbracket l \vdash c; l' \leq l \rrbracket \Longrightarrow l' \vdash c$$

Proof. The proof is by induction on the typing derivation. Isabelle then solves each of the cases automatically. \square

Our confinement lemma is also still true.

Lemma 37 (Confinement).

$$\llbracket (c, s) \Rightarrow t; l \vdash c \rrbracket \Longrightarrow s = t (< l)$$

Proof. The proof is the same as before, first by induction on the big-step execution, then by using anti-monotonicity in the *IF* cases, and automation on the rest. \square

Before we can proceed to noninterference, we need one new fact about the new type system: any program that is type correct, but not at level 0 (only higher), must terminate. Intuitively that is easy to see: *WHILE* loops are the only cause of potential nontermination, and they can now only be typed at level 0. This means, if the program is type correct at some level, but not at level 0, it does not contain *WHILE* loops.

Lemma 38 (Termination).

$$\llbracket l \vdash c; l \neq 0 \rrbracket \Longrightarrow \exists t. (c, s) \Rightarrow t$$

¹Note that if our programs had output, this case might leak information as well.

Proof. The formal proof of this lemma does not directly talk about the occurrence of while loops, but encodes the argument in a contradiction. We start the proof by induction on the typing derivation. The base cases all terminate trivially, and the step cases terminate because all their branches terminate in the induction hypothesis. In the *WHILE* case we have the contradiction: our assumption says that $l \neq 0$, but the induction rule instantiates l with 0 , and we get $0 \neq 0$. \square

Equipped with these lemmas, we can finally proceed to our new statement of noninterference.

Theorem 39 (Noninterference).

$$\llbracket (c, s) \Rightarrow s'; 0 \vdash c; s = t (\leq l) \rrbracket \implies \exists t'. (c, t) \Rightarrow t' \wedge s' = t' (\leq l)$$

Proof. The proof is similar to the termination-insensitive case, it merely has to additionally show termination of the second command. For *SKIP*, assignment, and semicolon this is easy, the first two because they trivially terminate, the second, because Isabelle can put the induction hypotheses together automatically.

The *IF* case is slightly more interesting. If the condition does not depend on secret variables, the induction hypothesis is strong enough for us to conclude the goal directly. However, if the condition does depend on secret variables, i.e. $\neg \text{sec } b \leq l$, we make use of confinement again, as we did in our previous proof. However, we first have to show that the second execution terminates, i.e. that a final state exists. This follows from our termination lemma and the fact that if the security level $\text{sec } b$ is greater than l , it cannot be 0 . The rest goes through as before.

The *WHILE* case becomes easier than in our previous proof. Since we know from the typing statement that the boolean expression does not contain any high variables, we know that the loops started in s and t will continue to make the same decision whether to terminate or not. That was the whole point of our type system change. In the *WhileFalse* case that is all that is needed, in the *WhileTrue* case, we can make use of this fact to access the induction hypothesis: from the fact that the loop is type correct at level 0 , we know by rule inversion that $0 \vdash c$. We also know, by virtue of being in the *WhileTrue* case, that $\text{bval } b \ s, (c, s) \Rightarrow s''$, and $(w, s'') \Rightarrow s'$. We now need to construct a terminating execution of the loop starting in t , ending in some state t' that agrees with s' below l . We start by noting $\text{bval } b \ t$ using noninterference for boolean expressions. Per induction hypothesis we conclude that there is a t'' with $(c, t) \Rightarrow t''$ that agrees with s'' below l . Using the second induction hypothesis, we repeat the process for w , and conclude that there must be such a t' that agrees with s' below l . \square

The predicate $l \vdash c$ is phrased to be executable. The standard formulation, however, is again slightly different, replacing the maximum computation by the anti-monotonicity rule. Figure 14 introduces the standard system.

As before, we can show equivalence with our formulation.

Lemma 40 (Equivalence to standard formulation).

$$l \vdash c \iff l' \vdash c$$

$$\begin{array}{c}
\frac{}{l \vdash' \text{SKIP}} \quad \frac{\text{sec } a \leq \text{sec } x \quad l \leq \text{sec } x}{l \vdash' x ::= a} \quad \frac{l \vdash' c_1 \quad l \vdash' c_2}{l \vdash' c_1;; c_2} \\
\frac{\text{sec } b \leq l \quad l \vdash' c_1 \quad l \vdash' c_2}{l \vdash' \text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2} \quad \frac{\text{sec } b = 0 \quad 0 \vdash' c}{0 \vdash' \text{WHILE } b \text{ DO } c} \\
\frac{l \vdash' c \quad l' \leq l}{l' \vdash' c}
\end{array}$$

Figure 14. Termination-sensitive security type system — standard formulation.

Proof. As with the equivalence proofs of different security type system formulations in previous sections, this proof goes first by considering each direction of the if-and-only-if separately, and then by induction on the type system in the assumption of that implication. As before, Isabelle then proves each sub case of the respective induction automatically. \square

3.3. Summary and Further Reading

In this section we have analysed two kinds of type systems: a standard type system that tracks types of values and prevents type errors at run time, and a security type system that prevents information flow from higher-level to lower-level variables.

Sound, static type systems enjoy widespread application in popular programming languages such as Java, C#, Haskell, and ML, but also on low-level languages such as the Java Virtual Machine and its bytecode verifier [8]. Some of these languages require types to be declared explicitly, such as in Java. In other languages, such as Haskell, these declarations can be left out, and types are inferred automatically.

The purpose of type systems is to prevent errors. In essence, a type derivation is a proof, which means type checking performs basic automatic proofs about programs.

The second type system we explored ensured absence of information flow. The field of language-based security is substantial [15]. As mentioned, the type system and the soundness statement in the sections above go back to Volpano, Irvine, and Smith [18]. While language-based security had been investigated before Volpano et al, 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. As we have seen, such non-trivial properties are comfortably within the reach of machine-checked interactive proof. Our type system deviated a little from the standard presentation of Volpano et al: we derive anti-monotonicity as a lemma, whereas Volpano, Irvine, and Smith have it as a typing rule. In exchange, they can avoid an explicit *max* calculation. We saw that our syntax directed form of the rules is equivalent and allowed us to execute examples. We also mentioned that our simple security levels based on natural numbers can be generalised to arbitrary lattices. This observation goes back to Denning [4].

A popular alternative to security type systems is dynamic tracking of information flows, or so-called **taint analysis** [17]. It has been long-time folklore in the

field that static security analysis of programs must be more precise than dynamic analysis, because dynamic (run-time) analysis can only track one execution of the program at a time, whereas the soundness property of our static type system for instance compares two executions. Many dynamic taint analysis implementations to date do not track implicit flows. Sabelfeld and Russo showed for termination-insensitive noninterference that this is *not* a theoretical restriction, and dynamic monitoring can in fact be more precise than the static type system [16]. However, since their monitor essentially turns implicit flow-violations into non-termination, the question is still open for the more restrictive termination-sensitive case. For more sophisticated, so-called *flow-sensitive* type systems, the dynamic and static versions are incomparable: there are some programs where purely dynamic flow-sensitive analysis fails, but the static type system succeeds, and the other way around [14].

The name non-interference was coined by Goguen and Meseguer [5], but the property goes back further to Ellis Cohen who called its inverse *Strong Dependency* [3]. The concept of covert information flow channels already precedes this idea [7]. Non-interference can be applied beyond language-based security, for instance by directly proving the property about a specific system. This is interesting for systems that have inherent security requirements and are written in low-level languages such as C or in settings where the security policy cannot directly be attached to variables in a program. Operating systems are an example of this class, where the security policy is configurable at runtime. It is feasible to prove such theorems in Isabelle down to the C code level: the seL4 microkernel is an operating system kernel with such a non-interference theorem in Isabelle [11].

4. Program Analysis

Program analysis, also known as *static analysis*, describes a whole field of techniques for the static (i.e. compile-time) analysis of programs. Most compilers or programming environments perform more or less ambitious program analyses. The two most common objectives are the following:

Optimisation The purpose is to improve the behaviour of the program, usually by reducing its running time or space requirements.

Error detection The purpose is to detect common programming errors that lead to runtime exceptions or other undesirable behaviour.

Program optimisation is a special case of program transformation (for example for code refactoring) and consists of two phases: the analysis (to determine if certain required properties hold) and the transformation.

There are a number of different approaches to program analysis that employ different techniques to achieve similar aims. In the previous section we used type systems for error detection. In this section we employ what is known as *data-flow analysis*. We study two analyses (and associated transformations):

1. Definite initialisation analysis determines if all variables have been initialised before they are read. This falls into the category of error detection analyses. There is no transformation.
2. Constant folding is an optimisation that tries to replace variables by constants. For example, the second assignment in `x := 1; y := x` can be replaced by the (typically faster) `y := 1`.

Throughout this section we continue the naive approach to program analysis that ignores boolean conditions. That is, we treat them as nondeterministic: we assume that both values are possible every time the conditions are tested. More precisely, our analyses are correct w.r.t. a (big or small-step) semantics where we have simply dropped the preconditions involving boolean expressions from the rules, thus resulting in a nondeterministic language.

Limitations

Program analyses, no matter what techniques they employ, are always limited. This is a consequence of Rice's Theorem from computability theory. It roughly tells us that *Nontrivial semantic properties of programs (e.g. termination) are undecidable*. That is, no semantic property P has a magic analyser that

- terminates on every input program,
- only says Yes if the input program has property P (correctness),
- only says No if the input program does not have property P (completeness).

For concreteness, let us consider definite initialisation analysis of the following program:

```
FOR ALL positive integers x, y, z, n DO
  IF n > 2 ∧ xn + yn = zn THEN u := u ELSE SKIP
```

For convenience we have extended our programming language with a **FOR ALL** loop and an exponentiation operation: both could be programmed in pure IMP, although it would be painful. The program searches for a counterexample to Fermat's conjecture that no three positive integers x , y , and z can satisfy the equation $x^n + y^n = z^n$ for any integer $n > 2$. It reads the uninitialised variable u (thus violating the definite initialisation property) iff such a counterexample exists. It would be asking a bit much from a humble program analyser to determine the truth of a statement that was in the Guinness Book of World Records for "most difficult mathematical problems" prior to its 1995 proof by Wiles.

As a consequence, we cannot expect program analysers to terminate, be correct and be complete. Since we do not want to sacrifice termination and correctness, we sacrifice completeness: we allow analysers to say No although the program has the desired semantic property but the analyser was unable to determine that.

4.1. Definite Initialisation Analysis

The first program analysis we investigate is called **definite initialisation**. The Java Language Specification has the following to say on definite initialisation. [6, chapter 16, p. 527]

Each local variable [...] must have a definitely assigned value when any access of its value occurs. [...] A compiler must carry out a specific conservative flow analysis to make sure that, for every access of a local variable [...] f , f is definitely assigned before the access; otherwise a compile-time error must occur.

Java was the first mainstream language to force programmers to initialise their variables.

In most programming languages, objects allocated on the heap are automatically initialised to zero or a suitable default value, but local variables are not. Uninitialised variables are a common cause of program defects that are very hard to find. A C program for instance, that uses an uninitialised local integer variable will not necessarily crash on the first access to that integer. Instead, it will read the value that is stored there by accident. On the developer's machine and operating system that value may happen to be zero and the defect will go unnoticed. On the user's machine, that same memory may contain different values left over from a previous run or from a different application. What is more, this random value might not directly lead to a crash either, but only cause misbehaviour at a much later point of execution, leading to bug reports that are next to impossible to reproduce for the developer.

Removing the potential for such errors automatically is the purpose of the definite initialisation analysis.

Consider the following example with an already initialised x .

```
IF  $x < 1$  THEN  $y := x$  ELSE  $y := x + 1$ ;  $y := y + 1$ 
IF  $x < x$  THEN  $y := y + 1$  ELSE  $y := x$ ;  $y := y + 1$ 
```

The first line is clearly fine: in both branches of the IF, y gets initialised before it is used in the statement after. The second line is also fine: even though the *True* branch uses y where it is potentially uninitialised, we know that the *True* branch can never be taken. However, we only know that, because we can prove that $x < x$ will always be *False*.

What about the following example? Assume x and y are initialised.

```
WHILE  $x < y$  DO  $z := x$ ;  $z := z + 1$ 
```

Here it depends: if $x < y$, the program is fine (it will never terminate, but at least it does so without using uninitialised variables), but if $x < y$ is not the case, the program is unsafe. So, if our goal is to reject all *potentially* unsafe programs, we have to reject this one.

As mentioned in the introduction, we do not analyse boolean expressions statically to make predictions about program execution. Instead we take both potential outcomes into account. This means, the analysis we are about to develop will only accept the first program, but reject the other two.

Java is more discerning in this case, and will perform the optimisation of **constant folding**, which we discuss in [Section 4.2](#), before definite initialisation analysis. If during that pass it turns out an expression is always *True* or always *False*, this can be taken into account. This is a nice example of positive interaction

between different kinds of optimisation and program analysis, where one enhances the precision and predictive power of the other.

As discussed, we cannot hope for completeness of any program analysis, so there will be cases of safe programs that are rejected. For this specific analysis, this is usually the case when the programmer is smarter than the boolean constant folding the compiler performs. As with any restriction in a programming language, some programmers will complain about the shackles of definite initialisation analysis, and Java developer forums certainly contain such complaints. Completely eliminating this particularly hard-to-find class of Heisenbugs well justifies the occasional program refactoring, though.

In the following sections, we construct our definite initialisation analysis, define a semantics where initialisation failure is observable, and then proceed to prove the analysis correct by showing that these failures will not occur.

4.1.1. Definite Initialisation

The Java Language Specification quotes a number of rules that definite initialisation analysis should implement to achieve the desired result. They have the following form (adjusted for IMP):

Variable x is definitely initialised after *SKIP*
iff x is definitely initialised before *SKIP*.

Similar statements exist for each language construct. Our task is simply to formalise them. Each of these rules talks about variables, or more precisely sets of variables. For instance, to check an assignment statement, we will want to start with a set of variables that is already initialised, we will check that set against the set of variables that is used in the assignment expression, and we will add the assigned variable to the initialised set after the assignment has completed.

So, the first formal tool we need is the set of variables mentioned in an expression. The Isabelle theory *Vars* provides an overloaded function *vars* for this:

```

fun vars :: aexp  $\Rightarrow$  vname set where
  vars (N n)      = {}
  vars (V x)      = {x}
  vars (Plus a1 a2) = vars a1  $\cup$  vars a2

fun vars :: bexp  $\Rightarrow$  vname set where
  vars (Bc v)     = {}
  vars (Not b)    = vars b
  vars (And b1 b2) = vars b1  $\cup$  vars b2
  vars (Less a1 a2) = vars a1  $\cup$  vars a2

```

With this we can define our main definite initialisation analysis. The purpose is to check whether each variable in the program is assigned to before it is used. This means we ultimately want a predicate of type $com \Rightarrow bool$, but we have

$$\begin{array}{c}
\frac{}{D A \text{ SKIP } A} \quad \frac{\text{vars } a \subseteq A}{D A (x ::= a) (\text{insert } x A)} \\
\frac{D A_1 c_1 A_2 \quad D A_2 c_2 A_3}{D A_1 (c_1;; c_2) A_3} \\
\frac{\text{vars } b \subseteq A \quad D A c_1 A_1 \quad D A c_2 A_2}{D A (\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2) (A_1 \cap A_2)} \\
\frac{\text{vars } b \subseteq A \quad D A c A'}{D A (\text{WHILE } b \text{ DO } c) A}
\end{array}$$

Figure 15. Definite initialisation $D :: \text{vname set} \Rightarrow \text{com} \Rightarrow \text{vname set} \Rightarrow \text{bool}$

already seen in the examples that we need a slightly more general form for the computation itself. In particular, we carry around a set of variables that we know are definitely initialised at the beginning of a command. The analysis then has to do two things: check whether the command only uses these variables, and produce a new set of variables that we know are initialised afterwards. This leaves us with the following type signature:

$$D :: \text{vname set} \Rightarrow \text{com} \Rightarrow \text{vname set} \Rightarrow \text{bool}$$

We want the notation $D A c A'$ to mean:

If all variables in A are initialised before c is executed, then no uninitialised variable is accessed during execution, and all variables in A' are initialised afterwards.

Figure 15 shows how we can inductively define this analysis with one rule per syntactic construct. We walk through them step by step:

- The *SKIP* rule is obvious, and translates exactly the text rule we have mentioned above.
- Similarly, the assignment rule follows our example above: the predicate $D A (x ::= a) A'$ is *True* if the variables of the expression a are contained in the initial set A , and if A' is precisely the initial A plus the variable x we just assigned to.
- Sequential composition has the by now familiar form: we simply pass through the result A_2 of c_1 to c_2 , and the composition is definitely initialised if both commands are definitely initialised.
- In the *IF* case, we check that the variables of the boolean expression are all initialised, and we check that each of the branches is definitely initialised. We pass back the intersection of the results produced by c_1 and c_2 , because we do not know which branch will be taken at runtime. If we were to analyse boolean expression more precisely, we could introduce further case distinctions into this rule.
- Finally, the *WHILE* case. It also checks that the variables in the boolean expression are all in the initialised set A , and it also checks that the command c is definitely initialised starting in the same set A , but it ignores the

result A' of c . Again, this must be so, because we have to be conservative: it is possible that the loop will never be executed at runtime, because b may be already *False* before the first iteration. In this case no additional variables will be initialised, no matter what c does. It may be possible for specific loop structures, such as for-loops to statically determine that their body will be executed at least once, but no mainstream language currently does that.

We can now decide whether a command is definitely initialised, namely exactly when we can start with the empty set of initialised variables and find a resulting set such the our inductive predicate D is *True*:

$$\mathcal{D} c = (\exists A'. D \{ \} c A')$$

Defining a program analysis such as definite initialisation by an inductive predicate makes the connection to type systems clear: in a sense, all program analyses can be phrased as sufficiently complex type systems. Since our rules are syntax directed, they also directly suggest a recursive execution strategy. In fact, for this analysis it is straightforward to turn the inductive predicate into two recursive functions in Isabelle that compute our set A' if it exists, and check whether all expressions mention only initialised variables. We leave this recursive definition and proof of equivalence as an exercise to the reader and turn our attention to proving correctness of the analysis instead.

4.1.2. Initialisation Sensitive Expression Evaluation

As in type systems, to phrase what correctness of the definite initialisation analysis means, we first have to identify what could possibly go wrong.

Here, this is easy: we should observe an error when the program uses a variable that has not been initialised. That is, we need a new, finer-grained semantics that keeps track which variables have been initialised and leads to an error if the program accesses any other variable.

To that end, we enrich our set of values with an additional element that we will read as *uninitialised*. Isabelle provides the *option* data type for this:

```
datatype 'a option = None | Some 'a
```

We simply redefine our program state as

```
type-synonym state = vname  $\Rightarrow$  val option
```

and take *None* as the uninitialised value. The *option* data type comes with additional useful notation: $s(x \mapsto y)$ means $s(x := \text{Some } y)$, and $\text{dom } s = \{a. s a \neq \text{None}\}$.

Now that we can distinguish initialised from uninitialised values, we can check the evaluation of expressions. We have had a similar example of potentially failing expression evaluation in type systems in [Section 3.1](#). There we opted for an inductive predicate, reasoning that in the functional style where we would return *None* for failure, we would have to consider all failure cases explicitly. This argument also holds here. Nevertheless, for the sake of variety, we will this time show the

functional variant with *option*. It is less elegant, but not so horrible as to become unusable. It has the advantage of being functional, and therefore easier to apply automatically in proofs.

```

fun aval :: aexp ⇒ state ⇒ val option where
aval (N i) s      = Some i
aval (V x) s      = s x
aval (Plus a1 a2) s = (case (aval a1 s, aval a2 s) of
                          (Some i1, Some i2) ⇒ Some(i1+i2)
                          | - ⇒ None)

fun bval :: bexp ⇒ state ⇒ bool option where
bval (Bc v) s      = Some v
bval (Not b) s      = (case bval b s of
                       None ⇒ None | Some bv ⇒ Some(¬ bv))
bval (And b1 b2) s = (case (bval b1 s, bval b2 s) of
                       (Some bv1, Some bv2) ⇒ Some(bv1 ∧ bv2)
                       | - ⇒ None)
bval (Less a1 a2) s = (case (aval a1 s, aval a2 s) of
                       (Some i1, Some i2) ⇒ Some(i1 < i2)
                       | - ⇒ None)

```

We can reward ourselves for all these case distinctions with two concise lemmas that confirm that expressions indeed evaluate without failure if they only mention initialised variables.

Lemma 41 (Initialised arithmetic expressions).

$\text{vars } a \subseteq \text{dom } s \implies \exists i. \text{aval } a \ s = \text{Some } i$

Lemma 42 (Initialised boolean expressions).

$\text{vars } b \subseteq \text{dom } s \implies \exists bv. \text{bval } b \ s = \text{Some } bv$

Both lemmas are proved automatically after structural induction on the expression.

4.1.3. Initialisation Sensitive Small-Step Semantics

From here, the development towards the correctness proof is standard: we define a small-step semantics, and we prove progress and preservation as we would for a type system.

In fact, the development is so standard that we only show the small-step semantics in [Figure 16](#) and give one hint for the soundness proof. It needs the following lemma.

Lemma 43 (*D* is increasing). $D \ A \ c \ A' \implies A \subseteq A'$

Proof. This lemma holds independently of the small-step semantics. The proof is automatic after structural induction on *c*. \square

$$\begin{array}{c}
\frac{aval\ a\ s = Some\ i}{(x ::= a, s) \rightarrow (SKIP, s(x \mapsto i))} \\
\\
\frac{}{(SKIP;; c, s) \rightarrow (c, s)} \quad \frac{(c_1, s) \rightarrow (c_1', s')}{(c_1;; c_2, s) \rightarrow (c_1';; c_2, s')} \\
\\
\frac{bval\ b\ s = Some\ True}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \rightarrow (c_1, s)} \\
\\
\frac{bval\ b\ s = Some\ False}{(IF\ b\ THEN\ c_1\ ELSE\ c_2, s) \rightarrow (c_2, s)} \\
\\
\frac{}{(WHILE\ b\ DO\ c, s) \rightarrow (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP, s)}
\end{array}$$

Figure 16. Small-step semantics, initialisation sensitive

The soundness statement then is as in the type system in [Section 3.1](#).

Theorem 44 (*D* is sound).

If $(c, s) \rightarrow^* (c', s')$ and $D (dom\ s)\ c\ A'$ then $(\exists cs''. (c', s') \rightarrow cs'') \vee c' = SKIP$.

The proof goes by showing progress and preservation separately and making use of [Lemma 43](#). We leave its details as an exercise and present an alternative way of proving soundness of the definite initialisation analysis in the next section instead.

4.1.4. Initialisation Sensitive Big-Step Semantics

In the previous section we learned that a formalisation in the small-step style and a proof with progress and preservation as we know them from type systems are sufficient to prove correctness of definite initialisation. In this section, we investigate how to adjust a big-step semantics such that it can be used for the same purpose of proving the definite initialisation analysis correct. We will see that this is equally possible and that big-step semantics can be used for such proofs. This may be attractive for similar kinds of correctness statements, because big-step semantics are often easier to write down. However, we will also see the price we have to pay: a larger number of big-step rules and therefore a larger number of cases in inductive proofs about them.

The plan for adjusting the big-step semantics is simple: we need to be able to observe error states, so we will make errors explicit and propagate them to the result. Formally, we want something of the form

$$com \times state \Rightarrow state\ option$$

where *None* would indicate that an error occurred during execution, in our case that the program accessed an uninitialised variable.

There is a small complication with the type above. Consider for instance this attempt to write the semicolon rule.

$$\frac{(c_1, s_1) \Rightarrow \text{Some } s_2 \quad (c_2, s_2) \Rightarrow s}{(c_1;; c_2, s_1) \Rightarrow s} \qquad \frac{(c_1, s_1) \Rightarrow \text{None}}{(c_1;; c_2, s_1) \Rightarrow \text{None}}$$

There is no problem with the soundness of these rules. The left rule is the case where no error occurs, the right rule terminates the execution when an error does occur. The problem is that we will need at least these two cases for any construct that has more than one command. It would be nicer to just specify once and for all how error propagates.

We can make the rules more compositional by ensuring that the result type is the same as the start type for an execution, i.e. that we can plug a result state directly into the start of the next execution without any additional operation or case distinction for unwrapping the *option* type. We achieve this by making the start type *state option* as well.

$$\text{com} \times \text{state option} \Rightarrow \text{state option}$$

We can now write one rule that defines how error (*None*) propagates:

$$(c, \text{None}) \Rightarrow \text{None}$$

Consequently, in the rest of the semantics in [Figure 17](#) we only have to locally consider the case where we directly produce an error, and the case of normal execution. An example of the latter is the assignment rule, where we update the state as usual if the arithmetic expression evaluates normally:

$$\frac{\text{aval } a \text{ } s = \text{Some } i}{(x ::= a, \text{Some } s) \Rightarrow \text{Some } (s(x \mapsto i))}$$

An example of the former is the assignment rule, where expression evaluation leads to failure:

$$\frac{\text{aval } a \text{ } s = \text{None}}{(x ::= a, \text{Some } s) \Rightarrow \text{None}}$$

The remaining rules in [Figure 17](#) follow the same pattern. They only have to worry about producing errors, not about propagating them.

If we are satisfied that this semantics encodes failure for accessing uninitialised variables, we can proceed to proving correctness of our program analysis *D*.

The statement we want in the end is, paraphrasing Milner, *well-initialised programs cannot go wrong*.

$$\llbracket D (\text{dom } s) \text{ } c \text{ } A'; (c, \text{Some } s) \Rightarrow s' \rrbracket \Longrightarrow s' \neq \text{None}$$

The plan is to use rule induction on the big-step semantics to prove this property directly, without the detour over progress and preservation. Looking at the rules for $D \ A \ c \ A'$, it is clear that we will not be successful with a constant pattern of *dom s* for *A*, because the rules produce different patterns. This means, both

$$\begin{array}{c}
\frac{}{(c, \text{None}) \Rightarrow \text{None}} \quad \frac{}{(\text{SKIP}, s) \Rightarrow s} \\
\\
\frac{\text{aval } a \text{ } s = \text{Some } i}{(x ::= a, \text{Some } s) \Rightarrow \text{Some } (s(x \mapsto i))} \quad \frac{\text{aval } a \text{ } s = \text{None}}{(x ::= a, \text{Some } s) \Rightarrow \text{None}} \\
\\
\frac{(c_1, s_1) \Rightarrow s_2 \quad (c_2, s_2) \Rightarrow s_3}{(c_1;; c_2, s_1) \Rightarrow s_3} \\
\\
\frac{\text{bval } b \text{ } s = \text{Some True} \quad (c_1, \text{Some } s) \Rightarrow s'}{(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \text{Some } s) \Rightarrow s'} \\
\\
\frac{\text{bval } b \text{ } s = \text{Some False} \quad (c_2, \text{Some } s) \Rightarrow s'}{(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \text{Some } s) \Rightarrow s'} \\
\\
\frac{\text{bval } b \text{ } s = \text{None}}{(\text{IF } b \text{ THEN } c_1 \text{ ELSE } c_2, \text{Some } s) \Rightarrow \text{None}} \\
\\
\frac{\text{bval } b \text{ } s = \text{Some False}}{(\text{WHILE } b \text{ DO } c, \text{Some } s) \Rightarrow \text{Some } s} \\
\\
\frac{\text{bval } b \text{ } s = \text{Some True} \quad (c, \text{Some } s) \Rightarrow s' \quad (\text{WHILE } b \text{ DO } c, s') \Rightarrow s''}{(\text{WHILE } b \text{ DO } c, \text{Some } s) \Rightarrow s''} \\
\\
\frac{\text{bval } b \text{ } s = \text{None}}{(\text{WHILE } b \text{ DO } c, \text{Some } s) \Rightarrow \text{None}}
\end{array}$$

Figure 17. Big-step semantics with error propagation

A and A' need to be variables in the statement to produce suitably general induction hypotheses. Replacing $\text{dom } s$ with a plain variable A in turn means we have to find a suitable side condition such that our statement remains true, and we have show that this side condition is preserved. A suitable such condition is $A \subseteq \text{dom } s$, i.e. it is OK if our program analysis succeeds with fewer variables than are currently initialised in the state. After this process of generalising the statement for induction, we arrive at the following lemma.

Lemma 45 (Soundness of D).

$$\begin{array}{l}
\llbracket (c, \text{Some } s) \Rightarrow s'; D \ A \ c \ A'; A \subseteq \text{dom } s \rrbracket \implies \\
\exists t. s' = \text{Some } t \wedge A' \subseteq \text{dom } t
\end{array}$$

Proof. The proof is by rule induction on $(c, \text{Some } s) \Rightarrow s'$; Isabelle solves all sub-cases but *WHILE* automatically. In the *WHILE* case, we apply the induction hypothesis to the body c manually and can then let the automation figure out the rest. Applying the induction hypothesis to c is interesting, because we need to make use of D 's *increasing* property we proved in Lemma 43. Recall that the D rule for *WHILE* requires $D \ A \ c \ A'$ for the body c . Per induction hypothesis, we get that the result state t after execution of c has the property $A' \subseteq \text{dom } t$. To

apply the induction hypothesis for the rest of the *WHILE* loop, however, we need $A \subseteq \text{dom } t$. From Lemma 43 we know that $A \subseteq A'$ and can therefore proceed. \square

After this proof, we can now better compare the small-step and big-step approaches to showing soundness of *D*: While the small-step semantics is more concise, the soundness proof is longer, and while the big-step semantics has a larger number of rules, its soundness proof is more direct and shorter. As always, the trade-off depends on the particular application. With machine-checked proofs, it is in general better to err on the side of nicer and easier-to-understand definitions than on the side of shorter proofs.

4.2. Constant Folding and Propagation

The previous section presented an analysis that prohibits a common programming error, uninitialised variables. This section presents an analysis that enables program optimisations, namely constant folding and propagation.

Constant folding and constant propagation are two very common compiler optimisations. Constant folding means computing the value of constant (sub) expressions at compile time and substituting their value for the computation. Constant propagation means determining if a variable has constant value, and propagating that constant value to the use-occurrences of that variable, for instance to perform further constant folding:

```
x = 42 - 5;  
y = x * 2
```

In the first line, the compiler would fold the expression `42 - 5` into its value `37`, and in the second line, it would propagate this value into the expression `x * 2` to replace it with `74` and arrive at

```
x = 37;  
y = 74
```

Further liveness analysis could then for instance conclude that `x` is not live in the program and can therefore be eliminated, which frees up one more register for other local variables and could thereby improve time as well as space performance of the program.

Constant folding can be especially effective when used on boolean expressions, because it allows the compiler to recognise and eliminate further dead code. A common pattern is something like

```
IF debug THEN debug_command ELSE SKIP
```

where `debug` is a global constant that if set to `False` could eliminate debugging code from the program. Other common uses are the explicit construction of constants from their constituents for documentation and clarity.

Despite its common use for debug statements as above, we stay with our general policy in this section and will *not* analyse boolean expressions for constant

folding. Instead, we leave it as a medium-sized exercise project for the reader to apply the techniques covered in this section.

The semantics of full-scale programming language can be tricky for constant folding (which is why one should prove correctness, of course). For instance, folding of floating point operations may depend on the rounding mode of the machine, which may only be known at run time. Some languages demand that errors such as arithmetic overflow or division by zero are preserved and raised at runtime, others may allow the compiler to refuse to compile such programs, yet others allow the compiler to silently produce any code it likes in those cases.

A widely known tale of caution for constant folding is that of the Intel Pentium FDIV bug in 1994 which led to a processor recall costing Intel roughly half a billion US dollars. In processors exhibiting the fault, the FDIV instruction would perform an incorrect floating point division for some specific operands (10^{37} combinations would lead to wrong results). Constant folding was not responsible for this bug, but it gets its mention in the test for the presence of the FDIV problem. To make it possible for consumers to figure out if they had a processor exhibiting the defect, a number of small programs were written that performed the division with specific operands known to trigger the bug. Testing for the incorrect result, the program would then print a message whether the bug was present or not.

If a developer compiled this test program naïvely, the compiler would perform this computation statically and optimise the whole program to a binary that just returned a constant **yes** or **no**. This way, every single computer in a whole company could be marked as defective, even though only the developer's CPU actually had the bug.

In all of this, the compiler was operating entirely correctly, and would have acted the same way if it was proved correct. We see that our proofs critically rely on the extra-logical assumption that the hardware behaves as specified. Usually, this assumption underlies everything programmers do. However, trying to distinguish correct from incorrect hardware under the assumption that the hardware is correct, is not a good move.

In the following, we are not attempting to detect defective hardware, and can focus on how constant propagation works, how it can be formalised, and how it can be proved correct.

4.2.1. Folding

As usual, we begin with arithmetic expressions. The first optimisation is pure constant folding: the aim is to write a function that takes an arithmetic expression and statically evaluates all constant sub expressions within it. However, since we are going to mix constant folding with constant propagation, if we know the constant value of a variable by propagation, we should use it. To do this, we keep a table or environment that tells us which variables we know to have constant value, and what that value is. This is the same technique we already used in type systems and other static analyses.

type-synonym $tab = vname \Rightarrow val\ option$

We can now formally define our new function *afold* that performs constant folding on arithmetic expressions under the assumption that we already know constant values for some of the variables.

```

fun afold :: aexp ⇒ tab ⇒ aexp where
afold (N n) _      = N n
afold (V x) t     = (case t x of None ⇒ V x | Some x ⇒ N x)
afold (Plus e1 e2) t = (case (afold e1 t, afold e2 t) of
                               (N n1, N n2) ⇒ N (n1+n2)
                               | (e1' , e2') ⇒ Plus e1' e2')

```

For example, the value of *afold* (Plus (V "x") (N 3)) *t* now depends on the value of *t* at "x". If *t* "x" = Some 5, for instance, *afold* will return N 8. If nothing is known about "x", i.e. *t* "x" = None, then we get back the original Plus (V "x") (N 3).

The correctness criterion for this simple optimisation is that the result of execution with optimisation is the same as without:

$$\text{aval } (\text{afold } a \ t) \ s = \text{aval } a \ s$$

As with type system soundness and its corresponding type environments, however, we need the additional assumption that the static table *t* conforms with, or in this case *approximates*, the runtime state *s*. The idea is again that the static value needs to agree with the dynamic value if the former exists:

definition *approx* *t s* = (∀ *x k*. *t x* = Some *k* → *s x* = *k*)

With this assumption the statement is provable.

Lemma 46. *Correctness of afold*
approx t s ⇒ *aval (afold a t) s* = *aval a s*

Proof. Automatic, after induction on the expression. □

The definitions and the proof reflect that the constant folding part of the folding and propagation optimisation is the easy part. For more complex languages, one would have to consider further operators and cases, but nothing fundamental changes in the structure of proof or definition.

As mentioned, in more complex languages, care must be taken in the definition of constant folding to preserve the failure semantics of that language. For some languages it is permissible for the compiler to return a valid result for an invalid program, for others the program must fail in the right way.

4.2.2. Propagation

At this point, we have a function that will fold constants in arithmetic expressions for us. To lift this to commands for full constant propagation, we just apply the same technique, defining a new function *fold* :: *com* ⇒ *tab* ⇒ *com*. The idea is to take a command and a constant table and produce a new command. The first interesting case in any of these analyses usually is assignment. This is easy here, because we can just use *afold*:

fun <i>defs</i> :: <i>com</i> \Rightarrow <i>tab</i> \Rightarrow <i>tab</i> where	
<i>defs</i> <i>SKIP</i> <i>t</i>	= <i>t</i>
<i>defs</i> (<i>x</i> ::= <i>a</i>) <i>t</i>	= (case <i>afold</i> <i>a</i> <i>t</i> of
	<i>N</i> <i>k</i> \Rightarrow <i>t</i> (<i>x</i> \mapsto <i>k</i>)
	$_ \Rightarrow$ <i>t</i> (<i>x</i> := <i>None</i>))
<i>defs</i> (<i>c</i> ₁ ;; <i>c</i> ₂) <i>t</i>	= (<i>defs</i> <i>c</i> ₂ \circ <i>defs</i> <i>c</i> ₁) <i>t</i>
<i>defs</i> (<i>IF</i> <i>b</i> <i>THEN</i> <i>c</i> ₁ <i>ELSE</i> <i>c</i> ₂) <i>t</i>	= <i>merge</i> (<i>defs</i> <i>c</i> ₁ <i>t</i>) (<i>defs</i> <i>c</i> ₂ <i>t</i>)
<i>defs</i> (<i>WHILE</i> <i>b</i> <i>DO</i> <i>c</i>) <i>t</i>	= <i>t</i> _(- <i>lvars</i> <i>c</i>)

Figure 18. Definition of *defs*.

fold (*x* ::= *a*) *t* = *x* ::= *afold* *a* *t*

What about sequential composition? Given *c*₁;; *c*₂ and *t*, we will still need to produce a new sequential composition, and we will obviously want to use *fold* recursively. The question is, which *t* do we pass to the call *fold* *c*₂ for the second command? We need to pick up any new values that might have been assigned in the execution of *c*₁. This is basically the analysis part of the optimisation, whereas *fold* is the code adjustment.

We define a new function for this job and call it *defs* :: *com* \Rightarrow *tab* \Rightarrow *tab* for *definitions*. Given a command and a constant table, it should give us a new constant table that describes the variables with known constant values after the execution of this command.

Figure 18 shows the main definition. Auxiliary function *lvars* computes the set of variables on the left-hand side of assignments. Function *merge* computes the intersection of two tables:

merge *t*₁ *t*₂ = (λm . if *t*₁ *m* = *t*₂ *m* then *t*₁ *m* else *None*)

Let's walk through the equations of *defs* one by one.

- For *SKIP* there is nothing to do, as usual.
- In the assignment case, we attempt to perform constant folding on the expression. If this is successful, i.e. if we get a constant, we note in the result that the variable has a known value. Otherwise, we note that the variable does not have a known value, even if it might have had one before.
- In the semicolon case, we return the effect of *c*₂ under the table we get from *c*₁.
- In the *IF* case, we can only determine the values of variables with certainty if they have been assigned the same value after both branches, hence our use of the table intersection *merge* defined above.
- The *WHILE* case, as almost always, is interesting. Since we don't know statically whether we will ever execute the loop body, we cannot add any new variable assignments to the table. The situation is even worse, though. We need to *remove* all values from the table that are for variables mentioned on the left-hand side of assignment statements in the loop body, because they may contradict what the initial table has stored. A plain merge as in the *IF* case would not be strong enough, because it would only cover the

first iteration. Depending on the behaviour of the body, a different value might be assigned to a variable in the body in a later iteration. Unless we employ a full static analysis on the loop body, which constant propagation usually does not, we need to be conservative. The formalisation achieves this by first computing the names of all variables on the left-hand side of assignment statements in c by means of $lvars$, and by then restricting the table to the complement of that set. The notation $t \upharpoonright_S$ is defined as follows.

$$t \upharpoonright_S = (\lambda x. \text{if } x \in S \text{ then } t \ x \text{ else } None)$$

With all these auxiliary definitions in place, our definition of $fold$ is now as expected. In the *WHILE* case, we fold the body recursively, but again restrict the set of variables to those not written to in the body.

```

fun fold :: com  $\Rightarrow$  tab  $\Rightarrow$  com where
fold SKIP _ = SKIP
fold (x ::= a) t = x ::= afold a t
fold (c1;; c2) t = fold c1 t;; fold c2 (defs c1 t)
fold (IF b THEN c1 ELSE c2) t = IF b THEN fold c1 t ELSE fold c2 t
fold (WHILE b DO c) t = WHILE b DO fold c (t \upharpoonright_{(- lvars c)})

```

Let's test these definitions with some sample executions. Our first test is the first line in the example program at the beginning of this section. The program was:

```

x = 42 - 5;
y = x * 2

```

In IMP, the first line can be encoded as $"x" ::= Plus (N 42) (N -5)$. Running $fold$ on this with the *empty* table gives us $"x" ::= N 37$. This is correct. Encoding the second line as a *Plus* in IMP, and running $fold$ on it in isolation with the empty table should give us no simplification at all, and this is what we get: $"y" ::= Plus (V "x") (V "x")$. However, if we provide a table that sets x to some value, say 1, we should get a simplified result: $"y" ::= N 2$. Finally, testing propagation over semicolon, we run the whole statement with the empty table and get $"x" ::= N 37$; $"y" ::= N 74$. This is also as expected.

As always in these notes, programming and testing are not enough. We want proof that constant folding and propagation are correct. In this case we are performing a program transformation, so our notion of correctness is semantic equivalence.

Eventually, we are aiming for the following statement, where *empty* is the empty table, defined by the abbreviation $empty \equiv \lambda x. None$.

$$fold\ c\ empty \sim c$$

Since all our definitions are recursive in the commands, the proof plan is to proceed by induction on the command. Unsurprisingly, we need to generalise the statement from empty tables to arbitrary tables t . Further, we need to add a side condition for this t , namely the same as in our lemma about expressions: t needs to approximate the state s the command runs in. This leads us to the following interlude on equivalence of commands *up to a condition*.

4.2.3. Conditional Equivalence

This section describes a generalisation of the equivalence of commands, where commands do not need to agree in their executions for *all* states, but only for those states that satisfy a precondition. In [Section 2.4.4](#), we defined

$$(c \sim c') = (\forall s t. (c, s) \Rightarrow t = (c', s) \Rightarrow t)$$

Extending this concept to take a condition P into account is straightforward. We read $P \models c \sim c'$ as *c is equivalent to c' under the assumption P*.

definition

$$(P \models c \sim c') = (\forall s s'. P s \longrightarrow (c, s) \Rightarrow s' = (c', s) \Rightarrow s')$$

We can do the same for boolean expressions:

definition

$$(P \models b <\sim> b') = (\forall s. P s \longrightarrow \text{bval } b s = \text{bval } b' s)$$

Clearly, if we instantiate P to the predicate that returns *True* for all states, we get our old concept of unconditional semantic equivalence back.

Lemma 47. $((\lambda \tau. \text{True}) \models c \sim c') = (c \sim c')$

Proof. By unfolding definitions. □

For any fixed predicate, our new definition is an equivalence relation, i.e. it is reflexive, symmetric, and transitive.

Lemma 48 (Equivalence Relation).

$$\begin{aligned} P \models c \sim c \\ (P \models c \sim c') &= (P \models c' \sim c) \\ \llbracket P \models c \sim c'; P \models c' \sim c'' \rrbracket &\Longrightarrow P \models c \sim c'' \end{aligned}$$

Proof. Again automatic after unfolding definitions. □

It is easy to prove that, if we already know that two commands are equivalent under a condition P , we are allowed to weaken the statement by strengthening that precondition:

$$\llbracket P \models c \sim c'; \forall s. P' s \longrightarrow P s \rrbracket \Longrightarrow P' \models c \sim c'$$

For the old notion of semantic equivalence we had the concept of congruence rules, where two commands remain equivalent if equivalent sub-commands are substituted for each other. The corresponding rules in the new setting are slightly more interesting. [Figure 19](#) gives an overview. The first rule, for sequential composition, has three premises instead of two. The first two are standard, i.e. equivalence of c and c' as well as d and d' . Similar to the sets of initialised variables in the definite initialisation analysis of [Section 4.1](#), we allow the precondition to change. The first premise gets the same P as the conclusion $P \models c;; d \sim c';; d'$, but the second premise can use a new Q . The third premise describes the relationship

$$\begin{array}{c}
\frac{P \models c \sim c' \quad Q \models d \sim d' \quad \forall s s'. (c, s) \Rightarrow s' \longrightarrow P s \longrightarrow Q s'}{P \models c;; d \sim c';; d'} \\
\\
\frac{P \models b <\sim> b' \quad P \models c \sim c' \quad P \models d \sim d'}{P \models IF b THEN c ELSE d \sim IF b' THEN c' ELSE d'} \\
\\
\frac{P \models c \sim c' \quad \forall s s'. (c, s) \Rightarrow s' \longrightarrow P s \longrightarrow bval b s \longrightarrow P s' \quad P \models b <\sim> b'}{P \models WHILE b DO c \sim WHILE b' DO c'}
\end{array}$$

Figure 19. Congruence rules for conditional semantic equivalence.

between P and Q : Q must hold in the states after execution of c , provided P held in the initial state.

The rule for *IF* is simpler; it just demands that the constituent expressions and commands are equivalent under the same condition P . As for the semicolon case, we could provide a stronger rule here, that takes into account which branch of the *IF* we are looking at, i.e. adding b or $\neg b$ to the condition P . Since we do not analyse the content of boolean expressions, we will not need the added power and prefer the weaker, but simpler rule.

The *WHILE* rule is similar to the semicolon case, but again in a weaker formulation. We demand that b and b' are equivalent under P , as well as c and c' . We additionally need to make sure that P still holds after the execution of the body if it held before, because the loop might enter another iteration. In other words, we need to prove as a side condition that P is an *invariant* of the loop. Since we only need to know this in the iteration case, we can additionally assume that the boolean condition b evaluates to true.

This concludes our brief interlude into conditional semantic equivalence. As indicated in [Section 2.4.4](#), we leave the proof of the rules in [Figure 19](#) as an exercise, as well as the formulation of the strengthened rules that take boolean expressions further into account.

4.2.4. Correctness

So far we have defined constant folding and propagation, and we have developed a tool set for reasoning about conditional equivalence of commands. In this section, we apply this tool set to show correctness of our optimisation.

As mentioned before, the eventual aim for our correctness statement is unconditional equivalence between the original and the optimised command:

$$fold\ c\ empty \sim c$$

To prove this statement by induction, we generalise it by replacing the empty table with an arbitrary table t . The price we pay is that the equivalence is now only true under the condition that the table correctly approximates the state the commands are run from. The statement becomes

$$approx\ t \models c \sim fold\ c\ t$$

Note that the term $approx\ t$ is partially applied. It is a function that takes a state s and returns $True$ iff t is an approximation of s as defined previously in Section 4.2.1. Expanding the definition of equivalence we get the more verbose but perhaps easier to understand form.

$$\forall s\ s'.\ approx\ t\ s \longrightarrow (c, s) \Rightarrow s' = (fold\ c\ t, s) \Rightarrow s'$$

For the proof it is nicer not to unfold the definition equivalence and work with the congruence lemmas of the previous section instead. Now, proceeding to prove this property by induction on c it quickly turns out that we will need four key lemmas about the auxiliary functions mentioned in *fold*.

The most direct and intuitive one of these is that our *defs* correctly approximates real execution. Recall that *defs* statically analyses which constant values can be assigned to which variables.

Lemma 49 (*defs* approximates execution correctly).
 $\llbracket (c, s) \Rightarrow s'; approx\ t\ s \rrbracket \Longrightarrow approx\ (defs\ c\ t)\ s'$

Proof. The proof is by rule induction on the big-step execution:

- The *SKIP* base case is trivial.
- The assignment case needs some massaging to succeed. After unfolding of definitions, case distinction on the arithmetic expression and simplification we end up with

$$\forall n.\ afold\ a\ t = N\ n \longrightarrow aval\ a\ s = n$$

where we also know our general assumption $approx\ t\ s$. This is a reformulated instance of Lemma 46.

- Sequential composition is simply an application of the two induction hypotheses.
- The two *IF* cases reduce to this property of *merge* which embodies that it is an intersection:

$$approx\ t_1\ s \vee approx\ t_2\ s \Longrightarrow approx\ (merge\ t_1\ t_2)\ s$$

In each of the two *IF* cases we know from the induction hypothesis that the execution of the chosen branch is approximated correctly by *defs*, e.g. $approx\ (defs\ c_1\ t)\ s'$. With the above *merge* lemma, we can conclude the case.

- In the *False* case for *WHILE* we observe that we are restricting the existing table t , and that approximation is trivially preserved when dropping elements.
- In the *True* case we appeal to another lemma about *defs*. From applying induction hypotheses, we know $approx\ (defs\ c\ t|_{(-\ lvars\ c)})\ s'$, but our proof goal for *defs* applied to the while loop is $approx\ (t|_{(-\ lvars\ c)})\ s'$. Lemma 50 shows that these are equal.

□

The last case of our proof above rests on one lemma we have not shown yet. It says that our restriction to variables that do not occur on the left-hand sides

of assignments is broad enough, i.e. that it appropriately masks any new table entries we would get by running *defs* on the loop body.

Lemma 50. $defs\ c\ t \upharpoonright_{(-\ lvars\ c)} = t \upharpoonright_{(-\ lvars\ c)}$

Proof. This proof is by induction on c . Most cases are automatic, merely for sequential composition and *IF* Isabelle needs a bit of hand holding for applying the induction hypotheses at the right position in the term. In the *IF* case, we also make use of this property of merge:

$$\llbracket t_1 \upharpoonright_S = t \upharpoonright_S; t_2 \upharpoonright_S = t \upharpoonright_S \rrbracket \implies merge\ t_1\ t_2 \upharpoonright_S = t \upharpoonright_S$$

It allows us to merge the two equations we get for the two branches of the *IF* into one. \square

The final lemma we need before we can proceed to the main induction is again a property about the restriction of t to the complement of *lvars*. It is the remaining fact we need for the *WHILE* case of that induction and it says that runtime execution can at most change the values of variables that are mentioned on the left-hand side of assignments.

Lemma 51.

$$\llbracket (c, s) \Rightarrow s'; approx\ (t \upharpoonright_{(-\ lvars\ c)})\ s \rrbracket \implies approx\ (t \upharpoonright_{(-\ lvars\ c)})\ s'$$

Proof. This proof is by rule induction on the big-step execution. Its cases are very similar to [Lemma 50](#). \square

Putting everything together, we can now prove our main lemma.

Lemma 52 (Generalised correctness of constant folding).

$$approx\ t \models c \sim fold\ c\ t$$

Proof. As mentioned, the proof is by induction on c . *SKIP* is simple, and assignment reduces to the correctness of *afold*, i.e. [Lemma 46](#). Sequential composition uses the congruence rule for semicolon and [Lemma 49](#). The *IF* case is automatic given the *IF* congruence rule. The *WHILE* case reduces to [Lemma 51](#), the *WHILE* congruence rule, and strengthening of the equivalence condition. The strengthening uses the following property

$$\llbracket approx\ t_2\ s; t_1 \subseteq_m t_2 \rrbracket \implies approx\ t_1\ s$$

where $(m_1 \subseteq_m m_2) = (m_1 = m_2\ on\ dom\ m_1)$ and $t \upharpoonright_S \subseteq_m t$. \square

This leads us to the final result.

Theorem 53 (Correctness of constant folding).

$$fold\ c\ empty \sim c$$

Proof. Follows immediately from [Lemma 52](#) after observing that $approx\ empty = (\lambda_. True)$. \square

4.3. Summary and Further Reading

This section has explored two different, widely used data-flow analyses and associated program optimisations: definite initialisation analysis, and constant propagation. They can be classified according to two criteria:

Forward/backward

A **forward** analysis propagates information from the beginning to the end of a program.

A **backward** analysis propagates information from the end to the beginning of a program.

May/must

A **may** analysis checks if the given property is true on some path.

A **must** analysis checks if the given property is true on all paths.

According to this schema, both are a forward must analysis: in definite initialisation analysis, variables must be assigned on all paths before they are used, in constant propagation, a variable must have the same constant value on all paths.

Data-flow analysis arose in the context of compiler construction and is treated in some detail in all decent books on the subject, e.g. [1], but in particular in the book by Muchnik [10]. The book by Nielson, Nielson and Hankin [12] provides a comprehensive and more theoretical account of program analysis.

Acknowledgements

We thank David Sands and Andrei Sabelfeld for feedback on earlier drafts of this material.

Tobias Nipkow was partially supported by NICTA. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

- [1] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
- [2] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
- [3] Ellis Cohen. Information transmission in computational systems. In *Proceedings of the sixth ACM symposium on Operating systems principles (SOSP’77)*, pages 133–139, West Lafayette, Indiana, USA, 1977. ACM.
- [4] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
- [5] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
- [6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, 3rd edition*. Addison-Wesley, 2005.
- [7] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

- [8] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, February 2013.
- [9] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences (JCSS)*, 17(3):348–375, 1978.
- [10] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [11] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, 2013.
- [12] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
- [13] Tobias Nipkow. Programming and proving in Isabelle/HOL. <http://isabelle.in.tum.de/dist/doc/prog-prove.pdf>, 2013.
- [14] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, pages 186–199. IEEE Computer Society, 2010.
- [15] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [16] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference (PSI)*, volume 5947 of *Lect. Notes in Comp. Sci.*, pages 352–365. Springer-Verlag, 2009.
- [17] Edward Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. IEEE Symposium on Security and Privacy*, pages 317–331. IEEE Computer Society, 2010.
- [18] 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.