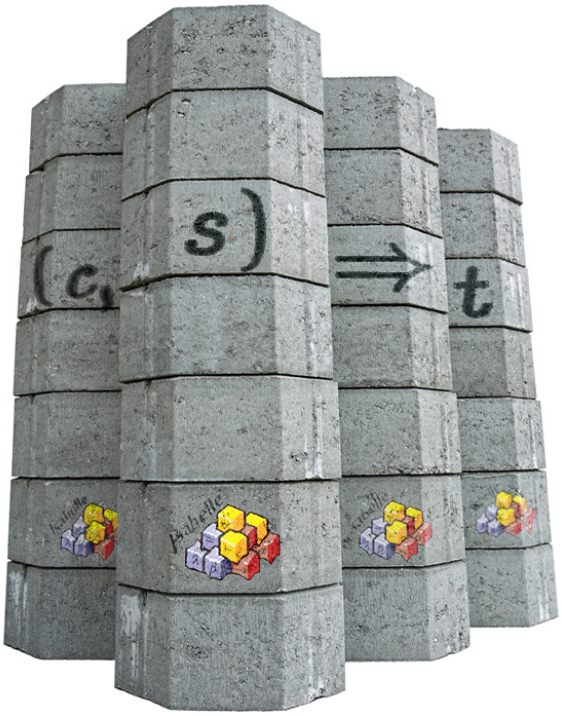
Tobias Nipkow, Gerwin Klein

Concrete Semantics

A Proof Assistant Approach

Draft: October 24, 2013 – hg id 8b28b4848d9e+



Preface

This book is two books. Part I is a practical introduction to working with the Isabelle proof assistant. It teaches you how to write functional programs and inductive definitions and how to prove properties about them in Isabelle's structured proof language. Part II is an introduction to the semantics of imperative languages with an emphasis on applications like compilers and program analysers. The distinguishing feature is that every bit of mathematics has been formalised in Isabelle and that much of it is executable. Part I focusses on the details of proofs in Isabelle. Part II can be read even without familiarity with Isabelle's proof language: all proofs are described in detail but informally. The Isabelle formalisation, including the proofs, is online: all the material, including accompanying slides, can be downloaded.

Although the subject matter is semantics and applications, the not-so-hidden agenda is to teach the reader two things: the art of precise logical reasoning and the practical use of a proof assistant as a surgical tool for formal proofs about computer science artefacts. In this sense the book represents a formal approach to computer science, not just semantics.

Why?

This book is the marriage of two areas: programming languages and theorem proving. Most programmers feel that they understand the programming language they use and the programs they write. Programming language semantics replaces a warm feeling with precision in the form of mathematical definitions of the meaning of programs. Unfortunately such definitions are still at the level of informal mathematics. They are mental tools, but their informal nature, their size, and the amount of detail makes them error prone. Since they are typically written in LaTeX, you do not even know whether they would type-check, let alone whether proofs about the semantics, e.g., compiler correctness, are free of bugs, e.g., missing cases.

This is where theorem proving systems (or "proof asistants") come in, and mathematical (im)precision is replaced by logical certainty. A proof assistant is a software system that supports the construction of mathematical theories as formal language texts that are checked for correctness. The beauty is that this includes checking the logical correctness of all proof text. No more 'proofs' that look more like LSD trips than coherent chains of logical arguments. Machine-checked (or formal) proofs offer the degree of certainty required for reliable software but impossible to achieve with informal methods.

In research, the marriage of programming languages and proof assistants has led to remarkable success stories like a verified C compiler [55] and a verified operating system kernel [48]. This book introduces students and professionals to the foundations and applications of this marriage.

Concrete?

- The book shows that a semantics is not a collection of abstract symbols on sheets of paper but formal text that can be *checked and executed* by the computer: Isabelle is also a programming environment and most of the definitions in the book are executable and can even be exported as programs in a number of (functional) programming languages. For a computer scientist, this is as concrete as it gets.
- Much of the book deals with concrete applications of semantics: compilers, type systems, program analysers.
- The predominant semantics in the book is operational semantics, the most concrete of the various forms of semantics.

Exercises!

The idea for this book goes back almost 20 years [66]. Only recently have proof assistants reached the maturity that we can inflict them on students without causing the students too much pain. Nevertheless proof assistants still require very detailed proofs. Learning this proof style (and all the syntactic details that come with any formal language) requires practice. Therefore the book contains a large number of exercises of varying difficulty. If you want to learn Isabelle, you have to work through (some of) the exercises.

A word of warning before you proceed: theorem proving can be addictive!

Contents

Part II Semantics

Part I

Isabelle

1

Introduction

Isabelle is a generic system for implementing logical formalisms, and Isabelle/HOL is the specialization of Isabelle for HOL, which abbreviates Higher-Order Logic. We introduce HOL step by step following the equation

$$\text{HOL} = \text{Functional Programming} + \text{Logic}.$$

We assume that the reader is used to logical and set theoretic notation and is familiar with the basic concepts of functional programming. Open-minded readers have been known to pick up functional programming through the wealth of examples in Chapter 2 and Chapter 3.

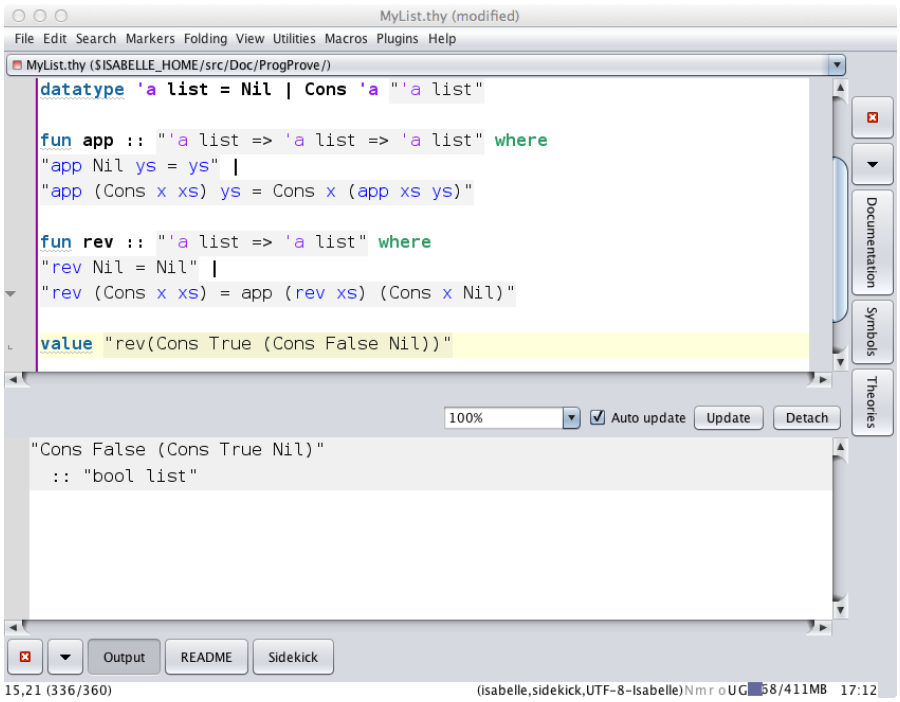
Chapter 2 introduces HOL as a functional programming language and explains how to write simple inductive proofs of mostly equational properties of recursive functions. Chapter 3 contains a little case study: arithmetic and boolean expressions, their evaluation, optimization and compilation. Chapter 4 introduces the rest of HOL: the language of formulas beyond equality, automatic proof tools, single step proofs, and inductive definitions, an essential specification construct. Chapter 5 introduces Isar, Isabelle's language for writing structured proofs.

This introduction to the core of Isabelle is intentionally concrete and example-based: we concentrate on examples that illustrate the typical cases; we do not explain the general case if it can be inferred from the examples. We cover the essentials (from a functional programming point of view) as quickly and compactly as possible. After all, this book is primarily about semantics.

For a comprehensive treatment of all things Isabelle we recommend the *Isabelle/Isar Reference Manual* [93], which comes with the Isabelle distribution. The tutorial by Nipkow, Paulson and Wenzel [69] (in its updated version that comes with the Isabelle distribution) is still recommended for the wealth of examples and material, but its proof style is outdated. In particular it fails to cover the structured proof language Isar.

Getting Started with Isabelle

If you have not done so already, download and install Isabelle from http://isabelle.in.tum.de. You can start it by clicking on the application icon. This will launch Isabelle's user interface based on the text editor **jedit**. Below you see a typical example snapshot of a jedit session. At this point we merely explain the layout of the window, not its contents.



The upper part of the window shows the input typed by the user, i.e. the gradually growing Isabelle text of definitions, theorems, proofs, etc. The interface processes the user input automatically while it is typed, just like modern Java IDEs. Isabelle's response to the user input is shown in the lower part of the window. You can examine the response to any input phrase by clicking on that phrase or by hovering over underlined text.

This should suffice to get started with the jedit interface. Now you need to learn what to type into it.

2

Programming and Proving

This chapter introduces HOL as a functional programming language and shows how to prove properties of functional programs by induction.

2.1 Basics

2.1.1 Types, Terms and Formulas

HOL is a typed logic whose type system resembles that of functional programming languages. Thus there are

base types, in particular *bool*, the type of truth values, *nat*, the type of natural numbers ($\mathbb{N}$), and *int*, the type of mathematical integers ($\mathbb{Z}$).

type constructors, in particular *list*, the type of lists, and *set*, the type of sets. Type constructors are written postfix, e.g. *nat list* is the type of lists whose elements are natural numbers.

function types, denoted by $\Rightarrow$.

type variables, denoted by $'a$, $'b$ etc., just like in ML.

Note that $'a \Rightarrow 'b\ list$ means $"'a \Rightarrow ('b\ list)"$, not $('a \Rightarrow 'b)\ list$: postfix type constructors have precedence over $\Rightarrow$.

Terms are formed as in functional programming by applying functions to arguments. If f is a function of type $\tau_1 \Rightarrow \tau_2$ and t is a term of type τ_1 then $f\ t$ is a term of type τ_2. We write $t :: \tau$ to mean that term t has type τ.

There are many predefined infix symbols like $+$ and $\leqslant$. The name of the corresponding binary function is $op\ +$, not just $+$. That is, $x + y$ is syntactic sugar for $op + x\ y$.

HOL also supports some basic constructs from functional programming:

$(\mathit{if}\ b\ \mathit{then}\ t_1\ \mathit{else}\ t_2)$
$(\mathit{let}\ x = t\ \mathit{in}\ u)$
$(\mathit{case}\ t\ \mathit{of}\ \mathit{pat}_1 \Rightarrow t_1 \mid \ldots \mid \mathit{pat}_n \Rightarrow t_n)$

The above three constructs must always be enclosed in parentheses if they occur inside other constructs.

Terms may also contain λ-abstractions. For example, $\lambda x.\ x$ is the identity function.

Formulas are terms of type *bool*. There are the basic constants *True* and *False* and the usual logical connectives (in decreasing order of precedence): $\neg$, $\wedge$, $\vee$, $\longrightarrow$.

Equality is available in the form of the infix function $=$ of type $'a \Rightarrow 'a \Rightarrow \mathit{bool}$. It also works for formulas, where it means "if and only if".

Quantifiers are written $\forall x.\ P$ and $\exists x.\ P$.

Isabelle automatically computes the type of each variable in a term. This is called **type inference**. Despite type inference, it is sometimes necessary to attach explicit **type constraints** (or **type annotations**) to a variable or term. The syntax is $t :: \tau$ as in $m < (n::\mathit{nat})$. Type constraints may be needed to disambiguate terms involving overloaded functions such as $+$, $*$ and $\leqslant$.

Finally there are the universal quantifier $\bigwedge$ and the implication $\Longrightarrow$. They are part of the Isabelle framework, not the logic HOL. Logically, they agree with their HOL counterparts $\forall$ and $\longrightarrow$, but operationally they behave differently. This will become clearer as we go along.

Right-arrows of all kinds always associate to the right. In particular, the formula $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ means $A_1 \Longrightarrow (A_2 \Longrightarrow A_3)$. The (Isabelle specific) notation $[\![\ A_1; \ldots; A_n\ ]\!] \Longrightarrow A$ is short for the iterated implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$. Sometimes we also employ inference rule notation: $\dfrac{A_1 \quad \ldots \quad A_n}{A}$

2.1.2 Theories

Roughly speaking, a **theory** is a named collection of types, functions, and theorems, much like a module in a programming language. All the Isabelle text that you ever type needs to go into a theory. The general format of a theory T is

theory T
imports $T_1\ \ldots\ T_n$
begin
definitions, theorems and proofs
end

where T_1 ... T_n are the names of existing theories that T is based on. The T_i are the direct **parent theories** of T. Everything defined in the parent theories (and their parents, recursively) is automatically visible. Each theory T must reside in a **theory file** named T*.thy*.


HOL contains a theory *Main*, the union of all the basic predefined theories like arithmetic, lists, sets, etc. Unless you know what you are doing, always include *Main* as a direct or indirect parent of all your theories.

In addition to the theories that come with the Isabelle/HOL distribution (see http://isabelle.in.tum.de/library/HOL/) there is also the *Archive of Formal Proofs* at http://afp.sourceforge.net, a growing collection of Isabelle theories that everybody can contribute to.

2.1.3 Quotation Marks

The textual definition of a theory follows a fixed syntax with keywords like **begin** and **datatype**. Embedded in this syntax are the types and formulas of HOL. To distinguish the two levels, everything HOL-specific (terms and types) must be enclosed in quotation marks: "...". To lessen this burden, quotation marks around a single identifier can be dropped. When Isabelle prints a syntax error message, it refers to the HOL syntax as the **inner syntax** and the enclosing theory language as the **outer syntax**.

2.2 Types *bool*, *nat* and *list*

These are the most important predefined types. We go through them one by one. Based on examples we learn how to define (possibly recursive) functions and prove theorems about them by induction and simplification.

2.2.1 Type *bool*

The type of boolean values is a predefined datatype

datatype *bool* = *True* | *False*

with the two values *True* and *False* and with many predefined functions: $\neg$, $\wedge$, $\vee$, $\longrightarrow$ etc. Here is how conjunction could be defined by pattern matching:

fun *conj* :: *"bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool"* **where**
"conj True True = *True"* |
"conj _ _ = *False"*

Both the datatype and function definitions roughly follow the syntax of functional programming languages.

2.2.2 Type *nat*

Natural numbers are another predefined datatype:

datatype $nat = 0 \mid Suc\ nat$

All values of type *nat* are generated by the constructors 0 and *Suc*. Thus the values of type *nat* are 0, *Suc* 0, *Suc* (*Suc* 0) etc. There are many predefined functions: $+$, $*$, $\leqslant$, etc. Here is how you could define your own addition:

fun $add :: "nat \Rightarrow nat \Rightarrow nat"$ **where**
$"add\ 0\ n = n" \mid$
$"add\ (Suc\ m)\ n = Suc(add\ m\ n)"$

And here is a proof of the fact that $add\ m\ 0 = m$:

lemma add_02: $"add\ m\ 0 = m"$
apply$(induction\ m)$
apply$(auto)$
done

The **lemma** command starts the proof and gives the lemma a name, add_02. Properties of recursively defined functions need to be established by induction in most cases. Command **apply**$(induction\ m)$ instructs Isabelle to start a proof by induction on m. In response, it will show the following proof state:

1. $add\ 0\ 0 = 0$
2. $\bigwedge m.\ add\ m\ 0 = m \Longrightarrow add\ (Suc\ m)\ 0 = Suc\ m$

The numbered lines are known as *subgoals*. The first subgoal is the base case, the second one the induction step. The prefix $\bigwedge m.$ is Isabelle's way of saying "for an arbitrary but fixed m". The $\Longrightarrow$ separates assumptions from the conclusion. The command **apply**$(auto)$ instructs Isabelle to try and prove all subgoals automatically, essentially by simplifying them. Because both subgoals are easy, Isabelle can do it. The base case $add\ 0\ 0 = 0$ holds by definition of *add*, and the induction step is almost as simple: $add\ (Suc\ m)\ 0 = Suc(add\ m\ 0) = Suc\ m$ using first the definition of *add* and then the induction hypothesis. In summary, both subproofs rely on simplification with function definitions and the induction hypothesis. As a result of that final **done**, Isabelle associates the lemma just proved with its name. You can now inspect the lemma with the command

thm add_02

which displays

$add\ ?m\ 0 = ?m$

The free variable m has been replaced by the **unknown** $?m$. There is no logical difference between the two but an operational one: unknowns can be instantiated, which is what you want after some lemma has been proved.

Note that there is also a proof method *induct*, which behaves almost like *induction*; the difference is explained in Chapter 5.

Terminology: We use **lemma**, **theorem** and **rule** interchangeably for propositions that have been proved.

Numerals (0, 1, 2, ...) and most of the standard arithmetic operations ($+$, $-$, $*$, $\leqslant$, $<$ etc) are overloaded: they are available not just for natural numbers but for other types as well. For example, given the goal $x + 0 = x$, there is nothing to indicate that you are talking about natural numbers. Hence Isabelle can only infer that x is of some arbitrary type where 0 and $+$ exist. As a consequence, you will be unable to prove the goal. To alert you to such pitfalls, Isabelle flags numerals without a fixed type in its output: $x + (0::'a) = x$. In this particular example, you need to include an explicit type constraint, for example $x+0 = (x::nat)$. If there is enough contextual information this may not be necessary: $Suc\ x = x$ automatically implies $x::nat$ because Suc is not overloaded.

An Informal Proof

Above we gave some terse informal explanation of the proof of $add\ m\ 0 = m$. A more detailed informal exposition of the lemma might look like this:

Lemma $add\ m\ 0 = m$
Proof by induction on m.

- Case 0 (the base case): $add\ 0\ 0 = 0$ holds by definition of add.
- Case $Suc\ m$ (the induction step): We assume $add\ m\ 0 = m$, the induction hypothesis (IH), and we need to show $add\ (Suc\ m)\ 0 = Suc\ m$. The proof is as follows:

  $add\ (Suc\ m)\ 0 = Suc\ (add\ m\ 0)$ by definition of add
  $= Suc\ m$ by IH

Throughout this book, **IH** will stand for "induction hypothesis".

We have now seen three proofs of $add\ m\ 0 = 0$: the Isabelle one, the terse four lines explaining the base case and the induction step, and just now a model of a traditional inductive proof. The three proofs differ in the level of detail given and the intended reader: the Isabelle proof is for the machine, the informal proofs are for humans. Although this book concentrates on Isabelle proofs, it is important to be able to rephrase those proofs as informal text comprehensible to a reader familiar with traditional mathematical proofs. Later on we will introduce an Isabelle proof language that is closer to traditional informal mathematical language and is often directly readable.

2.2.3 Type *list*

Although lists are already predefined, we define our own copy just for demonstration purposes:

datatype *'a list = Nil | Cons 'a "'a list"*

- Type *'a list* is the type of lists over elements of type *'a*. Because *'a* is a type variable, lists are in fact **polymorphic**: the elements of a list can be of arbitrary type (but must all be of the same type).
- Lists have two constructors: *Nil*, the empty list, and *Cons*, which puts an element (of type *'a*) in front of a list (of type *'a list*). Hence all lists are of the form *Nil*, or *Cons x Nil*, or *Cons x (Cons y Nil)* etc.
- **datatype** requires no quotation marks on the left-hand side, but on the right-hand side each of the argument types of a constructor needs to be enclosed in quotation marks, unless it is just an identifier (e.g. *nat* or *'a*).

We also define two standard functions, append and reverse:

fun *app :: "'a list ⇒ 'a list ⇒ 'a list"* **where**
"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"

fun *rev :: "'a list ⇒ 'a list"* **where**
"rev Nil = Nil" |
"rev (Cons x xs) = app (rev xs) (Cons x Nil)"

By default, variables *xs*, *ys* and *zs* are of *list* type.

Command **value** evaluates a term. For example,

value *"rev(Cons True (Cons False Nil))"*

yields the result *Cons False (Cons True Nil)*. This works symbolically, too:

value *"rev(Cons a (Cons b Nil))"*

yields *Cons b (Cons a Nil)*.

Figure 2.1 shows the theory created so far. Because *list*, *Nil*, *Cons* etc are already predefined, Isabelle prints qualified (long) names when executing this theory, for example, *MyList.Nil* instead of *Nil*. To suppress the qualified names you can insert the command `declare [[names_short]]`. This is not recommended in general but just for this unusual example.

Structural Induction for Lists

Just as for natural numbers, there is a proof principle of induction for lists. Induction over a list is essentially induction over the length of the list, al-

```
theory MyList
imports Main
begin

datatype 'a list = Nil | Cons 'a "'a list"

fun app :: "'a list => 'a list => 'a list" where
"app Nil ys = ys" |
"app (Cons x xs) ys = Cons x (app xs ys)"

fun rev :: "'a list => 'a list" where
"rev Nil = Nil" |
"rev (Cons x xs) = app (rev xs) (Cons x Nil)"

value "rev(Cons True (Cons False Nil))"

end
```

Fig. 2.1. A Theory of Lists

though the length remains implicit. To prove that some property P holds for all lists xs, i.e. $P\ xs$, you need to prove

1. the base case $P\ Nil$ and
2. the inductive case $P\ (Cons\ x\ xs)$ under the assumption $P\ xs$, for some arbitrary but fixed x and xs.

This is often called **structural induction**.

2.2.4 The Proof Process

We will now demonstrate the typical proof process, which involves the formulation and proof of auxiliary lemmas. Our goal is to show that reversing a list twice produces the original list.

theorem *rev_rev* [*simp*]: *"rev(rev xs) = xs"*

Commands **theorem** and **lemma** are interchangeable and merely indicate the importance we attach to a proposition. Via the bracketed attribute *simp* we also tell Isabelle to make the eventual theorem a **simplification rule**: future proofs involving simplification will replace occurrences of $rev\ (rev\ xs)$ by xs. The proof is by induction:

apply(*induction xs*)

As explained above, we obtain two subgoals, namely the base case (Nil) and the induction step ($Cons$):

1. $rev\ (rev\ Nil) = Nil$
2. $\bigwedge a\ xs.\ rev\ (rev\ xs) = xs \Longrightarrow rev\ (rev\ (Cons\ a\ xs)) = Cons\ a\ xs$

Let us try to solve both goals automatically:

apply$(auto)$

Subgoal 1 is proved, and disappears; the simplified version of subgoal 2 becomes the new subgoal 1:

1. $\bigwedge a\ xs.$
   $rev\ (rev\ xs) = xs \Longrightarrow$
   $rev\ (app\ (rev\ xs)\ (Cons\ a\ Nil)) = Cons\ a\ xs$

In order to simplify this subgoal further, a lemma suggests itself.

A First Lemma

We insert the following lemma in front of the main theorem:

lemma $rev\_app\ [simp]$: $"rev(app\ xs\ ys) = app\ (rev\ ys)\ (rev\ xs)"$

There are two variables that we could induct on: xs and ys. Because app is defined by recursion on the first argument, xs is the correct one:

apply$(induction\ xs)$

This time not even the base case is solved automatically:

apply$(auto)$

1. $rev\ ys = app\ (rev\ ys)\ Nil$

A total of 2 subgoals...

Again, we need to abandon this proof attempt and prove another simple lemma first.

A Second Lemma

We again try the canonical proof procedure:

lemma $app\_Nil2\ [simp]$: $"app\ xs\ Nil = xs"$
apply$(induction\ xs)$
apply$(auto)$
done

Thankfully, this worked. Now we can continue with our stuck proof attempt of the first lemma:

lemma $rev\_app\ [simp]$: $"rev(app\ xs\ ys) = app\ (rev\ ys)\ (rev\ xs)"$

apply(*induction xs*)
apply(*auto*)

We find that this time *auto* solves the base case, but the induction step merely simplifies to

1. $\bigwedge a\ xs.$
   $rev\ (app\ xs\ ys) = app\ (rev\ ys)\ (rev\ xs) \Longrightarrow$
   $app\ (app\ (rev\ ys)\ (rev\ xs))\ (Cons\ a\ Nil) =$
   $app\ (rev\ ys)\ (app\ (rev\ xs)\ (Cons\ a\ Nil))$

The missing lemma is associativity of *app*, which we insert in front of the failed lemma *rev_app*.

Associativity of *app*

The canonical proof procedure succeeds without further ado:

lemma *app_assoc* [*simp*]: "*app* (*app xs ys*) *zs* = *app xs* (*app ys zs*)"
apply(*induction xs*)
apply(*auto*)
done

Finally the proofs of *rev_app* and *rev_rev* succeed, too.

Another Informal Proof

Here is the informal proof of associativity of *app* corresponding to the Isabelle proof above.

Lemma $app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs)$
Proof by induction on *xs*.

- Case *Nil*: $app\ (app\ Nil\ ys)\ zs = app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$ holds by definition of *app*.
- Case $Cons\ x\ xs$: We assume

  $$app\ (app\ xs\ ys)\ zs = app\ xs\ (app\ ys\ zs) \qquad \text{(IH)}$$

  and we need to show

  $$app\ (app\ (Cons\ x\ xs)\ ys)\ zs = app\ (Cons\ x\ xs)\ (app\ ys\ zs).$$

  The proof is as follows:

  $app\ (app\ (Cons\ x\ xs)\ ys)\ zs$
  $= app\ (Cons\ x\ (app\ xs\ ys))\ zs$ by definition of *app*
  $= Cons\ x\ (app\ (app\ xs\ ys)\ zs)$ by definition of *app*
  $= Cons\ x\ (app\ xs\ (app\ ys\ zs))$ by IH
  $= app\ (Cons\ x\ xs)\ (app\ ys\ zs)$ by definition of *app*

Didn't we say earlier that all proofs are by simplification? But in both cases, going from left to right, the last equality step is not a simplification at all! In the base case it is $app\ ys\ zs = app\ Nil\ (app\ ys\ zs)$. It appears almost mysterious because we suddenly complicate the term by appending Nil on the left. What is really going on is this: when proving some equality $s = t$, both s and t are simplified to some common term u. This heuristic for equality proofs works well for a functional programming context like ours. In the base case s is $app\ (app\ Nil\ ys)\ zs$, t is $app\ Nil\ (app\ ys\ zs)$, and u is $app\ ys\ zs$.

2.2.5 Predefined Lists

Isabelle's predefined lists are the same as the ones above, but with more syntactic sugar:

- $[]$ is Nil,
- $x \# xs$ is $Cons\ x\ xs$,
- $[x_1, \ldots, x_n]$ is $x_1 \# \ldots \# x_n \# []$, and
- xs @ ys is $app\ xs\ ys$.

There is also a large library of predefined functions. The most important ones are the length function $length :: 'a\ list \Rightarrow nat$ (with the obvious definition), and the map function that applies a function to all elements of a list:

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list"
"map f [] = []" |
"map f (x # xs) = f x # map f xs"
```

Also useful are the **head** of a list, its first element, and the **tail**, the rest of the list:

```
fun hd :: 'a list ⇒ 'a
hd (x # xs) = x

fun tl :: 'a list ⇒ 'a list
tl [] = [] |
tl (x # xs) = xs
```

Note that since HOL is a logic of total functions, $hd\ []$ is defined, but we do now know what the result is. That is, $hd\ []$ is not undefined but underdefined.

From now on lists are always the predefined lists.

2.2.6 Exercises

Exercise 2.1. Use the **value** command to evaluate the following expressions: `"1 + (2::nat)"`, `"1 + (2::int)"`, `"1 - (2::nat)"` and `"1 - (2::int)"`.

Exercise 2.2. Start from the definition of *add* given above. Prove that *add* it is associative and commutative. Define a recursive function *double* :: *nat* ⇒ *nat* and prove *double m* = *add m m*.

Exercise 2.3. Define a function *count* :: *'a* ⇒ *'a list* ⇒ *nat* that counts the number of occurrences of an element in a list. Prove *count x xs* ⩽ *length xs*.

Exercise 2.4. Define a recursive function *snoc* :: *'a list* ⇒ *'a* ⇒ *'a list* that appends an element to the end of a list. With the help of *snoc* define a recursive function *reverse* :: *'a list* ⇒ *'a list* that reverses a list. Prove *reverse* (*reverse xs*) = *xs*.

Exercise 2.5. Define a recursive function *sum* :: *nat* ⇒ *nat* such that *sum n* = 0 + ... + *n* and prove *sum n* = *n* ∗ (*n* + 1) *div* 2.

2.3 Type and Function Definitions

Type synonyms are abbreviations for existing types, for example

type_synonym *string* = "*char list*"

Type synonyms are expanded after parsing and are not present in internal representation and output. They are mere conveniences for the reader.

2.3.1 Datatypes

The general form of a datatype definition looks like this:

$$\textbf{datatype}\ ('a_1,\ldots,'a_n)t = C_1\ "\tau_{1,1}"\ldots"\tau_{1,n_1}" \mid \ldots \mid C_k\ "\tau_{k,1}"\ldots"\tau_{k,n_k}"$$

It introduces the constructors $C_i :: \tau_{i,1} \Rightarrow \cdots \Rightarrow \tau_{i,n_i} \Rightarrow ('a_1,\ldots,'a_n)t$ and expresses that any value of this type is built from these constructors in a unique manner. Uniqueness is implied by the following properties of the constructors:

- *Distinctness:* $C_i\ \ldots \neq C_j\ \ldots$ if $i \neq j$
- *Injectivity:* $(C_i\ x_1 \ldots x_{n_i} = C_i\ y_1 \ldots y_{n_i}) = (x_1 = y_1 \wedge \ldots \wedge x_{n_i} = y_{n_i})$

The fact that any value of the datatype is built from the constructors implies the structural induction rule: to show $P\ x$ for all x of type $('a_1,\ldots,'a_n)t$, one needs to show $P(C_i\ x_1 \ldots x_{n_i})$ (for each i) assuming $P(x_j)$ for all j where $\tau_{i,j} = ('a_1,\ldots,'a_n)t$. Distinctness and injectivity are applied automatically by *auto* and other proof methods. Induction must be applied explicitly.

Datatype values can be taken apart with case-expressions, for example

$(case\ xs\ of\ [] \Rightarrow 0 \mid x\ \#\ \_ \Rightarrow Suc\ x)$

just like in functional programming languages. Case expressions must be enclosed in parentheses.

As an example, consider binary trees:

datatype *'a tree* = *Tip* | *Node* *"'a tree"* *'a* *"'a tree"*

with a mirror function:

fun *mirror* :: *"'a tree* ⇒ *'a tree"* **where**
"mirror Tip = *Tip"* |
"mirror (*Node l a r*) = *Node* (*mirror r*) *a* (*mirror l*)*"*

The following lemma illustrates induction:

lemma *"mirror*(*mirror t*) = *t"*
apply(*induction t*)

yields

1. *mirror* (*mirror Tip*) = *Tip*
2. ⋀*t1 a t2*.
   ⟦*mirror* (*mirror t1*) = *t1*; *mirror* (*mirror t2*) = *t2*⟧
   ⟹ *mirror* (*mirror* (*Node t1 a t2*)) = *Node t1 a t2*

The induction step contains two induction hypotheses, one for each subtree. An application of *auto* finishes the proof.

A very simple but also very useful datatype is the predefined

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

Its sole purpose is to add a new element *None* to an existing type *'a*. To make sure that *None* is distinct from all the elements of *'a*, you wrap them up in *Some* and call the new type *'a option*. A typical application is a lookup function on a list of key-value pairs, often called an association list:

fun *lookup* :: *"('a* ∗ *'b) list* ⇒ *'a* ⇒ *'b option"* **where**
"lookup [] *x* = *None"* |
"lookup ((*a*,*b*) # *ps*) *x* = (*if a* = *x then Some b else lookup ps x*)*"*

Note that $\tau_1 * \tau_2$ is the type of pairs, also written $\tau_1 \times \tau_2$. Pairs can be taken apart either by pattern matching (as above) or with the projection functions *fst* and *snd*: *fst* (*a*, *b*) = *a* and *snd* (*a*, *b*) = *b*. Tuples are simulated by pairs nested to the right: (*a*, *b*, *c*) abbreviates (*a*, (*b*, *c*)) and $\tau_1 \times \tau_2 \times \tau_3$ abbreviates $\tau_1 \times (\tau_2 \times \tau_3)$.

2.3.2 Definitions

Non recursive functions can be defined as in the following example:

definition $sq :: "nat \Rightarrow nat"$ **where**
$"sq\ n = n * n"$

Such definitions do not allow pattern matching but only $f\ x_1\ \ldots\ x_n = t$, where f does not occur in t.

2.3.3 Abbreviations

Abbreviations are similar to definitions:

abbreviation $sq' :: "nat \Rightarrow nat"$ **where**
$"sq'\ n \equiv n * n"$

The key difference is that sq' is only syntactic sugar: after parsing, $sq'\ t$ is replaced by $t * t$, and before printing, every occurrence of $u * u$ is replaced by $sq'\ u$. Internally, sq' does not exist. This is the advantage of abbreviations over definitions: definitions need to be expanded explicitly (Section 2.5.5) whereas abbreviations are already expanded upon parsing. However, abbreviations should be introduced sparingly: if abused, they can lead to a confusing discrepancy between the internal and external view of a term.

The ASCII representation of $\equiv$ is == or `\<equiv>`.

2.3.4 Recursive Functions

Recursive functions are defined with **fun** by pattern matching over datatype constructors. The order of equations matters. Just as in functional programming languages. However, all HOL functions must be total. This simplifies the logic—terms are always defined—but means that recursive functions must terminate. Otherwise one could define a function $f\ n = f\ n + 1$ and conclude $0 = 1$ by subtracting $f\ n$ on both sides.

Isabelle's automatic termination checker requires that the arguments of recursive calls on the right-hand side must be strictly smaller than the arguments on the left-hand side. In the simplest case, this means that one fixed argument position decreases in size with each recursive call. The size is measured as the number of constructors (excluding 0-ary ones, e.g. Nil). Lexicographic combinations are also recognized. In more complicated situations, the user may have to prove termination by hand. For details see [51].

Functions defined with **fun** come with their own induction schema that mirrors the recursion schema and is derived from the termination order. For example,

fun *div2* :: *"nat ⇒ nat"* **where**
"div2 0 = 0" |
"div2 (Suc 0) = 0" |
"div2 (Suc(Suc n)) = Suc(div2 n)"

does not just define *div2* but also proves a customized induction rule:

$$\frac{P\ 0 \qquad P\ (Suc\ 0) \qquad \bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ (Suc\ n))}{P\ m}$$

This customized induction rule can simplify inductive proofs. For example,

lemma *"div2(n+n) = n"*
apply(*induction n rule*: *div2.induct*)

yields the 3 subgoals

1. *div2 (0 + 0) = 0*
2. *div2 (Suc 0 + Suc 0) = Suc 0*
3. $\bigwedge n.$ *div2 (n + n) = n ⟹*
   div2 (Suc (Suc n) + Suc (Suc n)) = Suc (Suc n)

An application of *auto* finishes the proof. Had we used ordinary structural induction on n, the proof would have needed an additional case analysis in the induction step.

The general case is often called **computation induction**, because the induction follows the (terminating!) computation. For every defining equation

$$f(e) = \ldots f(r_1) \ldots f(r_k) \ldots$$

where $f(r_i)$, i=1...k, are all the recursive calls, the induction rule $f.induct$ contains one premise of the form

$$P(r_1) \Longrightarrow \ldots \Longrightarrow P(r_k) \Longrightarrow P(e)$$

If $f :: \tau_1 \Rightarrow \ldots \Rightarrow \tau_n \Rightarrow \tau$ then $f.induct$ is applied like this:

apply(*induction* $x_1 \ldots x_n$ *rule*: $f.induct$)

where typically there is a call $f\ x_1 \ldots x_n$ in the goal. But note that the induction rule does not mention f at all, except in its name, and is applicable independently of f.

2.3.5 Exercises

Exercise 2.6. Starting from the type *'a tree* defined in the text, define a function *contents* :: *'a tree ⇒ 'a list* that collects all values in a tree in a list, in any order, without removing duplicates. Then define a function *treesum* :: *nat tree ⇒ nat* that sums up all values in a tree of natural numbers and prove *treesum t = listsum (contents t)*.

Exercise 2.7. Define a new type *'a tree2* of binary trees where values are also stored in the leaves of the tree. Also reformulate the *mirror* function accordingly. Define two functions *pre_order* and *post_order* of type *'a tree2* ⇒ *'a list* that traverse a tree and collect all stored values in the respective order in a list. Prove *pre_order* (*mirror t*) = *rev* (*post_order t*).

Exercise 2.8. Prove that *div2* defined above divides every number by 2, not just those of the form *n*+*n*: *div2 n* = *n div* 2.

2.4 Induction Heuristics

We have already noted that theorems about recursive functions are proved by induction. In case the function has more than one argument, we have followed the following heuristic in the proofs about the append function:

Perform induction on argument number i
if the function is defined by recursion on argument number i.

The key heuristic, and the main point of this section, is to *generalize the goal before induction*. The reason is simple: if the goal is too specific, the induction hypothesis is too weak to allow the induction step to go through. Let us illustrate the idea with an example.

Function *rev* has quadratic worst-case running time because it calls append for each element of the list and append is linear in its first argument. A linear time version of *rev* requires an extra argument where the result is accumulated gradually, using only #:

fun *itrev* :: *"'a list* ⇒ *'a list* ⇒ *'a list"* **where**
"itrev [] *ys* = *ys"* |
"itrev (*x*#*xs*) *ys* = *itrev xs* (*x*#*ys*)*"*

The behaviour of *itrev* is simple: it reverses its first argument by stacking its elements onto the second argument, and it returns that second argument when the first one becomes empty. Note that *itrev* is tail-recursive: it can be compiled into a loop, no stack is necessary for executing it.

Naturally, we would like to show that *itrev* does indeed reverse its first argument provided the second one is empty:

lemma *"itrev xs* [] = *rev xs"*

There is no choice as to the induction variable:

apply(*induction xs*)
apply(*auto*)

Unfortunately, this attempt does not prove the induction step:

1. $\bigwedge a\ xs.\ itrev\ xs\ []\ =\ rev\ xs \Longrightarrow itrev\ xs\ [a]\ =\ rev\ xs\ @\ [a]$

The induction hypothesis is too weak. The fixed argument, $[]$, prevents it from rewriting the conclusion. This example suggests a heuristic:

Generalize goals for induction by replacing constants by variables.

Of course one cannot do this naïvely: $itrev\ xs\ ys\ =\ rev\ xs$ is just not true. The correct generalization is

lemma *"itrev xs ys = rev xs @ ys"*

If ys is replaced by $[]$, the right-hand side simplifies to $rev\ xs$, as required. In this instance it was easy to guess the right generalization. Other situations can require a good deal of creativity.

Although we now have two variables, only xs is suitable for induction, and we repeat our proof attempt. Unfortunately, we are still not there:

1. $\bigwedge a\ xs.$
   $itrev\ xs\ ys\ =\ rev\ xs\ @\ ys \Longrightarrow$
   $itrev\ xs\ (a\ \#\ ys)\ =\ rev\ xs\ @\ a\ \#\ ys$

The induction hypothesis is still too weak, but this time it takes no intuition to generalize: the problem is that the ys in the induction hypothesis is fixed, but the induction hypothesis needs to be applied with $a\ \#\ ys$ instead of ys. Hence we prove the theorem for all ys instead of a fixed one. We can instruct induction to perform this generalization for us by adding *arbitrary*: *ys*.

apply(*induction xs arbitrary*: *ys*)

The induction hypothesis in the induction step is now universally quantified over ys:

1. $\bigwedge ys.\ itrev\ []\ ys\ =\ rev\ []\ @\ ys$
2. $\bigwedge a\ xs\ ys.$
   $(\bigwedge ys.\ itrev\ xs\ ys\ =\ rev\ xs\ @\ ys) \Longrightarrow$
   $itrev\ (a\ \#\ xs)\ ys\ =\ rev\ (a\ \#\ xs)\ @\ ys$

Thus the proof succeeds:

apply *auto*
done

This leads to another heuristic for generalization:

Generalize induction by generalizing all free variables
(except the induction variable itself).

Generalization is best performed with *arbitrary*: $y_1\ \ldots\ y_k$. This heuristic prevents trivial failures like the one above. However, it should not be applied

blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that need to be quantified are typically those that change in recursive calls.

2.4.1 Exercises

Exercise 2.9. Write a tail-recursive variant of the *add* function on *nat*: *itadd*. Tail-recursive means that in the recursive case, *itadd* needs to call itself directly: *itadd* (*Suc m*) *n* = *itadd* .... Prove *itadd m n* = *add m n*.

2.5 Simplification

So far we have talked a lot about simplifying terms without explaining the concept. **Simplification** means

- using equations l = r from left to right (only),
- as long as possible.

To emphasize the directionality, equations that have been given the *simp* attribute are called **simplification** rules. Logically, they are still symmetric, but proofs by simplification use them only in the left-to-right direction. The proof tool that performs simplifications is called the **simplifier**. It is the basis of *auto* and other related proof methods.

The idea of simplification is best explained by an example. Given the simplification rules

$$\begin{aligned} 0 + n &= n && (1)\\ Suc\ m + n &= Suc\ (m + n) && (2)\\ (Suc\ m \leqslant Suc\ n) &= (m \leqslant n) && (3)\\ (0 \leqslant m) &= True && (4) \end{aligned}$$

the formula $0 + Suc\ 0 \leqslant Suc\ 0 + x$ is simplified to *True* as follows:

$$\begin{aligned} (0 + Suc\ 0 \leqslant Suc\ 0 + x) &\overset{(1)}{=}\\ (Suc\ 0 \leqslant Suc\ 0 + x) &\overset{(2)}{=}\\ (Suc\ 0 \leqslant Suc\ (0 + x)) &\overset{(3)}{=}\\ (0 \leqslant 0 + x) &\overset{(4)}{=}\\ True \end{aligned}$$

Simplification is often also called **rewriting** and simplification rules **rewrite rules**.

2.5.1 Simplification Rules

The attribute *simp* declares theorems to be simplification rules, which the simplifier will use automatically. In addition, **datatype** and **fun** commands implicitly declare some simplification rules: **datatype** the distinctness and injectivity rules, **fun** the defining equations. Definitions are not declared as simplification rules automatically! Nearly any theorem can become a simplification rule. The simplifier will try to transform it into an equation. For example, the theorem $\neg P$ is turned into $P = \mathit{False}$.

Only equations that really simplify, like $\mathit{rev}\ (\mathit{rev}\ xs) = xs$ and $xs\ @\ [] = xs$, should be declared as simplification rules. Equations that may be counterproductive as simplification rules should only be used in specific proof steps (see §2.5.4 below). Distributivity laws, for example, alter the structure of terms and can produce an exponential blow-up.

2.5.2 Conditional Simplification Rules

Simplification rules can be conditional. Before applying such a rule, the simplifier will first try to prove the preconditions, again by simplification. For example, given the simplification rules

$$p\ 0 = \mathit{True}$$
$$p\ x \Longrightarrow f\ x = g\ x,$$

the term $f\ 0$ simplifies to $g\ 0$ but $f\ 1$ does not simplify because $p\ 1$ is not provable.

2.5.3 Termination

Simplification can run forever, for example if both $f\ x = g\ x$ and $g\ x = f\ x$ are simplification rules. It is the user's responsibility not to include simplification rules that can lead to nontermination, either on their own or in combination with other simplification rules. The right-hand side of a simplification rule should always be "simpler" than the left-hand side—in some sense. But since termination is undecidable, such a check cannot be automated completely and Isabelle makes little attempt to detect nontermination.

When conditional simplification rules are applied, their preconditions are proved first. Hence all preconditions need to be simpler than the left-hand side of the conclusion. For example

$$n < m \Longrightarrow (n < \mathit{Suc}\ m) = \mathit{True}$$

is suitable as a simplification rule: both $n < m$ and True are simpler than $n < \mathit{Suc}\ m$. But

$Suc\ n < m \Longrightarrow (n < m) = True$

leads to nontermination: when trying to rewrite $n < m$ to *True* one first has to prove $Suc\ n < m$, which can be rewritten to *True* provided $Suc\ (Suc\ n) < m$, *ad infinitum*.

2.5.4 The *simp* Proof Method

So far we have only used the proof method *auto*. Method *simp* is the key component of *auto*, but *auto* can do much more. In some cases, *auto* is overeager and modifies the proof state too much. In such cases the more predictable *simp* method should be used. Given a goal

1. $[\![P_1; \ldots; P_m]\!] \Longrightarrow C$

the command

apply(*simp add*: $th_1 \ \ldots \ th_n$)

simplifies the assumptions P_i and the conclusion C using

- all simplification rules, including the ones coming from **datatype** and **fun**,
- the additional lemmas $th_1 \ \ldots \ th_n$, and
- the assumptions.

In addition to or instead of *add* there is also *del* for removing simplification rules temporarily. Both are optional. Method *auto* can be modified similarly:

apply(*auto simp add*: ... *simp del*: ...)

Here the modifiers are *simp add* and *simp del* instead of just *add* and *del* because *auto* does not just perform simplification.

Note that *simp* acts only on subgoal 1, *auto* acts on all subgoals. There is also *simp_all*, which applies *simp* to all subgoals.

2.5.5 Rewriting With Definitions

Definitions introduced by the command **definition** can also be used as simplification rules, but by default they are not: the simplifier does not expand them automatically. Definitions are intended for introducing abstract concepts and not merely as abbreviations. Of course, we need to expand the definition initially, but once we have proved enough abstract properties of the new constant, we can forget its original definition. This style makes proofs more robust: if the definition has to be changed, only the proofs of the abstract properties will be affected.

The definition of a function *f* is a theorem named *f_def* and can be added to a call of *simp* just like any other theorem:

apply(*simp add*: *f_def*)

In particular, let-expressions can be unfolded by making *Let_def* a simplification rule.

2.5.6 Case Splitting With *simp*

Goals containing if-expressions are automatically split into two cases by *simp* using the rule

$$P\ (\mathit{if}\ A\ \mathit{then}\ s\ \mathit{else}\ t) = ((A \longrightarrow P\ s) \wedge (\neg\ A \longrightarrow P\ t))$$

For example, *simp* can prove

$$(A \wedge B) = (\mathit{if}\ A\ \mathit{then}\ B\ \mathit{else}\ \mathit{False})$$

because both $A \longrightarrow (A \wedge B) = B$ and $\neg\ A \longrightarrow (A \wedge B) = \mathit{False}$ simplify to *True*.

We can split case-expressions similarly. For *nat* the rule looks like this:

$$P\ (\mathit{case}\ e\ \mathit{of}\ 0 \Rightarrow a \mid \mathit{Suc}\ n \Rightarrow b\ n) =$$
$$((e = 0 \longrightarrow P\ a) \wedge (\forall n.\ e = \mathit{Suc}\ n \longrightarrow P\ (b\ n)))$$

Case expressions are not split automatically by *simp*, but *simp* can be instructed to do so:

apply(*simp split*: *nat.split*)

splits all case-expressions over natural numbers. For an arbitrary datatype *t* it is *t.split* instead of *nat.split*. Method *auto* can be modified in exactly the same way.

2.5.7 Exercises

Exercise 2.10. Define a datatype *tree0* of binary tree skeletons which do not store any information, neither in the inner nodes nor in the leaves. Define a function *nodes* :: *tree0* $\Rightarrow$ *nat* that counts the total number all nodes (inner nodes and leaves) in such a tree. Consider the following recursive function:

fun *explode* :: *"nat* $\Rightarrow$ *tree0* $\Rightarrow$ *tree0"* **where**
"explode 0 t = t" |
"explode (Suc n) t = explode n (Node t t)"

Find an equation expressing the size of a tree after exploding it (*nodes* (*explode n t*)) as a function of *nodes t* and *n*. Prove your equation. You may use the usual arithmetic operators including the exponentiation operator " ^". For example, $2\ \hat{}\ 2 = 4$.

Hint: simplifying with the list of theorems *algebra_simps* takes care of common algebraic properties of the arithmetic operators.

Exercise 2.11. Define arithmetic expressions in one variable over integers (type *int*) as a data type:

datatype $exp = Var \mid Const\ int \mid Add\ exp\ exp \mid Mult\ exp\ exp$

Define a function $eval :: exp \Rightarrow int \Rightarrow int$ such that $eval\ e\ x$ evaluates e at the value x.

A polynomial can be represented as a list of coefficients, starting with the constant. For example, $[4, 2, -1, 3]$ represents the polynomial $4+2x-x^2+3x^3$. Define a function $evalp :: int\ list \Rightarrow int \Rightarrow int$ that evaluates a polynomial at the given value. Define a function $coeffs :: exp \Rightarrow int\ list$ that transforms an expression into a polynomial. This may require auxiliary functions. Prove that *coeffs* preserves the value of the expression: $evalp\ (coeffs\ e)\ x = eval\ e\ x$. Hint: consider the hint in Theorem 2.10.

3

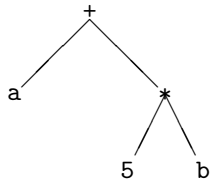
Case Study: IMP Expressions

The methods of the previous chapter suffice to define the arithmetic and boolean expressions of the programming language IMP that is the subject of this book. In this chapter we define their syntax and semantics, write little optimisers for them and show how to compile arithmetic expressions to a simple stack machine. Of course we also prove the correctness of the optimisers and compiler!

3.1 Arithmetic expressions thy

3.1.1 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`	$N\ 5$
`x`	$V\ ''x''$
`x + y`	$Plus\ (V\ ''x'')\ (V\ ''y'')$
`2 + (z + 3)`	$Plus\ (N\ 2)\ (Plus\ (V\ ''z'')\ (N\ 3))$

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 *aexp* 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.

3.1.2 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 ⇒ val
```

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

The value of an arithmetic expression is computed like this:

```
fun aval :: "aexp ⇒ state ⇒ val" where
"aval (N n) s = n" |
"aval (V x) s = s x" |
"aval (Plus a1 a2) s = aval a1 s + aval a2 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'')) (λx. 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:

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

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

$$<''x'' := 7,\ ''y'' := 3>$$

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

It would be easy to add subtraction and multiplication to *aexp* 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 $\mathit{aexp} \Rightarrow \mathit{state} \Rightarrow \mathit{val}$; the return type has to be more complicated.

3.1.3 Constant Folding

Program optimisation is a recurring theme of this book. We start with an extremely simple example, **constant folding**, i.e. the replacement of constant subexpressions by their value. It is performed routinely by compilers. For example, the expression $\mathit{Plus}\ (V\ ''x'')\ (\mathit{Plus}\ (N\ 3)\ (N\ 1))$ is simplified to $\mathit{Plus}\ (V\ ''x'')\ (N\ 4)$. Function *asimp_const* performs constant folding in a bottom-up manner:

```
fun asimp_const :: "aexp ⇒ aexp" where
"asimp_const (N n) = N n" |
```

```
"asimp_const (V x) = V x" |
"asimp_const (Plus a1 a2) =
 (case (asimp_const a1, asimp_const a2) of
   (N n1, N n2) ⇒ N(n1+n2) |
   (b1,b2) ⇒ Plus b1 b2)"
```

Neither N nor V can be simplified further. Given a *Plus*, first the two subexpressions are simplified. If both become numbers, they are added. In all other cases, the results are just recombined with *Plus*.

It is easy to show that *asimp_const* is correct. Correctness means that *asimp_const* does not change the semantics, i.e. the value of its argument:

lemma *"aval (asimp_const a) s = aval a s"*

The proof is by induction on *a*. The two base cases N and V are trivial. In the *Plus* a_1 a_2 case, the induction hypotheses are *aval (asimp_const* a_i*) s* = *aval* a_i *s* for i=1,2. If *asimp_const* a_i = N n_i for i=1,2, then

aval (asimp_const (Plus a_1 a_2*)) s*
= *aval* $(N(n_1+n_2))$ *s* = n_1+n_2
= *aval (asimp_const* a_1*) s* + *aval (asimp_const* a_2*) s*
= *aval (Plus* a_1 a_2*) s*.

Otherwise

aval (asimp_const (Plus a_1 a_2*)) s*
= *aval (Plus (asimp_const* a_1*) (asimp_const* a_2*)) s*
= *aval (asimp_const* a_1*) s* + *aval (asimp_const* a_2*) s*
= *aval (Plus* a_1 a_2*) s*.

This is rather a long proof for such a simple lemma, and boring to boot. In the future we shall refrain from going through such proofs in such excessive detail. We shall simply write "The proof is by induction on *a*." We will not even mention that there is a case distinction because that is obvious from what we are trying to prove, which contains the corresponding *case* expression, in the body of *asimp_const*. We can take this attitude because we merely suppress the obvious and because Isabelle has checked these proofs for us already and you can look at them in the files accompanying the book. The triviality of the proof is confirmed by the size of the Isabelle text:

```
apply (induction a)
apply (auto split: aexp.split)
done
```

The *split* modifier is the hint to *auto* to perform a case split whenever it sees a *case* expression over *aexp*. Thus we guide *auto* towards the case distinction we made in our proof above.

Let us extend constant folding: *Plus* $(N\,0)$ *a* and *Plus a* $(N\,0)$ should be replaced by *a*. Instead of extending *asimp_const* we split the optimisation

process into two functions: one performs the local optimisations, the other one traverses the term. The optimisations can be performed for each *Plus* separately and we define an optimising versions of *Plus*:

```
fun plus :: "aexp ⇒ aexp ⇒ aexp" where
"plus (N i₁) (N i₂) = N(i₁+i₂)" |
"plus (N i) a = (if i=0 then a else Plus (N i) a)" |
"plus a (N i) = (if i=0 then a else Plus a (N i))" |
"plus a₁ a₂ = Plus a₁ a₂"
```

It behaves like *Plus* under evaluation:

```
lemma aval_plus: "aval (plus a₁ a₂) s = aval a₁ s + aval a₂ s"
```

The proof is by induction on a_1 and a_2 using the computation induction rule for *plus* (*plus.induct*). Now we replace *Plus* by *plus* in a bottom-up manner throughout an expression:

```
fun asimp :: "aexp ⇒ aexp" where
"asimp (N n) = N n" |
"asimp (V x) = V x" |
"asimp (Plus a₁ a₂) = plus (asimp a₁) (asimp a₂)"
```

Correctness is expressed exactly as for *asimp_const*:

```
lemma "aval (asimp a) s = aval a s"
```

The proof is by structural induction on a; the *Plus* case follows with the help of Lemma *aval_plus*.

3.1.4 Exercises

Exercise 3.1. To show that *asimp_const* really folds all subexpressions of the form *Plus* (*N i*) (*N j*), define a function *optimal* :: *aexp* ⇒ *bool* that checks that its argument does not contain a subexpression of the form *Plus* (*N i*) (*N j*). Then prove *optimal* (*asimp_const a*).

Exercise 3.2. In this exercise we verify constant folding for *aexp* where we sum up all constants, even if they are not next to each other. For example, *Plus* (*N* 1) (*Plus* (*V x*) (*N* 2)) becomes *Plus* (*V x*) (*N* 3). This goes beyond *asimp*. Define a function *full_asimp* :: *aexp* ⇒ *aexp* that sums up all constants and prove its correctness: *aval* (*full_asimp a*) *s* = *aval a s*.

Exercise 3.3. Substitution is the process of replacing a variable by an expression in an expression. Define a substitution function *subst* :: *vname* ⇒ *aexp* ⇒ *aexp* ⇒ *aexp* such that *subst x a e* is the result of replacing every occurrence of variable *x* by *a* in *e*. For example:

$subst\ ''x''\ (N\ 3)\ (Plus\ (V\ ''x'')\ (V\ ''y'')) = Plus\ (N\ 3)\ (V\ ''y'')$

Prove the so-called **substitution lemma** that says that we can either substitute first and evaluate afterwards or evaluate with an updated state: $aval\ (subst\ x\ a\ e)\ s = aval\ e\ (s(x := aval\ a\ s))$. As a consequence prove $aval\ a_1\ s = aval\ a_2\ s \Longrightarrow aval\ (subst\ x\ a_1\ e)\ s = aval\ (subst\ x\ a_2\ e)\ s$.

Exercise 3.4. Take a copy of theory *AExp* and modify it as follows. Extend type *aexp* with a binary constructor *Times* that represents multiplication. Modify the definition of the functions *aval* and *asimp* accordingly. You can remove *asimp_const*. Function *asimp* should eliminate 0 and 1 from multiplications as well as evaluate constant subterms. Update all proofs concerned.

Exercise 3.5. Define a datatype *aexp2* of extended arithmetic expressions that has, in addition to the constructors of *aexp*, a constructor for modelling a C-like post-increment operation x++, where x must be a variable. Define an evaluation function $aval2 :: aexp \Rightarrow state \Rightarrow val \times state$ that returns both the value of the expression and the new state. The latter is required because post-increment changes the state.

Extend *aexp2* and *aval2* with a division operation. Model partiality of division by changing the return type of *aval2* to $(val \times state)\ option$. In case of division by 0 let *aval2* return *None*.

Exercise 3.6. The following type adds a *LET* construct to arithmetic expressions:

datatype $lexp = Nl\ int \mid Vl\ vname \mid Plusl\ lexp\ lexp \mid LET\ vname\ lexp\ lexp$

The *LET* constructor introduces a local variable: the value of $LET\ x\ e_1\ e_2$ is the value of e_2 in the state where x is bound to the value of e_1 in the original state. Define a function $lval :: lexp \Rightarrow state \Rightarrow int$ that evaluates *lexp* expressions. Remember $s(x := i)$.

Define a conversion $inline :: lexp \Rightarrow aexp$. The expression $LET\ x\ e_1\ e_2$ is inlined by substituting the converted form of e_1 for x in the converted form of e_2. See Exercise 3.3 for more on substitution. Prove that *inline* is correct w.r.t. evaluation.

3.2 Boolean expressions thy

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 $bexp = Bc\ bool \mid Not\ bexp \mid And\ bexp\ bexp \mid Less\ aexp\ aexp$

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 ⇒ state ⇒ bool" where
"bval (Bc v) s = v" |
"bval (Not b) s = (¬ bval b s)" |
"bval (And b1 b2) s = (bval b1 s ∧ bval b2 s)" |
"bval (Less a1 a2) s = (aval a1 s < aval a2 s)"
```

3.2.1 Constant Folding

Constant folding, including the elimination of *True* and *False* in compound expressions, works for *bexp* just like for *aexp*: define optimising versions of the constructors

```
fun not :: "bexp ⇒ bexp" where
"not (Bc True) = Bc False" |
"not (Bc False) = Bc True" |
"not b = Not b"

fun "and" :: "bexp ⇒ bexp ⇒ bexp" where
"and (Bc True) b = b" |
"and b (Bc True) = b" |
"and (Bc False) b = Bc False" |
"and b (Bc False) = Bc False" |
"and b1 b2 = And b1 b2"

fun less :: "aexp ⇒ aexp ⇒ bexp" where
"less (N n1) (N n2) = Bc(n1 < n2)" |
"less a1 a2 = Less a1 a2"
```

and replace the constructors in a bottom-up manner:

```
fun bsimp :: "bexp ⇒ bexp" where
"bsimp (Bc v) = Bc v" |
"bsimp (Not b) = not(bsimp b)" |
"bsimp (And b1 b2) = and (bsimp b1) (bsimp b2)" |
"bsimp (Less a1 a2) = less (asimp a1) (asimp a2)"
```

Note that in the *Less* case we must switch from *bsimp* to *asimp*.

3.2.2 Exercises

Exercise 3.7. Define functions *Eq*, *Le* :: *aexp* ⇒ *aexp* ⇒ *bexp* and prove *bval* (*Eq* a_1 a_2) *s* = (*aval* a_1 *s* = *aval* a_2 *s*) and *bval* (*Le* a_1 a_2) *s* = (*aval* a_1 *s* ⩽ *aval* a_2 *s*).

Exercise 3.8. Consider an alternative type of boolean expressions featuring a conditional:

datatype *ifexp* = *Bc2 bool* | *If ifexp ifexp ifexp* | *Less2 aexp aexp*

First define an evaluation function *ifval* :: *ifexp* ⇒ *bool* analogously to *bval*. Then define two functions *b2ifexp* :: *bexp* ⇒ *ifexp* and *if2bexp* :: *ifexp* ⇒ *bexp* and prove their correctness, i.e., that they preserve the value of an expression.

Exercise 3.9. Define a new type of purely boolean expressions

datatype *bexp* = *VAR vname* | *NOT bexp* | *AND bexp bexp* | *OR bexp bexp*

where variables range over values of type *bool*:

fun *bval* :: "*bexp* ⇒ (*vname* ⇒ *bool*) ⇒ *bool*" **where**
"*bval* (*VAR x*) *s* = *s x*" |
"*bval* (*NOT b*) *s* = (¬ *bval b s*)" |
"*bval* (*AND b*1 *b*2) *s* = (*bval b*1 *s* ∧ *bval b*2 *s*)" |
"*bval* (*OR b*1 *b*2) *s* = (*bval b*1 *s* ∨ *bval b*2 *s*)"

Define a function *is_nnf* :: *bexp* ⇒ *bool* that checks whether a boolean exression is in NNF (negation normal form), i.e., if *NOT* is only applied directly to *VAR*s. Also define a function *nnf* :: *bexp* ⇒ *bexp* that converts a *bexp* into NNF by pushing *NOT* inwards as much as possible. Prove that *nnf* preserves the value (*bval* (*nnf b*) *s* = *bval b s*) and returns an NNF (*is_nnf* (*nnf b*)).

An expression is in DNF (disjunctive normal form) if it is in NNF and if no *OR* occurs below an *AND*. Define a corresponding test *is_dnf* :: *bexp* ⇒ *bool*. An NNF can be converted into a DNF in a bottom-up manner. The critical case is the conversion of *AND* b_1 b_2. Having converted b_1 and b_2, apply distributivity of *AND* over *OR*. Define a conversion function *dnf_of_nnf* :: *bexp* ⇒ *bexp* from NNF to DNF. Prove the correctness of your function: *bval* (*dnf_of_nnf b*) *s* = *bval b s* and *is_nnf b* ⟹ *is_dnf* (*dnf_of_nnf b*).

3.3 Stack Machine and Compilation

This section describes a simple stack machine and compiler for arithmetic expressions. The stack machine has three instructions:

datatype *instr* = *LOADI val* | *LOAD vname* | *ADD*

The semantics of the three instructions will be the following: $LOADI\ n$ (load immediate) puts n on top of the stack, $LOAD\ x$ puts the value of x on top of the stack, and ADD replaces the two topmost elements of the stack by their sum. A stack is simply a list of values:

```
type_synonym stack = "val list"
```

The top of the stack is its first element, the head of the list (see Section 2.2.5). We define two further abbreviations: $hd2\ xs \equiv hd\ (tl\ xs)$ and $tl2\ xs \equiv tl\ (tl\ xs)$.

An instruction is executed in the context of a state and transforms a stack into a new stack:

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where
"exec1 (LOADI n) _ stk  =  n # stk" |
"exec1 (LOAD x) s stk  =  s(x) # stk" |
"exec1  ADD _ stk  =  (hd2 stk + hd stk) # tl2 stk"
```

A list of instructions is executed one by one:

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where
"exec [] _ stk = stk" |
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

The simplicity of this definition is due to the absence of jump instructions. Forward jumps could still be accommodated, but backward jumps would cause a serious problem: execution might not terminate.

Compilation of arithmetic expressions is straightforward:

```
fun comp :: "aexp ⇒ instr list" where
"comp (N n) = [LOADI n]" |
"comp (V x) = [LOAD x]" |
"comp (Plus e1 e2) = comp e1 @ comp e2 @ [ADD]"
```

The correctness statement says that executing a compiled expression is the same as putting the value of the expression on the stack:

```
lemma "exec (comp a) s stk = aval a s # stk"
```

The proof is by induction on a and relies on the lemma

$$exec\ (is_1\ @\ is_2)\ s\ stk = exec\ is_2\ s\ (exec\ is_1\ s\ stk)$$

which is proved by induction in is_1.

Compilation of boolean expressions is covered later and requires conditional jumps.

4

Logic and Proof Beyond Equality

4.1 Formulas

The core syntax of formulas (*form* below) provides the standard logical constructs, in decreasing order of precedence:

$$\begin{aligned} form ::= &(form) \mid \mathit{True} \mid \mathit{False} \mid term = term \\ &\mid \neg\, form \mid form \wedge form \mid form \vee form \mid form \longrightarrow form \\ &\mid \forall x.\ form \mid \exists x.\ form \end{aligned}$$

Terms are the ones we have seen all along, built from constants, variables, function application and λ-abstraction, including all the syntactic sugar like infix symbols, *if*, *case* etc.

Remember that formulas are simply terms of type *bool*. Hence $=$ also works for formulas. Beware that $=$ has a higher precedence than the other logical operators. Hence $s = t \wedge A$ means $(s = t) \wedge A$, and $A \wedge B = B \wedge A$ means $A \wedge (B = B) \wedge A$. Logical equivalence can also be written with $\longleftrightarrow$ instead of $=$, where $\longleftrightarrow$ has the same low precedence as $\longrightarrow$. Hence $A \wedge B \longleftrightarrow B \wedge A$ really means $(A \wedge B) \longleftrightarrow (B \wedge A)$.

Quantifiers need to be enclosed in parentheses if they are nested within other constructs (just like *if*, *case* and *let*).

The most frequent logical symbols and their ASCII representations are shown in Fig. 4.1. The first column shows the symbols, the other columns ASCII representations. The \<...> form is always converted into the symbolic form by the Isabelle interfaces, the treatment of the other ASCII forms depends on the interface. The ASCII forms /\ and \/ are special in that they are merely keyboard shortcuts for the interface and not logical symbols by themselves.

$\forall$	\<forall>	ALL
$\exists$	\<exists>	EX
λ	\<lambda>	%
$\longrightarrow$	-->	
$\longleftrightarrow$	<->	
$\wedge$	/\	&
$\vee$	\/	|
$\neg$	\<not>	~
$\neq$	\<noteq>	~=

Fig. 4.1. Logical symbols and their ASCII forms

The implication $\Longrightarrow$ is part of the Isabelle framework. It structures theorems and proof states, separating assumptions from conclusions. The implication $\longrightarrow$ is part of the logic HOL and can occur inside the formulas that make up the assumptions and conclusion. Theorems should be of the form $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$, not $A_1 \wedge \ldots \wedge A_n \longrightarrow A$. Both are logically equivalent but the first one works better when using the theorem in further proofs.

4.2 Sets

Sets of elements of type *'a* have type *'a set*. They can be finite or infinite. Sets come with the usual notation:

- $\{\}$, $\{e_1,\ldots,e_n\}$
- $e \in A$, $A \subseteq B$
- $A \cup B$, $A \cap B$, $A - B$, $-A$

(where $A - B$ and $-A$ are set difference and complement) and much more. $UNIV$ is the set of all elements of some type. Set comprehension is written $\{x.\ P\}$ rather than $\{x \mid P\}$.

In $\{x.\ P\}$ the x must be a variable. Set comprehension involving a proper term t must be written $\{t \mid x\ y.\ P\}$, where $x\ y$ are those free variables in t that occur in P. This is just a shorthand for $\{v.\ \exists x\ y.\ v = t \wedge P\}$, where v is a new variable. For example, $\{x + y \mid x.\ x \in A\}$ is short for $\{v.\ \exists x.\ v = x+y \wedge x \in A\}$.

Here are the ASCII representations of the mathematical symbols:

$\in$	\<in>	:
$\subseteq$	\<subseteq>	<=
$\cup$	\<union>	Un
$\cap$	\<inter>	Int

Sets also allow bounded quantifications $\forall x{\in}A.\ P$ and $\exists x{\in}A.\ P$.

For the more ambitious, there are also $\bigcup$ and $\bigcap$:

$$\bigcup A = \{x.\ \exists B{\in}A.\ x \in B\} \qquad \bigcap A = \{x.\ \forall B{\in}A.\ x \in B\}$$

The ASCII forms of $\bigcup$ are `\<Union>` and `Union`, those of $\bigcap$ are `\<Inter>` and `Inter`. There are also indexed unions and intersections:

$$(\bigcup_{x\in A} B\ x) = \{y.\ \exists x{\in}A.\ y \in B\ x\}$$
$$(\bigcap_{x\in A} B\ x) = \{y.\ \forall x{\in}A.\ y \in B\ x\}$$

The ASCII forms are `UN x:A. B` and `INT x:A. B` where `x` may occur in `B`. If `A` is `UNIV` you can just write `UN x. B` and `INT x. B`.

Some other frequently useful functions on sets are the following:

$set :: 'a\ list \Rightarrow 'a\ set$	converts a list to the set of its elements
$finite :: 'a\ set \Rightarrow bool$	is true iff its argument is finite
$card :: 'a\ set \Rightarrow nat$	is the cardinality of a finite set and is 0 for all infinite sets
$f\ `\ A = \{y.\ \exists x{\in}A.\ y = f\ x\}$	is the image of a function over a set

See [65] for the wealth of further predefined functions in theory *Main*.

4.3 Proof Automation

So far we have only seen *simp* and *auto*: Both perform rewriting, both can also prove linear arithmetic facts (no multiplication), and *auto* is also able to prove simple logical or set-theoretic goals:

lemma $"\forall x.\ \exists y.\ x = y"$
by *auto*

lemma $"A \subseteq B \cap C \Longrightarrow A \subseteq B \cup C"$
by *auto*

where

by *proof-method*

is short for

apply *proof-method*
done

The key characteristics of both *simp* and *auto* are

- They show you were they got stuck, giving you an idea how to continue.
- They perform the obvious steps but are highly incomplete.

A proof method is **complete** if it can prove all true formulas. There is no complete proof method for HOL, not even in theory. Hence all our proof methods only differ in how incomplete they are.

A proof method that is still incomplete but tries harder than *auto* is *fastforce*. It either succeeds or fails, it acts on the first subgoal only, and it can be modified just like *auto*, e.g. with *simp add*. Here is a typical example of what *fastforce* can do:

```
lemma "⟦ ∀xs ∈ A. ∃ys. xs = ys @ ys;  us ∈ A ⟧
  ⟹ ∃n. length us = n+n"
by fastforce
```

This lemma is out of reach for *auto* because of the quantifiers. Even *fastforce* fails when the quantifier structure becomes more complicated. In a few cases, its slow version *force* succeeds where *fastforce* fails.

The method of choice for complex logical goals is *blast*. In the following example, T and A are two binary predicates. It is shown that if T is total, A is antisymmetric and T is a subset of A, then A is a subset of T:

```
lemma
  "⟦ ∀x y. T x y ∨ T y x;
     ∀x y. A x y ∧ A y x ⟶ x = y;
     ∀x y. T x y ⟶ A x y ⟧
  ⟹ ∀x y. A x y ⟶ T x y"
by blast
```

We leave it to the reader to figure out why this lemma is true. Method *blast*

- is (in principle) a complete proof procedure for first-order formulas, a fragment of HOL. In practice there is a search bound.
- does no rewriting and knows very little about equality.
- covers logic, sets and relations.
- either succeeds or fails.

Because of its strength in logic and sets and its weakness in equality reasoning, it complements the earlier proof methods.

4.3.1 Sledgehammer

Command **sledgehammer** calls a number of external automatic theorem provers (ATPs) that run for up to 30 seconds searching for a proof. Some of these ATPs are part of the Isabelle installation, others are queried over the internet. If successful, a proof command is generated and can be inserted into your proof. The biggest win of **sledgehammer** is that it will take into account the whole lemma library and you do not need to feed in any lemma explicitly. For example,

```
lemma "⟦ xs @ ys = ys @ xs;  length xs = length ys ⟧ ⟹ xs = ys"
```

cannot be solved by any of the standard proof methods, but **sledgehammer** finds the following proof:

by (*metis append_eq_conv_conj*)

We do not explain how the proof was found but what this command means. For a start, Isabelle does not trust external tools (and in particular not the translations from Isabelle's logic to those tools!) and insists on a proof that it can check. This is what *metis* does. It is given a list of lemmas and tries to find a proof just using those lemmas (and pure logic). In contrast to *simp* and friends that know a lot of lemmas already, using *metis* manually is tedious because one has to find all the relevant lemmas first. But that is precisely what **sledgehammer** does for us. In this case lemma *append_eq_conv_conj* alone suffices:

$(xs \mathbin{@} ys = zs) = (xs = take\ (length\ xs)\ zs \wedge ys = drop\ (length\ xs)\ zs)$

We leave it to the reader to figure out why this lemma suffices to prove the above lemma, even without any knowledge of what the functions *take* and *drop* do. Keep in mind that the variables in the two lemmas are independent of each other, despite the same names, and that you can substitute arbitrary values for the free variables in a lemma.

Just as for the other proof methods we have seen, there is no guarantee that **sledgehammer** will find a proof if it exists. Nor is **sledgehammer** superior to the other proof methods. They are incomparable. Therefore it is recommended to apply *simp* or *auto* before invoking **sledgehammer** on what is left.

4.3.2 Arithmetic

By arithmetic formulas we mean formulas involving variables, numbers, $+$, $-$, $=$, $<$, $\leqslant$ and the usual logical connectives $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftrightarrow$. Strictly speaking, this is known as **linear arithmetic** because it does not involve multiplication, although multiplication with numbers, e.g. $2{*}n$ is allowed. Such formulas can be proved by *arith*:

lemma "$[\![\ (a::nat) \leqslant x + b;\ 2{*}x < c\ ]\!] \Longrightarrow 2{*}a + 1 \leqslant 2{*}b + c$"
by *arith*

In fact, *auto* and *simp* can prove many linear arithmetic formulas already, like the one above, by calling a weak but fast version of *arith*. Hence it is usually not necessary to invoke *arith* explicitly.

The above example involves natural numbers, but integers (type *int*) and real numbers (type *real*) are supported as well. As are a number of further operators like *min* and *max*. On *nat* and *int*, *arith* can even prove theorems with quantifiers in them, but we will not enlarge on that here.

4.3.3 Trying Them All

If you want to try all of the above automatic proof methods you simply type

try

You can also add specific simplification and introduction rules:

try *simp*: ... *intro*: ...

There is also a lightweight variant **try0** that does not call sledgehammer.

4.4 Single Step Proofs

Although automation is nice, it often fails, at least initially, and you need to find out why. When *fastforce* or *blast* simply fail, you have no clue why. At this point, the stepwise application of proof rules may be necessary. For example, if *blast* fails on $A \wedge B$, you want to attack the two conjuncts A and B separately. This can be achieved by applying *conjunction introduction*

$$\frac{?P \qquad ?Q}{?P \wedge ?Q}\ conjI$$

to the proof state. We will now examine the details of this process.

4.4.1 Instantiating Unknowns

We had briefly mentioned earlier that after proving some theorem, Isabelle replaces all free variables x by so called **unknowns** *?x*. We can see this clearly in rule *conjI*. These unknowns can later be instantiated explicitly or implicitly:

- By hand, using *of*. The expression *conjI*[*of "a=b" "False"*] instantiates the unknowns in *conjI* from left to right with the two formulas *a=b* and *False*, yielding the rule

  $$\frac{a = b \qquad False}{a = b \wedge False}$$

  In general, $th[of\ string_1\ \ldots\ string_n]$ instantiates the unknowns in the theorem *th* from left to right with the terms $string_1$ to $string_n$.
- By unification. **Unification** is the process of making two terms syntactically equal by suitable instantiations of unknowns. For example, unifying $?P \wedge ?Q$ with $a = b \wedge False$ instantiates *?P* with $a = b$ and *?Q* with *False*.

We need not instantiate all unknowns. If we want to skip a particular one we can just write _ instead, for example *conjI*[*of _ "False"*]. Unknowns can also be instantiated by name, for example *conjI*[*where ?P = "a=b" and ?Q = "False"*].

4.4.2 Rule Application

Rule application means applying a rule backwards to a proof state. For example, applying rule *conjI* to a proof state

1. $\ldots \Longrightarrow A \wedge B$

results in two subgoals, one for each premise of *conjI*:

1. $\ldots \Longrightarrow A$
2. $\ldots \Longrightarrow B$

In general, the application of a rule $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ to a subgoal $\ldots \Longrightarrow C$ proceeds in two steps:

1. Unify A and C, thus instantiating the unknowns in the rule.
2. Replace the subgoal C with n new subgoals A_1 to A_n.

This is the command to apply rule *xyz*:

apply(*rule xyz*)

This is also called **backchaining** with rule *xyz*.

4.4.3 Introduction Rules

Conjunction introduction (*conjI*) is one example of a whole class of rules known as **introduction rules**. They explain under which premises some logical construct can be introduced. Here are some further useful introduction rules:

$$\frac{?P \Longrightarrow ?Q}{?P \longrightarrow ?Q}\ impI \qquad \frac{\bigwedge x.\ ?P\ x}{\forall x.\ ?P\ x}\ allI$$

$$\frac{?P \Longrightarrow ?Q \qquad ?Q \Longrightarrow ?P}{?P = ?Q}\ iffI$$

These rules are part of the logical system of **natural deduction** (e.g. [45]). Although we intentionally de-emphasize the basic rules of logic in favour of automatic proof methods that allow you to take bigger steps, these rules are helpful in locating where and why automation fails. When applied backwards, these rules decompose the goal:

- *conjI* and *iffI* split the goal into two subgoals,
- *impI* moves the left-hand side of a HOL implication into the list of assumptions,
- and *allI* removes a $\forall$ by turning the quantified variable into a fixed local variable of the subgoal.

Isabelle knows about these and a number of other introduction rules. The command

apply *rule*

automatically selects the appropriate rule for the current subgoal.

You can also turn your own theorems into introduction rules by giving them the *intro* attribute, analogous to the *simp* attribute. In that case *blast*, *fastforce* and (to a limited extent) *auto* will automatically backchain with those theorems. The *intro* attribute should be used with care because it increases the search space and can lead to nontermination. Sometimes it is better to use it only in specific calls of *blast* and friends. For example, *le_trans*, transitivity of $\leqslant$ on type *nat*, is not an introduction rule by default because of the disastrous effect on the search space, but can be useful in specific situations:

lemma "$[\![(a::nat) \leqslant b;\ b \leqslant c;\ c \leqslant d;\ d \leqslant e]\!] \Longrightarrow a \leqslant e$"
by(*blast intro*: *le_trans*)

Of course this is just an example and could be proved by *arith*, too.

4.4.4 Forward Proof

Forward proof means deriving new theorems from old theorems. We have already seen a very simple form of forward proof: the *of* operator for instantiating unknowns in a theorem. The big brother of *of* is *OF* for applying one theorem to others. Given a theorem $A \Longrightarrow B$ called r and a theorem A' called r', the theorem $r[OF\ r']$ is the result of applying r to r', where r should be viewed as a function taking a theorem A and returning B. More precisely, A and A' are unified, thus instantiating the unknowns in B, and the result is the instantiated B. Of course, unification may also fail.


Application of rules to other rules operates in the forward direction: from the premises to the conclusion of the rule; application of rules to proof states operates in the backward direction, from the conclusion to the premises.

In general r can be of the form $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$ and there can be multiple argument theorems r_1 to r_m (with $m \leqslant n$), in which case $r[OF\ r_1\ \ldots\ r_m]$ is obtained by unifying and thus proving A_i with r_i, $i = 1 \ldots m$. Here is an example, where *refl* is the theorem $?t = ?t$:

thm *conjI*[*OF refl*[*of* "*a*"] *refl*[*of* "*b*"]]

yields the theorem $a = a \wedge b = b$. The command **thm** merely displays the result.

Forward reasoning also makes sense in connection with proof states. Therefore *blast*, *fastforce* and *auto* support a modifier *dest* which instructs

the proof method to use certain rules in a forward fashion. If r is of the form $A \Longrightarrow B$, the modifier *dest*: r allows proof search to reason forward with r, i.e. to replace an assumption A', where A' unifies with A, with the correspondingly instantiated B. For example, *Suc_leD* is the theorem $Suc\ m \leqslant n \Longrightarrow m \leqslant n$, which works well for forward reasoning:

lemma *"Suc(Suc(Suc a)) ⩽ b ⟹ a ⩽ b"*
by(*blast dest*: *Suc_leD*)

In this particular example we could have backchained with *Suc_leD*, too, but because the premise is more complicated than the conclusion this can easily lead to nontermination.

4.4.5 Finding Theorems

Command **find_theorems** searches for specific theorems in the current theory. Search criteria include pattern matching on terms and on names. For details see the Isabelle/Isar Reference Manual [93].


To ease readability we will drop the question marks in front of unknowns from now on.

4.5 Inductive Definitions

Inductive definitions are the third important definition facility, after datatypes and recursive function. In fact, they are the key construct in the definition of operational semantics in the second part of the book.

4.5.1 An Example: Even Numbers

Here is a simple example of an inductively defined predicate:

- 0 is even
- If n is even, so is $n + 2$.

The operative word "inductive" means that these are the only even numbers. In Isabelle we give the two rules the names *ev0* and *evSS* and write

inductive *ev* :: *"nat ⇒ bool"* **where**
ev0: *"ev 0"* |
evSS: *"ev n ⟹ ev (n + 2)"*

To get used to inductive definitions, we will first prove a few properties of *ev* informally before we descend to the Isabelle level.

How do we prove that some number is even, e.g. *ev* 4? Simply by combining the defining rules for *ev*:

$ev\ 0 \Longrightarrow ev\ (0 + 2) \Longrightarrow ev((0 + 2) + 2) = ev\ 4$

Rule Induction

Showing that all even numbers have some property is more complicated. For example, let us prove that the inductive definition of even numbers agrees with the following recursive one:

fun *even* :: *"nat ⇒ bool"* **where**
"even 0 = True" |
"even (Suc 0) = False" |
"even (Suc(Suc n)) = even n"

We prove $ev\ m \Longrightarrow even\ m$. That is, we assume $ev\ m$ and by induction on the form of its derivation prove $even\ m$. There are two cases corresponding to the two rules for ev:

Case *ev0*: $ev\ m$ was derived by rule $ev\ 0$:
$\Longrightarrow m = 0 \Longrightarrow even\ m = even\ 0 = True$
Case *evSS*: $ev\ m$ was derived by rule $ev\ n \Longrightarrow ev\ (n + 2)$:
$\Longrightarrow m = n + 2$ and by induction hypothesis $even\ n$
$\Longrightarrow even\ m = even(n + 2) = even\ n = True$

What we have just seen is a special case of **rule induction**. Rule induction applies to propositions of this form

$ev\ n \Longrightarrow P\ n$

That is, we want to prove a property $P\ n$ for all even n. But if we assume $ev\ n$, then there must be some derivation of this assumption using the two defining rules for ev. That is, we must prove

Case *ev0*: $P\ 0$
Case *evSS*: $[\![ev\ n;\ P\ n]\!] \Longrightarrow P\ (n + 2)$

The corresponding rule is called $ev.induct$ and looks like this:

$$\frac{ev\ n \qquad P\ 0 \qquad \bigwedge n.\ [\![ev\ n;\ P\ n]\!] \Longrightarrow P\ (n + 2)}{P\ n}$$

The first premise $ev\ n$ enforces that this rule can only be applied in situations where we know that n is even.

Note that in the induction step we may not just assume $P\ n$ but also $ev\ n$, which is simply the premise of rule *evSS*. Here is an example where the local assumption $ev\ n$ comes in handy: we prove $ev\ m \Longrightarrow ev\ (m - 2)$ by induction on $ev\ m$. Case *ev0* requires us to prove $ev\ (0 - 2)$, which follows

from $ev\ 0$ because $0 - 2 = 0$ on type *nat*. In case *evSS* we have $m = n + 2$ and may assume $ev\ n$, which implies $ev\ (m - 2)$ because $m - 2 = (n + 2) - 2 = n$. We did not need the induction hypothesis at all for this proof, it is just a case analysis of which rule was used, but having $ev\ n$ at our disposal in case *evSS* was essential. This case analysis of rules is also called "rule inversion" and is discussed in more detail in Chapter 5.

In Isabelle

Let us now recast the above informal proofs in Isabelle. For a start, we use *Suc* terms instead of numerals in rule *evSS*:

$ev\ n \Longrightarrow ev\ (Suc\ (Suc\ n))$

This avoids the difficulty of unifying $n+2$ with some numeral, which is not automatic.

The simplest way to prove $ev\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$ is in a forward direction: $evSS[OF\ evSS[OF\ ev0]]$ yields the theorem $ev\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$. Alternatively, you can also prove it as a lemma in backwards fashion. Although this is more verbose, it allows us to demonstrate how each rule application changes the proof state:

lemma *"ev(Suc(Suc(Suc(Suc 0))))"*

1. $ev\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$

apply(*rule evSS*)

1. $ev\ (Suc\ (Suc\ 0))$

apply(*rule evSS*)

1. $ev\ 0$

apply(*rule ev0*)
done

Rule induction is applied by giving the induction rule explicitly via the *rule*: modifier:

lemma *"ev m ⟹ even m"*
apply(*induction rule*: *ev.induct*)
by(*simp_all*)

Both cases are automatic. Note that if there are multiple assumptions of the form $ev\ t$, method *induction* will induct on the leftmost one.

As a bonus, we also prove the remaining direction of the equivalence of *ev* and *even*:

lemma *"even n ⟹ ev n"*
apply(*induction n rule*: *even.induct*)

This is a proof by computation induction on *n* (see Section 2.3.4) that sets up three subgoals corresponding to the three equations for *even*:

1. *even* 0 ⟹ *ev* 0
2. *even* (*Suc* 0) ⟹ *ev* (*Suc* 0)
3. ⋀*n*. ⟦*even n* ⟹ *ev n*; *even* (*Suc* (*Suc n*))⟧ ⟹ *ev* (*Suc* (*Suc n*))

The first and third subgoals follow with *ev*0 and *evSS*, and the second subgoal is trivially true because *even* (*Suc* 0) is *False*:

by (*simp_all add*: *ev*0 *evSS*)

The rules for *ev* make perfect simplification and introduction rules because their premises are always smaller than the conclusion. It makes sense to turn them into simplification and introduction rules permanently, to enhance proof automation:

declare *ev.intros*[*simp*,*intro*]

The rules of an inductive definition are not simplification rules by default because, in contrast to recursive functions, there is no termination requirement for inductive definitions.

Inductive Versus Recursive

We have seen two definitions of the notion of evenness, an inductive and a recursive one. Which one is better? Much of the time, the recursive one is more convenient: it allows us to do rewriting in the middle of terms, and it expresses both the positive information (which numbers are even) and the negative information (which numbers are not even) directly. An inductive definition only expresses the positive information directly. The negative information, for example, that 1 is not even, has to be proved from it (by induction or rule inversion). On the other hand, rule induction is tailor-made for proving *ev n* ⟹ *P n* because it only asks you to prove the positive cases. In the proof of *even n* ⟹ *P n* by computation induction via *even.induct*, we are also presented with the trivial negative cases. If you want the convenience of both rewriting and rule induction, you can make two definitions and show their equivalence (as above) or make one definition and prove additional properties from it, for example rule induction from computation induction.

But many concepts do not admit a recursive definition at all because there is no datatype for the recursion (for example, the transitive closure of

a relation), or the recursion would not terminate (for example, an interpreter for a programming language). Even if there is a recursive definition, if we are only interested in the positive information, the inductive definition may be much simpler.

4.5.2 The Reflexive Transitive Closure

Evenness is really more conveniently expressed recursively than inductively. As a second and very typical example of an inductive definition we define the reflexive transitive closure. It will also be an important building block for some of the semantics considered in the second part of the book.

The reflexive transitive closure, called *star* below, is a function that maps a binary predicate to another binary predicate: if r is of type $\tau \Rightarrow \tau \Rightarrow bool$ then *star* r is again of type $\tau \Rightarrow \tau \Rightarrow bool$, and *star* $r\ x\ y$ means that x and y are in the relation *star* r. Think r^* when you see *star* r, because *star* r is meant to be the reflexive transitive closure. That is, *star* $r\ x\ y$ is meant to be true if from x we can reach y in finitely many r steps. This concept is naturally defined inductively:

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

The base case *refl* is reflexivity: $x = y$. The step case *step* combines an r step (from x to y) and a *star* r step (from y to z) into a *star* r step (from x to z). The "**for** r" in the header is merely a hint to Isabelle that r is a fixed parameter of *star*, in contrast to the further parameters of *star*, which change. As a result, Isabelle generates a simpler induction rule.

By definition *star* r is reflexive. It is also transitive, but we need rule induction to prove that:

lemma *star_trans*: *"star r x y ⟹ star r y z ⟹ star r x z"*
apply(*induction rule*: *star.induct*)

The induction is over *star* $r\ x\ y$ and we try to prove *star* $r\ y\ z \Longrightarrow$ *star* $r\ x\ z$, which we abbreviate by $P\ x\ y$. These are our two subgoals:

1. $\bigwedge x.$ *star* $r\ x\ z \Longrightarrow$ *star* $r\ x\ z$
2. $\bigwedge u\ x\ y.$
   $[\![r\ u\ x;$ *star* $r\ x\ y;$ *star* $r\ y\ z \Longrightarrow$ *star* $r\ x\ z;$ *star* $r\ y\ z]\!]$
   $\Longrightarrow$ *star* $r\ u\ z$

The first one is $P\ x\ x$, the result of case *refl*, and it is trivial.

apply(*assumption*)

Let us examine subgoal 2, case *step*. Assumptions $r\ u\ x$ and $star\ r\ x\ y$ are the premises of rule *step*. Assumption $star\ r\ y\ z \Longrightarrow star\ r\ x\ z$ is $P\ x\ y$, the IH coming from $star\ r\ x\ y$. We have to prove $P\ u\ y$, which we do by assuming $star\ r\ y\ z$ and proving $star\ r\ u\ z$. The proof itself is straightforward: from $star\ r\ y\ z$ the IH leads to $star\ r\ x\ z$ which, together with $r\ u\ x$, leads to $star\ r\ u\ z$ via rule *step*:

apply(*metis step*)
done

4.5.3 The General Case

Inductive definitions have approximately the following general form:

inductive I :: "$\tau \Rightarrow bool$" **where**

followed by a sequence of (possibly named) rules of the form

$$[\![\ I\ a_1;\ \ldots;\ I\ a_n\ ]\!] \Longrightarrow I\ a$$

separated by |. As usual, n can be 0. The corresponding rule induction principle $I.induct$ applies to propositions of the form

$$I\ x \Longrightarrow P\ x$$

where P may itself be a chain of implications.

Rule induction is always on the leftmost premise of the goal. Hence $I\ x$ must be the first premise.

Proving $I\ x \Longrightarrow P\ x$ by rule induction means proving for every rule of I that P is invariant:

$$[\![\ I\ a_1;\ P\ a_1;\ \ldots;\ I\ a_n;\ P\ a_n\ ]\!] \Longrightarrow P\ a$$

The above format for inductive definitions is simplified in a number of respects. I can have any number of arguments and each rule can have additional premises not involving I, so-called **side conditions**. In rule inductions, these side-conditions appear as additional assumptions. The **for** clause seen in the definition of the reflexive transitive closure merely simplifies the form of the induction rule.

4.5.4 Exercises

Exercise 4.1. Consider the stack machine from Section 3.3. A **stack underflow** occurs when executing an instruction on a stack containing too few values, e.g., executing an ADD instruction on a stack of size less than two.

Define an inductive predicate $ok :: nat \Rightarrow instr\ list \Rightarrow nat \Rightarrow bool$ such that $ok\ n\ is\ n'$ means that with any initial stack of length n the instructions is can be executed without stack underflow and that the final stack has length n'. Prove that ok correctly computes the final stack size

$$[\![ok\ n\ is\ n';\ length\ stk = n]\!]$$
$$\Longrightarrow length\ (exec\ is\ s\ stk) = n'$$

and that instruction sequences generated by *comp* cannot cause stack underflow: $ok\ n\ (comp\ a)\ ?$ for some suitable value of $?$.

5

Isar: A Language for Structured Proofs

Apply-scripts are unreadable and hard to maintain. The language of choice for larger proofs is **Isar**. The two key features of Isar are:

- It is structured, not linear.
- It is readable without running it because you need to state what you are proving at any given point.

Whereas apply-scripts are like assembly language programs, Isar proofs are like structured programs with comments. A typical Isar proof looks like this:

proof
  assume *"formula*$_0$*"*
  have *"formula*$_1$*"* **by** *simp*
  ⋮
  have *"formula*$_n$*"* **by** *blast*
  show *"formula*$_{n+1}$*"* **by** ...
qed

It proves $formula_0 \Longrightarrow formula_{n+1}$ (provided each proof step succeeds). The intermediate **have** statements are merely stepping stones on the way towards the **show** statement that proves the actual goal. In more detail, this is the Isar core syntax:

proof = **by** *method*
  | **proof** [*method*] *step** **qed**

step = **fix** *variables*
  | **assume** *proposition*
  | [**from** *fact*$^+$] (**have** | **show**) *proposition proof*

proposition = [*name*:] *"formula"*

fact = *name* | ...

A proof can either be an atomic **by** with a single proof method which must finish off the statement being proved, for example *auto*. Or it can be a **proof**–**qed** block of multiple steps. Such a block can optionally begin with a proof method that indicates how to start off the proof, e.g. (*induction xs*).

A step either assumes a proposition or states a proposition together with its proof. The optional **from** clause indicates which facts are to be used in the proof. Intermediate propositions are stated with **have**, the overall goal with **show**. A step can also introduce new local variables with **fix**. Logically, **fix** introduces $\bigwedge$-quantified variables, **assume** introduces the assumption of an implication ($\Longrightarrow$) and **have**/**show** the conclusion.

Propositions are optionally named formulas. These names can be referred to in later **from** clauses. In the simplest case, a fact is such a name. But facts can also be composed with *OF* and *of* as shown in §4.4.4—hence the ... in the above grammar. Note that assumptions, intermediate **have** statements and global lemmas all have the same status and are thus collectively referred to as **facts**.

Fact names can stand for whole lists of facts. For example, if *f* is defined by command **fun**, *f.simps* refers to the whole list of recursion equations defining *f*. Individual facts can be selected by writing *f.simps*(2), whole sublists by *f.simps*(2–4).

5.1 Isar by Example

We show a number of proofs of Cantor's theorem that a function from a set to its powerset cannot be surjective, illustrating various features of Isar. The constant *surj* is predefined.

```
lemma "¬ surj(f :: 'a ⇒ 'a set)"
proof
  assume 0: "surj f"
  from 0 have 1: "∀ A. ∃ a. A = f a" by(simp add: surj_def)
  from 1 have 2: "∃ a. {x. x ∉ f x} = f a" by blast
  from 2 show "False" by blast
qed
```

The **proof** command lacks an explicit method how to perform the proof. In such cases Isabelle tries to use some standard introduction rule, in the above case for ¬:

$$\frac{P \Longrightarrow \mathit{False}}{\neg\ P}$$

In order to prove ¬ *P*, assume *P* and show *False*. Thus we may assume *surj f*. The proof shows that names of propositions may be (single!) digits—

meaningful names are hard to invent and are often not necessary. Both **have** steps are obvious. The second one introduces the diagonal set $\{x.\ x \notin f\ x\}$, the key idea in the proof. If you wonder why 2 directly implies *False*: from 2 it follows that $(a \notin f\ a) = (a \in f\ a)$.

5.1.1 *this*, *then*, *hence* and *thus*

Labels should be avoided. They interrupt the flow of the reader who has to scan the context for the point where the label was introduced. Ideally, the proof is a linear flow, where the output of one step becomes the input of the next step, piping the previously proved fact into the next proof, just like in a UNIX pipe. In such cases the predefined name *this* can be used to refer to the proposition proved in the previous step. This allows us to eliminate all labels from our proof (we suppress the **lemma** statement):

```
proof
  assume "surj f"
  from this have "∃ a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
  from this show "False" by blast
qed
```

We have also taken the opportunity to compress the two **have** steps into one.

To compact the text further, Isar has a few convenient abbreviations:

then = **from** *this*
thus = **then show**
hence = **then have**

With the help of these abbreviations the proof becomes

```
proof
  assume "surj f"
  hence "∃ a. {x. x ∉ f x} = f a" by(auto simp: surj_def)
  thus "False" by blast
qed
```

There are two further linguistic variations:

(**have**|**show**) *prop* **using** *facts* = **from** *facts* (**have**|**show**) *prop*
with *facts* = **from** *facts this*

The **using** idiom de-emphasizes the used facts by moving them behind the proposition.

5.1.2 Structured Lemma Statements: fixes, assumes, shows

Lemmas can also be stated in a more structured fashion. To demonstrate this feature with Cantor's theorem, we rephrase $\neg$ *surj f* a little:

```
lemma
  fixes f :: "'a ⇒ 'a set"
  assumes s: "surj f"
  shows "False"
```

The optional **fixes** part allows you to state the types of variables up front rather than by decorating one of their occurrences in the formula with a type constraint. The key advantage of the structured format is the **assumes** part that allows you to name each assumption; multiple assumptions can be separated by **and**. The **shows** part gives the goal. The actual theorem that will come out of the proof is $surj\ f \Longrightarrow False$, but during the proof the assumption $surj\ f$ is available under the name s like any other fact.

```
proof −
  have "∃ a. {x. x ∉ f x} = f a" using s
    by(auto simp: surj_def)
  thus "False" by blast
qed
```

In the **have** step the assumption $surj\ f$ is now referenced by its name s. The duplication of $surj\ f$ in the above proofs (once in the statement of the lemma, once in its proof) has been eliminated.

Note the dash after the **proof** command. It is the null method that does nothing to the goal. Leaving it out would ask Isabelle to try some suitable introduction rule on the goal *False*—but there is no suitable introduction rule and **proof** would fail.

Stating a lemma with **assumes-shows** implicitly introduces the name *assms* that stands for the list of all assumptions. You can refer to individual assumptions by $assms(1)$, $assms(2)$ etc, thus obviating the need to name them individually.

5.2 Proof Patterns

We show a number of important basic proof patterns. Many of them arise from the rules of natural deduction that are applied by **proof** by default. The patterns are phrased in terms of **show** but work for **have** and **lemma**, too.

We start with two forms of **case analysis**: starting from a formula P we have the two cases P and $\neg\ P$, and starting from a fact $P \vee Q$ we have the two cases P and Q:

```
show "R"
proof cases
  assume "P"
   ⋮
 show "R" ...
next
  assume "¬ P"
   ⋮
 show "R" ...
qed
```

```
have "P ∨ Q" ...
then show "R"
proof
  assume "P"
   ⋮
 show "R" ...
next
  assume "Q"
   ⋮
 show "R" ...
qed
```

How to prove a logical equivalence:

```
show "P ⟷ Q"
proof
  assume "P"
   ⋮
 show "Q" ...
next
  assume "Q"
   ⋮
 show "P" ...
qed
```

Proofs by contradiction:

```
show "¬ P"
proof
  assume "P"
   ⋮
 show "False" ...
qed
```

```
show "P"
proof (rule ccontr)
  assume "¬P"
   ⋮
 show "False" ...
qed
```

The name *ccontr* stands for "classical contradiction".

How to prove quantified formulas:

```
show "∀x. P(x)"
proof
  fix x
   ⋮
 show "P(x)" ...
qed
```

```
show "∃x. P(x)"
proof
   ⋮
 show "P(witness)" ...
qed
```

In the proof of $\forall x.\ P(x)$, the step **fix** x introduces a locally fixed variable x into the subproof, the proverbial "arbitrary but fixed value". Instead of x we could have chosen any name in the subproof. In the proof of $\exists x.\ P(x)$, *witness* is some arbitrary term for which we can prove that it satisfies P.

How to reason forward from $\exists x.\ P(x)$:

have *"*$\exists x.\ P(x)$*"* ...
then obtain x **where** p: *"*$P(x)$*"* **by** *blast*

After the **obtain** step, x (we could have chosen any name) is a fixed local variable, and p is the name of the fact $P(x)$. This pattern works for one or more x. As an example of the **obtain** command, here is the proof of Cantor's theorem in more detail:

lemma *"*$\neg$ *surj*$(f :: 'a \Rightarrow 'a\ set)$*"*
proof
  assume *"surj f"*
  hence *"*$\exists a.\ \{x.\ x \notin f\ x\} = f\ a$*"* **by**(*auto simp*: *surj_def*)
  then obtain a **where** *"*$\{x.\ x \notin f\ x\} = f\ a$*"* **by** *blast*
  hence *"*$a \notin f\ a \longleftrightarrow a \in f\ a$*"* **by** *blast*
  thus *"False"* **by** *blast*
qed

Finally, how to prove set equality and subset relationship:

show *"*$A = B$*"*
proof
  show *"*$A \subseteq B$*"* ...
next
  show *"*$B \subseteq A$*"* ...
qed

show *"*$A \subseteq B$*"*
proof
  fix x
  assume *"*$x \in A$*"*
  ⋮
  show *"*$x \in B$*"* ...
qed

5.3 Streamlining Proofs

5.3.1 Pattern Matching and Quotations

In the proof patterns shown above, formulas are often duplicated. This can make the text harder to read, write and maintain. Pattern matching is an abbreviation mechanism to avoid such duplication. Writing

show *formula* (**is** *pattern*)

matches the pattern against the formula, thus instantiating the unknowns in the pattern for later use. As an example, consider the proof pattern for $\longleftrightarrow$:

show *"formula$_1$ ⟷ formula$_2$"* (**is** *"?L ⟷ ?R"*)
proof
  assume *"?L"*
  ⋮
 show *"?R"* ...
next
  assume *"?R"*
  ⋮
 show *"?L"* ...
qed

Instead of duplicating *formula$_i$* in the text, we introduce the two abbreviations *?L* and *?R* by pattern matching. Pattern matching works wherever a formula is stated, in particular with **have** and **lemma**.

The unknown *?thesis* is implicitly matched against any goal stated by **lemma** or **show**. Here is a typical example:

lemma *"formula"*
proof −
  ⋮
 show *?thesis* ...
qed

Unknowns can also be instantiated with **let** commands

let *?t* = *"some-big-term"*

Later proof steps can refer to *?t*:

have *"... ?t ... "*

! Names of facts are introduced with *name*: and refer to proved theorems. Unknowns *?X* refer to terms or formulas.

Although abbreviations shorten the text, the reader needs to remember what they stand for. Similarly for names of facts. Names like 1, 2 and 3 are not helpful and should only be used in short proofs. For longer proofs, descriptive names are better. But look at this example:

have *x_gr_0*: *"x > 0"*
⋮
from *x_gr_0* ...

The name is longer than the fact it stands for! Short facts do not need names, one can refer to them easily by quoting them:

have *"x > 0"*
⋮
from *`x>0`* ...

Note that the quotes around $x>0$ are **back quotes**. They refer to the fact not by name but by value.

5.3.2 moreover

Sometimes one needs a number of facts to enable some deduction. Of course one can name these facts individually, as shown on the right, but one can also combine them with **moreover**, as shown on the left:

have *"P_1"* ...	**have** lab_1: *"P_1"* ...
moreover have *"P_2"* ...	**have** lab_2: *"P_2"* ...
moreover	⋮
⋮	**have** lab_n: *"P_n"* ...
moreover have *"P_n"* ...	**from** lab_1 lab_2 ...
ultimately have *"P"* ...	**have** *"P"* ...

The **moreover** version is no shorter but expresses the structure more clearly and avoids new names.

5.3.3 Raw Proof Blocks

Sometimes one would like to prove some lemma locally within a proof. A lemma that shares the current context of assumptions but that has its own assumptions and is generalized over its locally fixed variables at the end. This is what a **raw proof block** does:

{ **fix** x_1 ... x_n
  assume A_1 ... A_m
  ⋮
  have B
}

proves $[\![A_1; \ldots; A_m]\!] \Longrightarrow B$ where all x_i have been replaced by unknowns $?x_i$.

! The conclusion of a raw proof block is *not* indicated by **show** but is simply the final **have**.

As an example we prove a simple fact about divisibility on integers. The definition of *dvd* is $(b \ dvd \ a) = (\exists k.\ a = b * k)$.

```
lemma fixes a b :: int assumes "b dvd (a+b)" shows "b dvd a"
proof −
  { fix k assume k: "a+b = b*k"
    have "∃k'. a = b*k'"
    proof
      show "a = b*(k − 1)" using k by(simp add: algebra_simps)
    qed }
  then show ?thesis using assms by(auto simp add: dvd_def)
qed
```

Note that the result of a raw proof block has no name. In this example it was directly piped (via **then**) into the final proof, but it can also be named for later reference: you simply follow the block directly by a **note** command:

note *name* = *this*

This introduces a new name *name* that refers to *this*, the fact just proved, in this case the preceding block. In general, **note** introduces a new name for one or more facts.

5.3.4 Exercises

Exercise 5.1. Give a readable, structured proof of the following lemma:

```
lemma assumes T: "∀ x y. T x y ∨ T y x"
  and A: "∀ x y. A x y ∧ A y x ⟶ x = y"
  and TA: "∀ x y. T x y ⟶ A x y" and "A x y"
shows "T x y"
```

Exercise 5.2. Give a readable, structured proof of the following lemma:

```
lemma "(∃ys zs. xs = ys @ zs ∧ length ys = length zs)
      ∨ (∃ys zs. xs = ys @ zs ∧ length ys = length zs + 1)"
```

Hint: There are predefined functions *take* :: *nat* ⇒ *'a list* ⇒ *'a list* and *drop* :: *nat* ⇒ *'a list* ⇒ *'a list* such that *take k* [x_1,...] = [x_1,...,x_k] and *drop k* [x_1,...] = [x_{k+1},...]. Let *simp* and especially sledgehammer find and apply the relevant *take* and *drop* lemmas for you.

5.4 Case Analysis and Induction

5.4.1 Datatype Case Analysis

We have seen case analysis on formulas. Now we want to distinguish which form some term takes: is it 0 or of the form *Suc n*, is it [] or of the form *x* # *xs*, etc. Here is a typical example proof by case analysis on the form of *xs*:

lemma *"length(tl xs) = length xs − 1"*
proof (*cases xs*)
  assume *"xs = []"*
  thus *?thesis* **by** *simp*
next
  fix *y ys* **assume** *"xs = y#ys"*
  thus *?thesis* **by** *simp*
qed

Function *tl* ("tail") is defined by *tl [] = []* and *tl (x # xs) = xs*. Note that the result type of *length* is *nat* and *0 − 1 = 0*.

This proof pattern works for any term *t* whose type is a datatype. The goal has to be proved for each constructor *C*:

fix $x_1 \ldots x_n$ **assume** *"t = C* $x_1 \ldots x_n$*"*

Each case can be written in a more compact form by means of the **case** command:

case (*C* $x_1 \ldots x_n$)

This is equivalent to the explicit **fix**-**assume** line but also gives the assumption *"t = C* $x_1 \ldots x_n$*"* a name: *C*, like the constructor. Here is the **case** version of the proof above:

proof (*cases xs*)
  case *Nil*
  thus *?thesis* **by** *simp*
next
  case (*Cons y ys*)
  thus *?thesis* **by** *simp*
qed

Remember that *Nil* and *Cons* are the alphanumeric names for [] and #. The names of the assumptions are not used because they are directly piped (via **thus**) into the proof of the claim.

5.4.2 Structural Induction

We illustrate structural induction with an example based on natural numbers: the sum ($\sum$) of the first *n* natural numbers ({0..*n*::*nat*}) is equal to *n* ∗ (*n* + 1) *div* 2. Never mind the details, just focus on the pattern:

lemma *"$\sum${0..n::nat} = n∗(n+1) div 2"*
proof (*induction n*)
  show *"$\sum${0..0::nat} = 0∗(0+1) div 2"* **by** *simp*

next
  fix n **assume** "$\sum\{0..n::nat\} = n*(n+1)\ div\ 2$"
  thus "$\sum\{0..Suc\ n\} = Suc\ n*(Suc\ n+1)\ div\ 2$" **by** *simp*
qed

Except for the rewrite steps, everything is explicitly given. This makes the proof easily readable, but the duplication means it is tedious to write and maintain. Here is how pattern matching can completely avoid any duplication:

lemma "$\sum\{0..n::nat\} = n*(n+1)\ div\ 2$" (**is** "$?P\ n$")
proof (*induction* n)
  show "$?P\ 0$" **by** *simp*
next
  fix n **assume** "$?P\ n$"
  thus "$?P(Suc\ n)$" **by** *simp*
qed

The first line introduces an abbreviation $?P\ n$ for the goal. Pattern matching $?P\ n$ with the goal instantiates $?P$ to the function $\lambda n.\ \sum\{0..n\} = n * (n + 1)\ div\ 2$. Now the proposition to be proved in the base case can be written as $?P\ 0$, the induction hypothesis as $?P\ n$, and the conclusion of the induction step as $?P(Suc\ n)$.

Induction also provides the **case** idiom that abbreviates the **fix-assume** step. The above proof becomes

proof (*induction* n)
  case 0
  show *?case* **by** *simp*
next
  case ($Suc\ n$)
  thus *?case* **by** *simp*
qed

The unknown *?case* is set in each case to the required claim, i.e. $?P\ 0$ and $?P(Suc\ n)$ in the above proof, without requiring the user to define a $?P$. The general pattern for induction over *nat* is shown on the left-hand side:

show *"P(n)"*
proof *(induction n)*
  case 0 — **let** *?case* = *"P(0)"*
  ⋮
  show *?case* ...
next
  case *(Suc n)* — **fix** *n* **assume** *Suc*: *"P(n)"*
  ⋮ — **let** *?case* = *"P(Suc n)"*
  show *?case* ...
qed

On the right side you can see what the **case** command on the left stands for.

In case the goal is an implication, induction does one more thing: the proposition to be proved in each case is not the whole implication but only its conclusion; the premises of the implication are immediately made assumptions of that case. That is, if in the above proof we replace **show** *"P(n)"* by **show** *"A(n) ⟹ P(n)"* then **case** 0 stands for

assume 0: *"A(0)"*
let *?case* = *"P(0)"*

and **case** *(Suc n)* stands for

fix *n*
assume *Suc*: *"A(n) ⟹ P(n)"*
"A(Suc n)"
let *?case* = *"P(Suc n)"*

The list of assumptions *Suc* is actually subdivided into *Suc.IH*, the induction hypotheses (here *A(n) ⟹ P(n)*) and *Suc.prems*, the premises of the goal being proved (here *A(Suc n)*).

Induction works for any datatype. Proving a goal ⟦ $A_1(x)$; ...; $A_k(x)$ ⟧ ⟹ *P(x)* by induction on *x* generates a proof obligation for each constructor *C* of the datatype. The command *case (C* x_1 ... x_n*)* performs the following steps:

1. **fix** x_1 ... x_n
2. **assume** the induction hypotheses (calling them *C.IH*) and the premises A_i(*C* x_1 ... x_n) (calling them *C.prems*) and calling the whole list *C*
3. **let** *?case* = *"P(C* x_1 ... x_n*)"*

5.4.3 Rule Induction

Recall the inductive and recursive definitions of even numbers in Section 4.5:

```
inductive ev :: "nat ⇒ bool" where
ev0: "ev 0" |
evSS: "ev n ⟹ ev(Suc(Suc n))"

fun even :: "nat ⇒ bool" where
"even 0 = True" |
"even (Suc 0) = False" |
"even (Suc(Suc n)) = even n"
```

We recast the proof of $ev\ n \Longrightarrow even\ n$ in Isar. The left column shows the actual proof text, the right column shows the implicit effect of the two **case** commands:

```
lemma "ev n ⟹ even n"
proof(induction rule: ev.induct)
  case ev0                                  let ?case = "even 0"
  show ?case by simp
next
  case evSS                                 fix n
                                            assume evSS: "ev n"
                                                         "even n"
                                            let ?case = "even(Suc(Suc n))"
  thus ?case by simp
qed
```

The proof resembles structural induction, but the induction rule is given explicitly and the names of the cases are the names of the rules in the inductive definition. Let us examine the two assumptions named *evSS*: $ev\ n$ is the premise of rule *evSS*, which we may assume because we are in the case where that rule was used; $even\ n$ is the induction hypothesis.


Because each **case** command introduces a list of assumptions named like the case name, which is the name of a rule of the inductive definition, those rules now need to be accessed with a qualified name, here *ev.ev0* and *ev.evSS*

In the case *evSS* of the proof above we have pretended that the system fixes a variable n. But unless the user provides the name n, the system will just invent its own name that cannot be referred to. In the above proof, we do not need to refer to it, hence we do not give it a specific name. In case one needs to refer to it one writes

```
case (evSS m)
```

just like **case** $(Suc\ n)$ in earlier structural inductions. The name m is an arbitrary choice. As a result, case *evSS* is derived from a renamed version of

rule *evSS*: $ev\ m \Longrightarrow ev(Suc(Suc\ m))$. Here is an example with a (contrived) intermediate step that refers to m:

lemma *"ev n ⟹ even n"*
proof(*induction rule*: *ev.induct*)
  case *ev0* **show** *?case* **by** *simp*
next
  case (*evSS m*)
  have *"even(Suc(Suc m)) = even m"* **by** *simp*
  thus *?case* **using** *`even m`* **by** *blast*
qed

In general, let I be a (for simplicity unary) inductively defined predicate and let the rules in the definition of I be called $rule_1$, ..., $rule_n$. A proof by rule induction follows this pattern:

show *"I x ⟹ P x"*
proof(*induction rule*: *I.induct*)
  case $rule_1$
  ⋮
  show *?case* ...
next
⋮
next
  case $rule_n$
  ⋮
  show *?case* ...
qed

One can provide explicit variable names by writing **case** ($rule_i\ x_1\ \ldots\ x_k$), thus renaming the first k free variables in rule i to $x_1\ \ldots\ x_k$, going through rule i from left to right.

5.4.4 Assumption Naming

In any induction, **case** *name* sets up a list of assumptions also called *name*, which is subdivided into three parts:

name.IH contains the induction hypotheses.
name.hyps contains all the other hypotheses of this case in the induction rule. For rule inductions these are the hypotheses of rule *name*, for structural inductions these are empty.
name.prems contains the (suitably instantiated) premises of the statement being proved, i.e. the A_i when proving $⟦\ A_1; \ldots; A_n\ ⟧ \Longrightarrow A$.

Proof method *induct* differs from *induction* only in this naming policy: *induct* does not distinguish *IH* from *hyps* but subsumes *IH* under *hyps*.

More complicated inductive proofs than the ones we have seen so far often need to refer to specific assumptions—just *name* or even *name.prems* and *name.IH* can be too unspecific. This is where the indexing of fact lists comes in handy, e.g. *name.IH*(2) or *name.prems*(1–2).

5.4.5 Rule Inversion

Rule inversion is case analysis of which rule could have been used to derive some fact. The name **rule inversion** emphasizes that we are reasoning backwards: by which rules could some given fact have been proved? For the inductive definition of *ev*, rule inversion can be summarized like this:

$$ev\ n \Longrightarrow n = 0 \vee (\exists k.\ n = Suc\ (Suc\ k) \wedge ev\ k)$$

The realisation in Isabelle is a case analysis. A simple example is the proof that $ev\ n \Longrightarrow ev\ (n - 2)$. We already went through the details informally in Section 4.5.1. This is the Isar proof:

```
assume "ev n"
from this have "ev(n - 2)"
proof cases
  case ev0 thus "ev(n - 2)" by (simp add: ev.ev0)
next
  case (evSS k) thus "ev(n - 2)" by (simp add: ev.evSS)
qed
```

The key point here is that a case analysis over some inductively defined predicate is triggered by piping the given fact (here: **from** *this*) into a proof by *cases*. Let us examine the assumptions available in each case. In case *ev0* we have $n = 0$ and in case *evSS* we have $n = Suc\ (Suc\ k)$ and $ev\ k$. In each case the assumptions are available under the name of the case; there is no fine grained naming schema like for induction.

Sometimes some rules could not have been used to derive the given fact because constructors clash. As an extreme example consider rule inversion applied to $ev\ (Suc\ 0)$: neither rule *ev0* nor rule *evSS* can yield $ev\ (Suc\ 0)$ because $Suc\ 0$ unifies neither with 0 nor with $Suc\ (Suc\ n)$. Impossible cases do not have to be proved. Hence we can prove anything from $ev\ (Suc\ 0)$:

```
assume "ev(Suc 0)" then have P by cases
```

That is, $ev\ (Suc\ 0)$ is simply not provable:

```
lemma "¬ ev(Suc 0)"
```

proof
  assume *"ev(Suc 0)"* **then show** *False* **by** *cases*
qed

Normally not all cases will be impossible. As a simple exercise, prove that $\neg\ ev\ (Suc\ (Suc\ 0))$.

5.4.6 Advanced Rule Induction

So far, rule induction was always applied to goals of the form $I\ x\ y\ z \Longrightarrow \ldots$ where I is some inductively defined predicate and x, y, z are variables. In some rare situations one needs to deal with an assumption where not all arguments r, s, t are variables:

lemma *"I r s t ⟹ ..."*

Applying the standard form of rule induction in such a situation will lead to strange and typically unproveable goals. We can easily reduce this situation to the standard one by introducing new variables x, y, z and reformulating the goal like this:

lemma *"I x y z ⟹ x = r ⟹ y = s ⟹ z = t ⟹ ..."*

Standard rule induction will worke fine now, provided the free variables in r, s, t are generalized via *arbitrary*.

However, induction can do the above transformation for us, behind the curtains, so we never need to see the expanded version of the lemma. This is what we need to write:

lemma *"I r s t ⟹ ..."*
proof(*induction "r" "s" "t" arbitrary*: *...* *rule*: *I.induct*)

Just like for rule inversion, cases that are impossible because of constructor clashes will not show up at all. Here is a concrete example:

lemma *"ev (Suc m) ⟹ ¬ ev m"*
proof(*induction "Suc m" arbitrary*: *m* *rule*: *ev.induct*)
  fix *n* **assume** *IH*: *"⋀m. n = Suc m ⟹ ¬ ev m"*
  show *"¬ ev (Suc n)"*
  proof — contradition
    assume *"ev(Suc n)"*
    thus *False*
    proof *cases* — rule inversion
      fix *k* **assume** *"n = Suc k"* *"ev k"*
      thus *False* **using** *IH* **by** *auto*
    qed

qed
qed

Remarks:

- Instead of the **case** and *?case* magic we have spelled all formulas out. This is merely for greater clarity.
- We only need to deal with one case because the *ev*0 case is impossible.
- The form of the *IH* shows us that internally the lemma was expanded as explained above: $ev\ x \Longrightarrow x = Suc\ m \Longrightarrow \neg\ ev\ m$.
- The goal $\neg\ ev\ (Suc\ n)$ may suprise. The expanded version of the lemma would suggest that we have a **fix** m **assume** $Suc\ (Suc\ n) = Suc\ m$ and need to show $\neg\ ev\ m$. What happened is that Isabelle immediately simplified $Suc\ (Suc\ n) = Suc\ m$ to $Suc\ n = m$ and could then eliminate m. Beware of such nice surprises with this advanced form of induction.


This advanced form of induction does not support the *IH* naming schema explained in Section 5.4.4: the induction hypotheses are instead found under the name *hyps*, like for the simpler *induct* method.

5.4.7 Exercises

Exercise 5.3. Define a recursive function $elems :: 'a\ list \Rightarrow 'a\ set$ and prove $x \in elems\ xs \Longrightarrow \exists\ ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin elems\ ys$.

Exercise 5.4. A context-free grammar can be seen as an inductive definition where each nonterminal A is an inductively defined predicate on lists of terminal symbols: $A(w)$ mans that w is in the language generated by A. For example, the production $S \to aSb$ can be viewed as the implication $S\ w \Longrightarrow S\ (a\ \#\ w\ @\ [b])$ where a and b are constructors of some datatype of terminal symbols: **datatype** $tsymbs = a \mid b \mid \ldots$

Define the two grammars

$$\begin{array}{lcl} S & \to & \varepsilon \mid a\ S\ b \mid S\ S \\ T & \to & \varepsilon \mid T\ a\ T\ b \end{array}$$

(ε is the empty word) as two inductive predicates and prove $S\ w = T\ w$.

Part II

Semantics

6

Introduction

An introduction.

Explain that/why we present minimalistic language, not a real one.

FIXME: from IMP chapter

The following chapters will then lead us through more interesting applications of the semantics of programming languages and towards deeper theorems about these applications.

Miscellaneous Global Remarks

For reasons of readability we simplify the Isabelle syntax in two respects:

- We no longer enclose types and terms in quotation marks.
- We no longer separate clauses in function definitions or inductive definitions with "|".

We call a proof "automatic" if it requires only a single invocation of a basic Isabelle proof method like *simp*, *auto*, *blast* or *metis*, of course possibly modified with specific lemmas. Inductions are not automatic, although each case can be.

7

IMP: A Simple Imperative Language

To talk about semantics, we first need a programming language. This chapter defines one: a minimalistic imperative programming language called **IMP**.

The main aim of this chapter is to introduce the concepts of commands and their abstract syntax, and to 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.

7.1 IMP Commands thy

Before we jump into any formalisation or define the abstract syntax of commands, we need to determine which constructs the language IMP should contain. The basic constraints are given by our aim to formalise the semantics of an imperative language and to keep things simple. For an imperative language, we will want the basics such as assignments, sequential composition (semicolon), and conditionals (*IF*). To make it Turing-complete, we will want to include *WHILE* loops. 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 will re-use the arithmetic expressions that we have already defined in Chapter 3, and similarly, the conditions in *IF* and *WHILE* will be the boolean expressions defined in the same chapter. A program is then simply one, possibly complex, command in this language.

We have already seen the formalisation of expressions and their semantics in Chapter 3. The abstract syntax of commands is:

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

In the definitions, proofs, and examples further along in this book, 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* c_1 c_2 as $c_1;; c_2$, the term *If b* c_1 c_2 as *IF b THEN* c_1 *ELSE* c_2, 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 $[\![\ldots]\!]$ notation. Nevertheless we still pronounce ";;" as "semicolon".

Example 7.1. 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, the same as for the arithmetic expressions in Chapter 3. 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 this book.

7.2 Big-Step Semantics thy

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.

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 mathematical structures as meanings to programs, e.g. in the so-called denotational style in Chapter 11, or they may be interested in capturing the meaning of programs by describing how to reason about them, e.g. in the axiomatic style in Chapter 12.

7.2.1 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*.

$$\frac{}{(SKIP,\ s) \Rightarrow s}\ Skip \qquad \frac{}{(x ::= a,\ s) \Rightarrow s(x := aval\ a\ s)}\ Assign$$

$$\frac{(c_1,\ s_1) \Rightarrow s_2 \qquad (c_2,\ s_2) \Rightarrow s_3}{(c_1;;\ c_2,\ s_1) \Rightarrow s_3}\ Seq$$

$$\frac{bval\ b\ s \qquad (c_1,\ s) \Rightarrow t}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t}\ IfTrue$$

$$\frac{\neg\ bval\ b\ s \qquad (c_1,\ s) \Rightarrow t}{(IF\ b\ THEN\ c_2\ ELSE\ c_1,\ s) \Rightarrow t}\ IfFalse$$

$$\frac{\neg\ bval\ b\ s}{(WHILE\ b\ DO\ c,\ s) \Rightarrow s}\ WhileFalse$$

$$\frac{bval\ b\ s_1 \qquad (c,\ s_1) \Rightarrow s_2 \qquad (WHILE\ b\ DO\ c,\ s_2) \Rightarrow s_3}{(WHILE\ b\ DO\ c,\ s_1) \Rightarrow s_3}\ WhileTrue$$

Fig. 7.1. The big-step rules of IMP.

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 7.1 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.
- 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

```
inductive
  big_step :: com × state ⇒ state ⇒ bool (infix ⇒ 55)
where
Skip: (SKIP,s) ⇒ s |
Assign: (x ::= a,s) ⇒ s(x := aval a s) |
Seq: ⟦ (c1,s1) ⇒ s2;  (c2,s2) ⇒ s3 ⟧ ⟹ (c1;;c2, s1) ⇒ s3 |
IfTrue: ⟦ bval b s;  (c1,s) ⇒ t ⟧ ⟹ (IF b THEN c1 ELSE c2, s) ⇒ t |
IfFalse: ⟦ ¬bval b s;  (c2,s) ⇒ t ⟧ ⟹ (IF b THEN c1 ELSE c2, s) ⇒ t |
WhileFalse: ¬bval b s ⟹ (WHILE b DO c,s) ⇒ s |
WhileTrue:
⟦ bval b s1;  (c,s1) ⇒ s2;  (WHILE b DO c, s2) ⇒ s3 ⟧
⟹ (WHILE b DO c, s1) ⇒ s3
```

Fig. 7.2. Isabelle definition of the big-step semantics.

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.

Following the rules of Figure 7.1, the corresponding Isabelle definition is straightforward, using the command **inductive** that we introduced in Section 4.5. Again, we make use of the facility for introducing concrete syntax together with the definition in Figure 7.2. Note that we use a tuple as the left side of our $(\_,\_) \Rightarrow \_$ notation and so only need to declare the arrow $\Rightarrow$ as new syntax.

The striking similarity between the rules in the Isabelle definition in Figure 7.2 and the rules in Figure 7.1 is no accident: Figure 7.1 is generated directly from the Isabelle output of Figure 7.2. We will refrain from cluttering the text with such details in the following chapters and use the more readable style of Figure 7.1 in the future. The interested reader will find the full details in the accompanying Isabelle theories.

7.2.2 Deriving IMP Executions

As Figure 7.3 demonstrates, we can use the rules of Figure 7.1 to construct a so-called **derivation tree** that shows a particular execution is admitted

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

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

Fig. 7.3. Derivation tree for execution an IMP program.

by the IMP language. Figure 7.3 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 7.3, 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 7.1 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 7.3 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 $\{t.\ (SKIP,\ \lambda\_.\ 0) \Rightarrow t\}$

but this only shows us $\{\_\}$, 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 the set-comprehension notation introduced in Section 4.2:

values $\{map\ t\ [''x'',\ ''y''] \mid t.\ (''x'' ::= N\ 2,\ \lambda\_.\ 0) \Rightarrow t\}$

This has the result {[2,0]}. In the following chapters, 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.

7.2.3 Rule Inversion

What can we conclude from $(SKIP,\ s) \Rightarrow t$? Clearly $t = s$. This is an example of rule inversion which we had discussed previously in Section 5.4.5. It 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:

$$(SKIP,\ s) \Rightarrow t \Longrightarrow t = s$$

$$(x ::= a,\ s) \Rightarrow t \Longrightarrow t = s(x := aval\ a\ s)$$

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

$$(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \Rightarrow t \Longrightarrow$$
$$bval\ b\ s \wedge (c_1,\ s) \Rightarrow t \vee \neg\ bval\ b\ s \wedge (c_2,\ s) \Rightarrow t$$

$$(WHILE\ b\ DO\ c,\ s) \Rightarrow t \Longrightarrow$$
$$\neg\ bval\ b\ s \wedge t = s \vee$$
$$bval\ b\ s \wedge (\exists s'.\ (c,\ s) \Rightarrow s' \wedge (WHILE\ b\ DO\ c,\ s') \Rightarrow t)$$

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

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

Every $\Longrightarrow$ in the inverted rules can be turned into $\longleftrightarrow$ because the $\Longleftarrow$ 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 7.2. $(c_1;;\ c_2;;\ c_3,\ s) \Rightarrow s' \ \longleftrightarrow\ (c_1;;\ (c_2;;\ c_3),\ s) \Rightarrow s'$

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. □

7.2.4 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* ⇒ *com* ⇒ *bool* (**infix** ∼ 50) **where**
$c \sim c' \equiv (\forall\, s\ t.\ (c,s) \Rightarrow t\ =\ (c',s) \Rightarrow t)$

! Note that the ∼ symbol in this definition is not the standard tilde ~, 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 7.3.
WHILE b DO c $\sim$ *IF b THEN c;; WHILE b DO c ELSE SKIP*

Another example is a trivial contraction of *IF*:

Lemma 7.4. *IF b THEN c ELSE c* $\sim$ *c*

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

Lemma 7.5. $c \sim c' \Longrightarrow$ *WHILE b DO c* $\sim$ *WHILE b DO* c'

is a corollary of

Lemma 7.6.
$[\![(\mathit{WHILE}\ b\ \mathit{DO}\ c,\ s) \Rightarrow t;\ c \sim c']\!] \Longrightarrow (\mathit{WHILE}\ b\ \mathit{DO}\ c',\ s) \Rightarrow t$

This lemma needs the third main proof technique for inductive definitions: rule induction. We covered rule induction in Section 4.5.1. Recall that for the big-step semantics, rule induction applies to properties of the form $(c,\ s) \Rightarrow s' \Longrightarrow P\ c\ s\ s'$. 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. Lemma 7.6 uses the advanced kind of this scheme described in Section 5.4.6 because the big-step premise in the lemma involves not just variables but the proper term *WHILE b DO c*.

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 7.7 (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 7.8. *The semantic equivalence* $\sim$ *is an equivalence relation. It is reflexive:* $c \sim c$, *symmetric:* $c \sim c' \Longrightarrow c' \sim c$, *and transitive:*
$[\![c \sim c';\ c' \sim c'']\!] \Longrightarrow c \sim c''$.

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 why we called Lemma 7.5 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 chapters to show the correctness of program transformations and optimisations.

7.2.5 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 7.9 (IMP is deterministic).
$[\![(c,\ s) \Rightarrow t;\ (c,\ s) \Rightarrow t']\!] \Longrightarrow t' = t$

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. □

While the above proof is nice to show off Isabelle's automation capabilities, it does not give much insight into why the property is true. As mentioned in the first part of this book, we can use the Isar proof language to expand the proof steps for the interesting cases, and omit boring cases using automation. Here is a proof that is closer to a presentation on a blackboard.

theorem
$(c,s) \Rightarrow t \implies (c,s) \Rightarrow t' \implies t' = t$
proof (*induction arbitrary*: t' *rule*: *big_step.induct*)
— the only interesting case, *WhileTrue*:
fix $b\ c\ s\ s_1\ t\ t'$
— The assumptions of the rule:
assume *bval* $b\ s$ **and** $(c,s) \Rightarrow s_1$ **and** $(WHILE\ b\ DO\ c,s_1) \Rightarrow t$
— Ind.Hyp; note the $\bigwedge$ because of arbitrary:
assume *IHc*: $\bigwedge t'.\ (c,s) \Rightarrow t' \Longrightarrow t' = s_1$
assume *IHw*: $\bigwedge t'.\ (WHILE\ b\ DO\ c,s_1) \Rightarrow t' \Longrightarrow t' = t$
— Premise of implication:
assume $(WHILE\ b\ DO\ c,s) \Rightarrow t'$
with *'bval b s'* **obtain** s_1' **where**
c: $(c,s) \Rightarrow s_1'$ **and**
w: $(WHILE\ b\ DO\ c,s_1') \Rightarrow t'$
by *auto*
from *c IHc* **have** $s_1' = s_1$ **by** *blast*
with *w IHw* **show** $t' = t$ **by** *blast*
qed *blast*+ — prove the rest automatically

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.

7.3 Small-Step Semantics thy

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.

$$\frac{}{(x ::= a,\ s) \to (SKIP,\ s(x := aval\ a\ s))}\ Assign$$

$$\frac{}{(SKIP;;\ c_2,\ s) \to (c_2,\ s)}\ Seq1 \qquad \frac{(c_1,\ s) \to (c_1',\ s')}{(c_1;;\ c_2,\ s) \to (c_1';;\ c_2,\ s')}\ Seq2$$

$$\frac{bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_1,\ s)}\ IfTrue$$

$$\frac{\neg\ bval\ b\ s}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_2,\ s)}\ IfFalse$$

$$\frac{}{(WHILE\ b\ DO\ c,\ s) \to (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,\ s)}\ While$$

Fig. 7.4. The small-step rules of IMP.

Going through the rules in Figure 7.4 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 7.2.4. 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 $\rightarrow$, using the *star* that we defined in Section 4.5.2:

abbreviation $op\ \rightarrow\!*\ ::\ com \times state \Rightarrow com \times state \Rightarrow bool$ **where**
$x\ \rightarrow\!*\ y \equiv star\ small\_step\ x\ y$

Example 7.10. 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

$$c = ''x'' ::= V\ ''z'';;\ ''y'' ::= V\ ''x''$$

and an initial state s with

$$s = <''x'' := 3,\ ''y'' := 7,\ ''z'' := 5>$$

we issue the following query to Isabelle

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

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*:

$\{(''x'' ::= V\ ''z'';;\ ''y'' ::= V\ ''x'',\ [3, 7, 5]),$
$(SKIP;;\ ''y'' ::= V\ ''x'',\ [5, 7, 5]),$
$(''y'' ::= V\ ''x'',\ [5, 7, 5]),$
$(SKIP,\ [5, 5, 5])\}$

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 7.11. $[\![cs \rightarrow cs';\ cs \rightarrow cs'']\!] \Longrightarrow cs'' = cs'$

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


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.

7.3.1 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 $cs \Rightarrow t \Longrightarrow cs \rightarrow* (SKIP,\ t)$. 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 7.12.
$(c_1,\ s) \rightarrow* (c_1',\ s') \Longrightarrow (c_1;;\ c_2,\ s) \rightarrow* (c_1';;\ c_2,\ s')$

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. □

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 7.13. $[\![(c_1,\ s_1) \rightarrow* (SKIP,\ s_2);\ (c_2,\ s_2) \rightarrow* (SKIP,\ s_3)]\!]$
$\Longrightarrow (c_1;;\ c_2,\ s_1) \rightarrow* (SKIP,\ s_3)$

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 7.12 together with transitivity of *star*. □

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

Lemma 7.14. $cs \Rightarrow t \Longrightarrow cs \rightarrow* (SKIP,\ t)$

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 7.13. The theory file *Small_Step* also contains a long version of this proof that goes into more detail. □

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 $(c, s) \rightarrow* (SKIP, t) \Longrightarrow (c, s) \Rightarrow t$. 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:

$$[\![(c, s) \rightarrow* (c', s'); (c', s') \Rightarrow t]\!] \Longrightarrow (c, s) \Rightarrow t$$

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 7.15 (Step case). $[\![cs \rightarrow cs'; cs' \Rightarrow t]\!] \Longrightarrow cs \Rightarrow t$

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

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

Lemma 7.16 (Small-step implies big-step).
$[\![cs \rightarrow* cs'; cs' \Rightarrow t]\!] \Longrightarrow cs \Rightarrow t$

Proof. As mentioned, the proof is by induction on the reflexive transitive closure, and the step case is solved by Lemma 7.15. □

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

Corollary 7.17. $cs \rightarrow* (SKIP, t) \Longrightarrow cs \Rightarrow t$

Proof. As planned, we use Lemma 7.16 and instantiate cs' to $(SKIP, t)$, which collapses the second premise of Lemma 7.16 to *True*. □

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

Corollary 7.18. $(c, s) \Rightarrow t \longleftrightarrow (c, s) \rightarrow* (SKIP, t)$

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.

7.3.2 Final configurations, infinite reductions, and termination

As opposed to the big-step case, in the small-step semantics it is possible to talk about non-terminating executions directly. We can easily distinguish final configurations from those that can make further progress:

definition *final* :: $com \times state \Rightarrow bool$ **where**
$final\ cs \longleftrightarrow \neg (\exists\, cs'.\ cs \rightarrow cs')$

In our semantics, these happen to be exactly the configurations that have $SKIP$ as their command.

Lemma 7.19. $final\ (c, s) = (c = SKIP)$

Proof. One direction is easy: clearly, if the command c is $SKIP$, the configuration is final. The other direction is not much harder. It is proved automatically after induction on c. □

With this we can show that $\Rightarrow$ yields a final state iff $\rightarrow$ terminates:

Lemma 7.20. $(\exists\, t.\ cs \Rightarrow t) \longleftrightarrow (\exists\, cs'.\ cs \rightarrow* cs' \wedge final\ cs')$

Proof. Using Lemma 7.19 we can replace *final* with configurations that have $SKIP$ as the command. The rest follows from the equivalence of small and big-step semantics and considering each direction separately. □

This lemma says that the absence of a big-step result is equivalent to non-termination in IMP. This is not necessarily the case for any language. Another reason for the absence of a big-step result may be a runtime error in the execution of the program that leads to no big-step rule being applicable. In the big-step semantics this is often indistinguishable from non-termination. In the small-step semantics the concept of final configurations neatly distinguishes the two causes.

Since IMP is deterministic, there is no difference between may and must termination in this formulation. Consider a language with non-determinism. In such a language, if we had an execution sequence in the small-step semantics that ended in a final configuration, we would only know that one of the possible executions terminates — the command *may* terminate. A stronger formulation would be to require that all executions end in a final configuration, i.e. that the command *must* terminate. This would require a non-trivial reformulation of Lemma 7.20 and its proof.

7.4 Summary and Further Reading

This concludes the chapter on the operational semantics for IMP. In the first part of this chapter, we have defined the abstract syntax of IMP commands, we have defined the semantics of IMP in terms of a big-step operational semantics, and we have experimented with the concepts of semantic equivalence and determinism. In the second part of this chapter, we have defined an alternative form of operational semantics, namely small-step semantics, and we have proved that this alternative form describes the same behaviours as the big-step semantics. The two forms of semantics have different application trade-offs: big-step semantics were easier to define and understand, small-step semantics let us talk explicitly about intermediate states of execution and about termination.

We have looked at three main proof techniques: derivation trees, rule inversion, and rule induction. These three techniques form the basic tool set that will accompany us in the following chapters, which show deeper applications of semantics, including compiler correctness, type systems, program analysis and abstract interpretation.

Operational semantics in its small-step form goes back to Plotkin [71, 73, 72]. Big-step semantics was popularised by Kahn [47] and his co-workers.

There are many programming language constructs that we have left out in IMP. Some examples that are relevant for imperative and object-oriented languages are the following.

Syntax. `For` loops, `do ... while` loops, the `if ... then` command, etc are just further syntactic forms of the basic commands above. They could either be formalised directly, or they could be transformed into equivalent basic forms by syntactic de-sugaring.

Jumps. The `goto` construct, even though considered harmful [28], is relatively easy to formalise. It merely requires the introduction of some notion of program position, be that as an explicit program counter, or a set of labels for jump targets. Some intermediate languages used for program

verification, such as Boogie [10] for instance, use `goto` as their main loop construct and treat all others as syntactic sugar.

Blocks and local variables. As the other constructs they do not add anything in terms of computability, but they are a good tool for programmers to make use of data hiding and encapsulation. The main formalisation challenge with local variables is their visibility scope. There are a number of choices. The most intuitive one may be to have local variables appear and disappear from the state space of the program according to the scoping rules of the language. This is hard to formalise in a theorem prover, because the type of program states will then change throughout the formalisation and depend on the program. Another choice (among many) would be to include all possible local variables in the state space and formalise access to their values according to the visibility rules of the language.

Procedures. Procedure parameters introduce issues similar to those encountered for local variables, but they may have additional complexities depending on which calling conventions the language implements (call by reference, call by value, call by name, etc). Recursion is not usually a problem to formalise, although it took the field a while to figure out what the right Hoare logic rules for procedure calls are. Procedures also influence the definition of what a program is: instead of a single command, a program now usually becomes a list or collection of procedures.

Exceptions. Throwing and catching exceptions again is usually reasonably easy to integrate into a language formalisation, it merely adds a further concept into the semantics. Formalising exceptions may interact with features like procedures and local variables, because exceptions provide new ways to exit scopes and procedures.

Further data types, structs, pointers, references, arrays. Data types are often orthogonal to the control structures of a language. Further primitive datatypes such as fixed size machine words, records, and arrays are easy to include when the corresponding high-level concept is available or easy to formalise in the theorem prover (IEEE floating point values may induce an interesting amount of work, for instance [25]). The semantics of heap concepts such as pointers and references again is an orthogonal issue and can be treated at various level of detail, from raw bytes [88] up to multiple heaps separated by type and field names [14].

Objects, classes, methods. Object oriented features in many variations have been the target of a lot of study in the past decade. Objects and methods lead to a stronger connection between control structures and memory. Which virtual method will be executed, for instance, depends on the type of the object in memory at runtime. It is by now well un-

derstood how to formalise them in a theorem prover. For a study on a Java-like language in Isabelle/HOL, see for instance Jinja [49] or Featherweight Java [46].

All of the features above can be formalised in a theorem prover. Many add interesting complications, but the song remains the same. Schirmer [79], for instance, shows an Isabelle formalisation of big-step and small-step semantics in the style of this chapter for the generic imperative language Simpl. The formalisation includes procedures, blocks, exceptions, and further advanced concepts. With techniques similar to those described Chapter 12, he progresses the language to a point where it can directly be used for large-scale program verification.

Exercises

Exercise 7.21. Define a function *skip* :: *com* ⇒ *bool* that determines if a command behaves like *SKIP*. Prove *skip c* ⟹ *c* ∼ *SKIP* by induction on *c*.

Exercise 7.22. Define a function *deskip* :: *com* ⇒ *com* that eliminates as many *SKIP*s as possible from a command. For example:

deskip (*SKIP*;; *WHILE b DO* (*x* ::= *a*;; *SKIP*)) = *WHILE b DO x* ::= *a*

Prove *deskip c* ∼ *c* by induction on *c*. Remember Lemma 7.5 for the *WHILE* case.

Exercise 7.23. Extend IMP with a *REPEAT c UNTIL b* command. Adjust the definition of big-step and small-step semantics and update the equivalence proof between them.

Exercise 7.24. Extend IMP with a new command c_1 *OR* c_2 that is a nondeterministic choice: it may execute either c_1 or c_2. Adjust the definition of big-step semantics and small-step semantics (you need two more rules for each semantics), prove (c_1 *OR* c_2) ∼ (c_2 *OR* c_1) and update the equivalence proof between the two semantics.

Exercise 7.25. Extend IMP with exceptions. In particular, add commands *THROW* and *CATCH c* such that *THROW* stops normal control flow and skips commands until the program either terminates or a corresponding *CATCH c* command is encountered. The *CATCH c* command executes the exception handler *c* if an exception is active, and is equivalent to *SKIP* if no exception is active. Adjust the definition of big-step and small-step semantics and update the equivalence proof between them.

Exercise 7.26. Use the exception mechanism from the previous exercise to implement the commands *BREAK* and *CONTINUE* in the small-step or big-step semantics of IMP. Hint: extend the state space to indicate which kind of exception is active.

Exercise 7.27. Define a small-step semantics for the evaluation of boolean and arithmetic expressions. Prove that it is equivalent to the original definition.

Exercise 7.28. Extend the small-step semantics with parallel execution: c_1 *PAR* c_2 is executed by interleaving steps of c_1 and c_2. The scheduling is completely nondeterministic: at any point either c_1 or c_2 can make a step. Don't forget to take care of *SKIP*.

8

Compiler

This chapter presents a first application of programming language semantics: proving compiler correctness. To this end, we will define a small machine language based on a simple stack machine. Stack machines are common low-level intermediate languages, the Java Virtual Machine is one example. We then write a compiler from IMP to this language and prove that the compiled program has the same semantics as the source program. The compiler will perform a very simple standard optimisation for boolean expressions, but is otherwise non-optimising.

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

8.1 Instructions and Stack Machine thy

We begin by defining the instruction set architecture and semantics of our stack machine. We have already seen a very simple stack machine language in Section 3.3. In this section, we extend this language with memory writes and jump instructions.

Working with proofs on the machine language, we will find it convenient for the program counter to admit negative values, i.e. to be of type *int* instead of the initially more intuitive *nat*. The effect of this choice is that various decomposition lemmas about machine executions have nicer algebraic properties and less preconditions than their *nat* counterparts. Such effects are usually discovered during the proof.

As in Section 3.3, our machine language models programs as lists of instructions. Our *int* program counter will need to index into these lists. Isabelle comes with a predefined list index operator *nth*, but it works on *nat*. Instead of constantly converting between *int* and *nat* and dealing with the arising side-conditions in proofs, we define our own *int* version of *nth*, i.e. $i :: int$:

$$(x \# xs) !! i = (\mathit{if}\ i = 0\ \mathit{then}\ x\ \mathit{else}\ xs !! (i - 1))$$

However, we still need the conversion $int :: nat \Rightarrow int$ because the length of a list is of type *nat*. To reduce clutter we introduce the abbreviation

$$\mathit{size}\ xs \equiv \mathit{int}(\mathit{length}\ xs)$$

The !! operator distributes over @ in the expected way:

Lemma 8.1. *If* $0 \leqslant i$,
$(xs\ @\ ys) !! i = (\mathit{if}\ i < \mathit{size}\ xs\ \mathit{then}\ xs !! i\ \mathit{else}\ ys !! (i - \mathit{size}\ xs))$

We are now ready to define the machine itself. To keep things simple, we directly reuse the concepts of values and variable names from the source language. In a more realistic setting, we would explicitly map variable names to memory locations, instead of using strings as addresses. We skip this step here for clarity, adding it does not pose any fundamental difficulties.

The instructions in our machine are the following. The first three are familiar from Section 3.3:

datatype *instr* =
$LOADI\ int \mid LOAD\ vname \mid ADD \mid STORE\ vname \mid$
$JMP\ int \mid JMPLESS\ int \mid JMPGE\ int$

Instruction *LOADI* loads an immediate value onto the stack, *LOAD* loads the value of a variable, *ADD* adds the two topmost stack values, *STORE* stores the top of stack into memory, *JMP* jumps by a relative value, *JMPLESS* compares the two topmost stack elements and jumps if the second one is less, and finally *JMPGE* compares and jumps if the second one is greater or equal.

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

As in the source language, we proceed by defining the state such programs operate on, followed by the definition of the semantics itself.

Program configurations consist of an *int* program counter, a memory state for which we re-use the type *state* from the source language, and a stack which we model as a list of values:

type_synonym *stack* = *val list*
type_synonym *config* = *int* × *state* × *stack*

We now define the semantics of machine execution. Similarly to the small-step semantics of the source language, we do so in multiple levels: first, we define what effect a single instruction has on a configuration, then we define how an instruction is selected from the program, and finally we take the reflexive transitive closure to get full machine program executions.

We encode the behaviour of single instructions in the function *iexec*. The program counter is i, usually incremented by 1, except for the jump instructions. Variables are loaded from and stored into the variable state s with function application and function update. For the stack *stk*, we use standard list constructs as well as $hd2\ xs \equiv hd\ (tl\ xs)$ and $tl2\ xs \equiv tl\ (tl\ xs)$ from Section 3.3.

fun *iexec* :: *instr* ⇒ *config* ⇒ *config* **where**

$$
\begin{aligned}
&iexec\ (LOADI\ n)\ (i,\ s,\ stk) &&= (i + 1,\ s,\ n\ \#\ stk)\\
&iexec\ (LOAD\ x)\ (i,\ s,\ stk) &&= (i + 1,\ s,\ s\ x\ \#\ stk)\\
&iexec\ ADD\ (i,\ s,\ stk) &&= (i + 1,\ s,\ (hd2\ stk + hd\ stk)\ \#\ tl2\ stk)\\
&iexec\ (STORE\ x)\ (i,\ s,\ stk) &&= (i + 1,\ s(x := hd\ stk),\ tl\ stk)\\
&iexec\ (JMP\ n)\ (i,\ s,\ stk) &&= (i + 1 + n,\ s,\ stk)\\
&iexec\ (JMPLESS\ n)\ (i,\ s,\ stk) &&=\\
&\quad (if\ hd2\ stk < hd\ stk\ then\ i + 1 + n\ else\ i + 1,\ s,\ tl2\ stk)\\
&iexec\ (JMPGE\ n)\ (i,\ s,\ stk) &&=\\
&\quad (if\ hd\ stk \leqslant hd2\ stk\ then\ i + 1 + n\ else\ i + 1,\ s,\ tl2\ stk)
\end{aligned}
$$

The next level up, a single execution step selects the instruction the program counter (pc) points to and uses *iexec* to execute it. For execution to be well-defined, we additionally check if the pc points to a valid location in the list. We call this predicate *exec*1 and give it the notation $P \vdash c \to c'$ for *program P executes from configuration c to configuration c′*.

definition *exec*1 :: *instr list* ⇒ *config* ⇒ *config* ⇒ *bool* **where**

$$
\begin{aligned}
&P \vdash c \to c' =\\
&(\exists i\ s\ stk.\ c = (i,\ s,\ stk) \wedge c' = iexec\ (P\ !!\ i)\ c \wedge 0 \leqslant i < size\ P)
\end{aligned}
$$

where $x \leqslant y < z \equiv x \leqslant y \wedge y < z$ as usual in mathematics.

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

abbreviation $P \vdash c \to* c' \equiv star\ (exec1\ P)\ c\ c'$

This concludes our definition of the machine and its semantics. As usual in this book, the definitions are executable. This means, we can try out a simple example. Let $P = [LOAD\ ''y'',\ STORE\ ''x'']$, $s\ ''x'' = 3$, and $s\ ''y'' = 4$. Then

values $\{(i,\ map\ t\ [''x'',\ ''y''],\ stk)\ |i\ t\ stk.\ P \vdash (0,\ s,\ []) \rightarrow\!* (i,\ t,\ stk)\}$

will produce the following sequence of configurations:

$\{(0,\ [3,\ 4],\ []),\ (1,\ [3,\ 4],\ [4]),\ (2,\ [4,\ 4],\ [])\}$

8.2 Reasoning about machine executions thy

The compiler proof is more involved than the short proofs we have seen so far. We will need a small number of technical lemmas before we get to the compiler correctness problem itself. Our aim in this section is to execute machine programs symbolically as far as possible using Isabelle's proof tools. We will then use this ability in the compiler proof to assemble machine executions from smaller into larger parts.

A first lemma to this end is that execution results are preserved if we append additional code to the left or right of a program. Appending at the right side is easy:

Lemma 8.2. $P \vdash c \rightarrow\!* c' \Longrightarrow P\ @\ P' \vdash c \rightarrow\!* c'$

Proof. The proof is by induction on the reflexive transitive closure. For the step case, we observe after unfolding of *exec*1 that appending program context on the right does not change the result of indexing into the original instruction list. □

Appending code on the left side requires shifting the program counter.

Lemma 8.3.
$P \vdash (i,\ s,\ stk) \rightarrow\!* (i',\ s',\ stk') \Longrightarrow$
$P'\ @\ P \vdash (size\ P' + i,\ s,\ stk) \rightarrow\!* (size\ P' + i',\ s',\ stk')$

Proof. The proof is again by induction on the reflexive transitive closure and reduction to *exec*1 in the step case. To show the lemma for *exec*1, we unfold its definition and observe Lemma 8.4. □

The execution of a single instruction can be relocated arbitrarily:

Lemma 8.4.
$(n + i',\ s',\ stk') = iexec\ x\ (n + i,\ s,\ stk) \longleftrightarrow$
$(i',\ s',\ stk') = iexec\ x\ (i,\ s,\ stk)$

Proof. We observe by case distinction on the instruction x that the only component of the result configuration that is influenced by the program counter i is the first one, and in this component only additively. For instance, in the $LOADI\ n$ instruction, we get on both sides $s' = s$ and $stk' = n \# stk$. The pc field for input i on the right-hand side is $i' = i + 1$. The pc field for $n + i$ on the left-hand side becomes $n + i + 1$, which is $n + i'$ as required. The other cases are analogous. □

Taking these two lemmas together, we can compose separate machine executions into one larger one.

Lemma 8.5 (Composing machine executions).
$[\![P \vdash (0,\ s,\ stk) \rightarrow* (i',\ s',\ stk');\ size\ P \leqslant i';$
$P' \vdash (i' - size\ P,\ s',\ stk') \rightarrow* (i'',\ s'',\ stk'');\ j'' = size\ P + i'']\!]$
$\Longrightarrow P\ @\ P' \vdash (0,\ s,\ stk) \rightarrow* (j'',\ s'',\ stk'')$

Proof. The proof is by suitably instantiating Lemma 8.2 and Lemma 8.3. □

8.3 Compilation thy

We are now ready to define the compiler, and will do so in the three usual steps: first for arithmetic expressions, then for boolean expressions, and finally for commands. We have already seen compilation of arithmetic expressions in Section 3.3. We define the same function for our extended machine language:

fun $acomp :: aexp \Rightarrow instr\ list$ **where**

$$\begin{aligned} acomp\ (N\ n) &= [LOADI\ n] \\ acomp\ (V\ x) &= [LOAD\ x] \\ acomp\ (Plus\ a_1\ a_2) &= acomp\ a_1\ @\ acomp\ a_2\ @\ [ADD] \end{aligned}$$

The correctness statement is not as easy any more as in Section 3.3, because program execution now is a relation, not simply a function. For our extended machine language a function is not suitable because we now have potentially non-terminating executions. This is not a big obstacle: we can still express naturally that the execution of a compiled arithmetic expression will leave the result on top of the stack, and that the program counter will point to the end of the compiled expression.

Lemma 8.6 (Correctness of $acomp$).
$acomp\ a \vdash (0,\ s,\ stk) \rightarrow* (size\ (acomp\ a),\ s,\ aval\ a\ s \# stk)$

Proof. The proof is by induction on the arithmetic expression. □

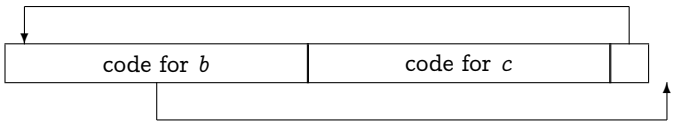


Fig. 8.1. Compilation of *WHILE b DO c*

The compilation schema for boolean expressions is best motivated by a preview of the layout of the code generated from *WHILE b DO c* shown in Figure 8.1. Arrows indicate jump instructions. Let *cb* be code generated from *b*. If *b* evaluates to *True*, the execution of *cb* should lead to the end of *cb*, continue with the execution of the code for *c* and jump back to the beginning of *cb*. If *b* evaluates to *False*, the execution of *cb* should jump behind all of the loop code. For example, when executing the compiled code for *WHILE And* b_1 b_2 *DO c*, after having found that b_1 evaluates to *False* we can safely jump out of the loop. There can be multiple such jumps: think of *And* b_1 (*And* b_2 b_3).

To support this schema, the *bexp* compiler takes two further parameters in addition to *b*: an offset *n* and a flag $f :: bool$ that determines for which value of *b* the generated code should jump to offset *n*. This enables us to perform a small bit of optimisation: boolean constants do not need to execute any code, they either compile to nothing or to a jump to the offset, depending on the value of *f*. The *Not* case simply inverts *f*. The *And* case performs shortcut evaluation as explained above. The *Less* operator uses the *acomp* compiler for a_1 and a_2 and then selects the appropriate compare and jump instruction according to *f*.

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

Example 8.7. Boolean constants are optimised away:

```
value bcomp (And (Bc True) (Bc True)) False 3
returns []
```

value *bcomp* (*And* (*Bc False*) (*Bc True*)) *True* 3
returns [*JMP* 1, *JMP* 3]

value *bcomp* (*And* (*Less* (*V* ''*x*'') (*V* ''*y*'')) (*Bc True*)) *False* 3
returns [*LOAD* ''*x*'', *LOAD* ''*y*'', *JMPGE* 3]

The second example shows that the optimisation is not perfect: it may generate dead code.

The correctness statement is the following: the stack and state should remain unchanged and the program counter should indicate if the expression evaluated to *True* or *False*. If $f = False$ then we end at *size* (*bcomp b f n*) in the *True* case and *size* (*bcomp b f n*) + *n* in the *False* case. If $f = True$ it is the other way around. This statement only makes sense for forward jumps, so we require n to be non-negative.

Lemma 8.8 (Correctness of *bcomp*).
Let $pc' =$ *size* (*bcomp b f n*) + (*if f* = *bval b s then n else* 0). *Then*
$0 \leqslant n \Longrightarrow$ *bcomp b f n* $\vdash$ (0, *s*, *stk*) $\rightarrow*$ (*pc*′, *s*, *stk*)

Proof. The proof is by induction on *b*. The constant and *Less* cases are solved automatically. For the *Not* case, we instantiate the induction hypothesis manually to $\neg f$. For *And*, we get two recursive cases. The first needs the induction hypothesis instantiated with *size* (*bcomp* b_2 *f n*) + (*if f then* 0 *else n*) for *n*, and with *False* for *f*, and the second goes through with simply *n* and *f*. □

With both expression compilers in place, we can now proceed to the command compiler *ccomp*. The idea is to compile *c* into a program that will perform the same state transformation as *c* and that will always end with the program counter at *size* (*ccomp c*). It may push and consume intermediate values on the stack for expressions, but at the end, the stack will be the same as in the beginning. Because of this modular behaviour of the compiled code, the compiler can be defined recursively as follows:

- *SKIP* compiles to the empty list;
- for assignment, we compile the expression and store the result;
- sequential composition just appends the corresponding machine programs;
- *IF* is compiled as shown in Figure 8.2;.
- *WHILE* is compiled as shown in Figure 8.1.

Figure 8.3 shows the formal definition.

Since everything in *ccomp* is executable, we can inspect the results of compiler runs directly in the theorem prover. For example, let p_1 be the command for `IF u < 1 THEN u := u + 1 ELSE v := u`. Then

value *ccomp* p_1

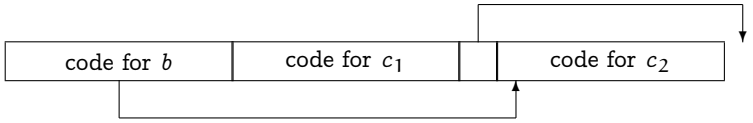


Fig. 8.2. Compilation of *IF b THEN* c_1 *ELSE* c_2

fun *ccomp* :: *com* ⇒ *instr list* **where**

ccomp SKIP = []
ccomp (*x* ::= *a*) = *acomp a* @ [*STORE x*]
ccomp (c_1;; c_2) = *ccomp* c_1 @ *ccomp* c_2
ccomp (*IF b THEN* c_1 *ELSE* c_2) =
(*let* cc_1 = *ccomp* c_1; cc_2 = *ccomp* c_2;
cb = *bcomp b False* (*size* cc_1 + 1)
in cb @ cc_1 @ *JMP* (*size* cc_2) # cc_2)
ccomp (*WHILE b DO c*) =
(*let cc* = *ccomp c*; *cb* = *bcomp b False* (*size cc* + 1)
in cb @ *cc* @ [*JMP* (− (*size cb* + *size cc* + 1))])

Fig. 8.3. Definition of *ccomp*.

results in

[*LOAD ''u''*, *LOADI* 1, *JMPGE* 5, *LOAD ''u''*, *LOADI* 1, *ADD*, *STORE ''u''*, *JMP* 2, *LOAD ''u''*, *STORE ''v''*]

Similarly for loops. Let p_2 be `WHILE u < 1 DO u := u + 1`. Then

value *ccomp* p_2

results in

[*LOAD ''u''*, *LOADI* 1, *JMPGE* 5, *LOAD ''u''*, *LOADI* 1, *ADD*, *STORE ''u''*, *JMP* −8]

8.4 Preservation of semantics thy

This section shows the correctness proof of our small toy compiler. For IMP, the correctness statement is fairly straightforward: the machine program should have precisely the same behaviour as the source program. It should cause the same state change and nothing more than that. It should terminate if and only if the source program terminates. Since we use the same type for states at the source level and machine level, the first part of the property is easy to express. Similarly, it is easy to say that the stack should not change.

Finally, we can express correct termination by saying that the machine execution, started at $pc = 0$ should stop at the end of the machine code with $pc = size\ (ccomp\ c)$. In total, we have

$$(c,\ s) \Rightarrow t \ \longleftrightarrow\ \ ccomp\ c \vdash (0,\ s,\ stk) \rightarrow* (size\ (ccomp\ c),\ t,\ stk)$$

In other words, the compiled code executes from s to t if and only if the big-step semantics executes the source code from s to t.

Fig. 8.4. Compiler correctness as two simulations

The two directions of the "$\longleftrightarrow$" are shown diagrammatically in Figure 8.4. The upper level is the big-step execution $(s,\ c) \Rightarrow t$. The lower level is stack machine execution. The relationship between states and configurations is given by $s' = (0,\ s,\ stk)$ and $t' = (size\ (ccomp\ c),\ t,\ stk)$.

Such diagrams should be read as follows: the solid lines and arrows imply the existence of the dashed ones. Thus the left diagram depicts the "$\longrightarrow$" direction (source code can be simulated by compiled code) and the right diagram depicts the "$\longleftarrow$" direction (compiled code can be simulated by source code). We have already seen the analogous simulation relations between big-step and small-step semantics in Section 7.3.1 and will see more such simulations later in the book.

In the case of the compiler, the crucial direction is the simulation of the compiled code by the source code: it tells us that every final state produced by the compiled code is justified by the source code semantics. Therefore we can trust all results produced by the compiled code, if it terminates. But it is still possible that the compiled code does not terminate although the source code does. That can be ruled out by proving also that the compiled code simulates the source code.

We will now prove the two directions of our compiler correctness statement separately. Simulation of the source code by the compiled code is comparatively easy.

Lemma 8.9. *If the source program executes from s to t, so will the compiled program. Formally:*

$$(c,\ s) \Rightarrow t \Longrightarrow ccomp\ c \vdash (0,\ s,\ stk) \rightarrow* (size\ (ccomp\ c),\ t,\ stk)$$

Proof. The proof is by rule induction on the big-step semantics. We go through each of the cases step by step.

SKIP:
: The *SKIP* case translates to the empty machine program and trivially satisfies the correctness statement.

$x ::= a$:
: In the assignment case, we use the correctness of arithmetic expression compilation (Lemma 8.6) together with the composition lemma (Lemma 8.5) for appending the store instruction. We let the prover symbolically execute this store instruction after the expression evaluation to conclude that the state is updated correctly.

$c_1;; c_2$:
: For sequential composition, we get correctness of the two executions for c_1 and c_2 as induction hypothesis, and merely need to lift them into the context of the entire program using our machine execution composition lemmas.

IF b THEN c_1 *ELSE* c_2:
: The two cases for *IF* are solved entirely automatically using correctness of boolean expression compilation (Lemma 8.8), symbolic execution of the jump instruction and the appropriate induction hypothesis for the respective branch.

WHILE b DO c:
: The *False* case of the loop is automatic and follows directly from the correctness lemma for boolean expressions (Lemma 8.8). In the *True* case, we have the two source level executions $(c, s_1) \Rightarrow s_2$ and $(\mathit{WHILE}\ b\ \mathit{DO}\ c, s_2) \Rightarrow s_3$ with corresponding induction hypotheses for the compiled body *ccomp c* started in s_1 and the compiled code of the entire loop started in s_2 ending in s_3. From this we need to construct an execution of the entire loop from s_1 to s_3. We again argue by correctness of boolean expression compilation (Lemma 8.8). We know that the *True* case of that compiled code will end in a program counter pointing to the beginning of the compiled body *ccomp c*. We use the induction hypothesis and the composition lemma (Lemma 8.5) to execute this code into state s_2 with a *pc* pointing to the jump instruction that returns us to the beginning of the loop. Executing this jump instruction symbolically, we get to the machine configuration that lets us use the induction hypothesis that executes the rest of the loop into the desired state s_3.

□

The second direction of the compiler correctness proof is more involved. We have to show that if the machine code executes from state s to t, so does the source code: the source code simulates the compiled code. Formally:

$$ccomp\ c \vdash (0,\ s,\ stk) \rightarrow\!* (size\ (ccomp\ c),\ t,\ stk') \Longrightarrow (c,\ s) \Rightarrow t$$

The main reason this direction is harder to show than the other one is the lack of a suitable structural induction principle that we could apply. Since rule induction on the semantics is not applicable, we have only two further induction principles left in the arsenal we have learned so far: structural induction on the command c or induction on the length of the $\rightarrow\!*$ execution. Neither is strong enough on its own. The solution is to combine them: we will use an outside structural induction on c which will take care of all cases but $WHILE$, and then a nested, inner induction on the length of the execution for the $WHILE$-case.

This idea takes care of the general proof structure. The second problem we encounter is the one of decomposing larger machine executions into smaller ones. Consider the semicolon case. We will have an execution of the form $ccomp\ c_1\ @\ ccomp\ c_2 \vdash cfg \rightarrow\!* cfg'$, and our first induction hypothesis will come with the precondition $ccomp\ c_1 \vdash cfg \rightarrow\!* cfg''$. It may seem intuitive that if $cs_1\ @\ cs_2 \vdash cfg \rightarrow\!* cfg'$ then there must be some intermediate executions $cs_1 \vdash cfg \rightarrow\!* cfg''$ and $cs_2 \vdash cfg'' \rightarrow\!* cfg$, but this is not true in general for arbitrary code sequences cs_1 and cs_2 or configurations cfg and cfg'. For instance, the code may be jumping between cs_1 and cs_2 continuously, and neither execution may make sense in isolation.

However, the code produced from our compiler is particularly well-behaved: execution of compiled code will never jump outside that code and will exit at precisely $pc = size\ (ccomp\ c)$. We merely need to formalise this concept and prove that it is adhered to. This requires a few auxiliary notions.

First we define $isuccs$, the possible successor program counters of a given instruction at position n:

definition $isuccs :: instr \Rightarrow int \Rightarrow int\ set$ **where**
$isuccs\ i\ n = (case\ i\ of$
$\quad JMP\ j \Rightarrow \{n + 1 + j\}\ |$
$\quad JMPLESS\ j \Rightarrow \{n + 1 + j,\ n + 1\}\ |$
$\quad JMPGE\ j \Rightarrow \{n + 1 + j,\ n + 1\}\ |$
$\quad \_ \Rightarrow \{n + 1\})$

Then we define $succs\ P\ n$ which gives us the successor program counters of an instruction sequence P which itself may be embedded in a larger program at position n. The possible successors program counters of an instruction list are the union of all instruction successors:

definition $succs :: instr\ list \Rightarrow int \Rightarrow int\ set$ **where**
$succs\ P\ n = \{s.\ \exists i \geqslant 0.\ i < size\ P \wedge s \in isuccs\ (P\ !!\ i)\ (n + i)\}$

Finally, we remove all jump targets internal to P from $succs\ P\ 0$ and arrive at the possible exit program counters of P. The notation $\{a..<b\}$ stands for

the set of numbers $\geqslant a$ and $< b$. Similarly, $\{a..b\}$ is the set of numbers $\geqslant a$ and $\leqslant b$.

definition *exits* :: *instr list* $\Rightarrow$ *int set* **where**
exits P = *succs* P 0 $-$ {0..<*size* P}

Unsurprisingly, we will need to reason about the successors and exits of composite instruction sequences.

Lemma 8.10 (Successors over append).
succs (*cs* @ *cs'*) n = *succs cs* $n \cup$ *succs cs'* (n + *size cs*)

Proof. The proof is by induction on the instruction list *cs*. To solve each case, we derive the equations for *Nil* and *Cons* in *succs* separately:

succs [] n = {}
succs (x # *xs*) n = *isuccs* x $n \cup$ *succs xs* (1 + n)

□

We could prove a similar lemma about *exits* (*cs* @ *cs'*), but this lemma would have a more complex right-hand side. For the results below it is easier to reason about *succs* first and then just apply the definition of *exits* to the result instead of decomposing *exits* directly.

Before we proceed to reason about decomposing machine executions and compiled code, we note that instead of using the reflexive transitive closure of single step execution, we can equivalently talk about n steps of execution. This will give us a more flexible induction principle and allow us to talk more precisely about sequences of execution steps. We write $P \vdash c \to\hat{}n\ c'$ to mean that the execution of program P starting in configuration c can reach configuration c' in n steps and define:

$P \vdash c \to\hat{}0\ c' = (c' = c)$
$P \vdash c \to\hat{}(Suc\ n)\ c'' = (\exists c'.\ P \vdash c \to c' \wedge P \vdash c' \to\hat{}n\ c'')$

Our old concept of $P \vdash c \to* c'$ is equivalent to saying that there exists an n such that $P \vdash c \to\hat{}n\ c'$.

Lemma 8.11. $(P \vdash c \to* c') = (\exists n.\ P \vdash c \to\hat{}n\ c')$

Proof. One direction is by induction on n, the other by induction on the reflexive transitive closure. □

The more flexible induction principle mentioned above is **complete induction** on n: $(\bigwedge n.\ \forall m{<}n.\ P\ m \Longrightarrow P\ n) \Longrightarrow P\ n$. That is, to show that a property P holds for any n, it suffices to show that it holds for an arbitrary n under the assumption that P already holds for any $m < n$. In our context, this means we are no longer limited to splitting off single execution steps at a time.

We can now derive lemmas about the possible exits for *acomp*, *bcomp*, and *ccomp*. Arithmetic expressions are the easiest, they don't contain any jump instructions, and we only need our append lemma for *succs* (Lemma 8.10).

Lemma 8.12. $exits\ (acomp\ a) = \{size\ (acomp\ a)\}$

Proof. The proof is by first computing all successors of *acomp*, and then deriving the *exits* from that result. For the successors of *acomp*, we show

$$succs\ (acomp\ a)\ n = \{n + 1..n + size\ (acomp\ a)\}$$

by induction on a. The set from $n + 1$ to $n + size\ (acomp\ a)$ is not empty, because we can show $1 \leqslant size\ (acomp\ a)$ by induction on a. □

Compilation for boolean expressions is less well behaved. The main idea is that *bcomp* has two possible *exits*, one for *True*, one for *False*. However, as we have seen in the examples, compiling a boolean expression might lead to the empty list of instructions, which has no successors or exits. More generally, the optimisation that *bcomp* performs may statically exclude one or both of the possible exits. Instead of trying to precisely describe each of these cases, we settle for providing an upper bound on the possible successors of *bcomp*. We are also only interested in positive offsets i.

Lemma 8.13. *If* $0 \leqslant i$, *then*
$exits\ (bcomp\ b\ f\ i) \subseteq \{size\ (bcomp\ b\ f\ i), i + size\ (bcomp\ b\ f\ i)\}$

Proof. Again, we reduce *exits* to *succs*, and first prove

$$\begin{aligned}&0 \leqslant i \Longrightarrow\\ &succs\ (bcomp\ b\ f\ i)\ n\\ &\subseteq \{n..n + size\ (bcomp\ b\ f\ i)\} \cup \{n + i + size\ (bcomp\ b\ f\ i)\}\end{aligned}$$

After induction on b, this proof is mostly automatic. We merely need to instantiate the induction hypothesis for the *And* case manually. □

Finally, we come to the *exits* of *ccomp*. Since we are building on the lemma for *bcomp*, we can again only give an upper bound: there are either no exits, or the exit is precisely $size\ (ccomp\ c)$. As an example that the former case can occur as a result of *ccomp*, consider the compilation of an endless loop

$$ccomp\ (WHILE\ Bc\ True\ DO\ SKIP) = [JMP\ -1]$$

That is, we get one jump instruction that jumps to itself and

$$exits\ [JMP\ -1] = \{\}$$

Lemma 8.14. $exits\ (ccomp\ c) \subseteq \{size\ (ccomp\ c)\}$

Proof. We first reduce the lemma to *succs*:

$$succs\ (ccomp\ c)\ n \subseteq \{n..n + size\ (ccomp\ c)\}$$

This proof is by induction on c and the corresponding *succs* results for *acomp* and *bcomp*. □

We have derived these *exits* results about *acomp*, *bcomp* and *ccomp* because we wanted to show that the machine code produced by these functions is well behaved enough that larger executions can be decomposed into smaller separate parts. The main lemma that describes this decomposition is somewhat technical. It states that, given an n-step execution of machine instructions cs that are embedded in a larger program $(P\ @\ cs\ @\ P')$, we can find a k-step sub-execution of cs in isolation, such that this sub-execution ends at one of the exit program counters of cs, and such that it can be continued to end in the same state as the original execution of the larger program. For this to be true, the initial pc of the original execution must point to somewhere within cs, and the final pc' to somewhere outside cs. Formally, we get the following.

Lemma 8.15 (Decomposition of machine executions).

$$\begin{array}{l}
[\![P\ @\ cs\ @\ P' \vdash (size\ P + pc,\ stk,\ s) \rightarrow\hat{}n\ (pc',\ stk',\ s');\\
\ pc \in \{0..<size\ cs\};\ pc' \notin \{size\ P..<size\ P + size\ cs\}]\!]\\
\Longrightarrow \exists pc''\ stk''\ s''\ k\ m.\\
\qquad cs \vdash (pc,\ stk,\ s) \rightarrow\hat{}k\ (pc'',\ stk'',\ s'') \wedge\\
\qquad pc'' \in exits\ cs \wedge\\
\qquad P\ @\ cs\ @\ P' \vdash (size\ P + pc'',\ stk'',\ s'') \rightarrow\hat{}m\ (pc',\ stk',\ s') \wedge\\
\qquad n = k + m
\end{array}$$

Proof. The proof is by induction on n. The base case is trivial, and the step case essentially reduces the lemma to a similar property for a single execution step:

$$\begin{array}{l}
[\![P\ @\ cs\ @\ P' \vdash (size\ P + pc,\ stk,\ s) \rightarrow (pc',\ stk',\ s');\\
\ pc \in \{0..<size\ cs\}]\!]\\
\Longrightarrow cs \vdash (pc,\ stk,\ s) \rightarrow (pc' - size\ P,\ stk',\ s')
\end{array}$$

This property is proved automatically after unfolding the definition of single step execution and case distinction on the instruction to be executed.

Considering the step case for $n + 1$ execution steps in our induction again, we note that this step case consists of one single step in the larger context and the n-step rest of the execution, also in the larger context. Additionally we have the induction hypothesis which gives us our property for executions of length n.

Using the property above, we can reduce the single step execution to just cs. To connect this up with the rest of the execution, we make use of the

induction hypothesis in the case where the execution of *cs* is not at an exit yet, which means the *pc* is still inside *cs*, or we observe that the execution has left *cs* and the *pc* must therefore be at an exit of of *cs*. In this case k is 1 and $m = n$, and we can just append the rest of the execution from our assumptions. □

The accompanying Isabelle theories contain a number of more convenient instantiations of this lemma. We omit these here, and directly move on to proving the correctness of *acomp*, *bcomp*, *ccomp*.

As always, arithmetic expressions are the least complex case.

Lemma 8.16 (Correctness of *acomp*).

$$acomp\ a \vdash (0,\ s,\ stk) \rightarrow\hat{}\ n\ (size\ (acomp\ a),\ s',\ stk') \Longrightarrow s' = s \wedge stk' = aval\ a\ s\ \#\ stk$$

Proof. The proof is by induction on the expression, and most cases are solved automatically, symbolically executing the compilation and resulting machine code sequence. In the *Plus*-case, we decompose the execution manually using Lemma 8.15, apply the induction hypothesis for the parts, and combine the results symbolically executing the *ADD* instruction. □

The next step are boolean expressions. Correctness here is mostly about the pc' at the end of the expression evaluation. Stack and state remain unchanged.

Lemma 8.17 (Correctness of *bcomp*).

$$\begin{aligned}&[\![bcomp\ b\ f\ j \vdash (0,\ s,\ stk) \rightarrow\hat{}\ n\ (pc',\ s',\ stk');\\ &\ size\ (bcomp\ b\ f\ j) \leqslant pc';\ 0 \leqslant j]\!]\\ &\Longrightarrow pc' = size\ (bcomp\ b\ f\ j) + (if\ f = bval\ b\ s\ then\ j\ else\ 0) \wedge\\ &\quad s' = s \wedge stk' = stk\end{aligned}$$

Proof. The proof is by induction on the expression. The *And* case is the only interesting one. We first split the execution into one for the left operand b_1 and one for the right operand b_2 with a suitable instantiation of our splitting lemma above (Lemma 8.15). We then determine by induction hypothesis that stack and state did not change for b_1 and that the program counter will either exit directly, in which case we are done, or it will point to the instruction sequence of b_2, in which case we apply the second induction hypothesis to conclude the case and the lemma. □

We are now ready to tackle the main lemma for *ccomp*.

Lemma 8.18 (Correctness of *ccomp*).

$$ccomp\ c \vdash (0,\ s,\ stk) \to\hat{}n\ (size\ (ccomp\ c),\ t,\ stk') \Longrightarrow (c,\ s) \Rightarrow t \wedge stk' = stk$$

Proof. As mentioned before, the main proof is an induction on c, and for the *WHILE* case a nested complete induction on the length of the execution. The cases of the structural induction are the following.

SKIP:
: This case is easy and automatic.

$x ::= a$:
: The assignment case makes use of the correct compilation of arithmetic expressions (Lemma 8.16) and is otherwise automatic.

$c_1;;\ c_2$:
: This case comes down to using Lemma 8.15 and combining the induction hypotheses as usual.

IF b THEN c_1 *ELSE* c_2:
: The *IF* case is more interesting. Let I stand for the whole *IF* expression. We start by noting that we need to prove

$$ccomp\ I \vdash (0,\ s,\ stk) \to\hat{}n\ (size\ (ccomp\ I),\ t,\ stk') \Longrightarrow (I,\ s) \Rightarrow t \wedge stk' = stk$$

and that we have the same property available for $ccomp\ c_1$ and $ccomp\ c_2$ as induction hypotheses. After splitting off the execution of the boolean expression using Lemma 8.17 and Lemma 8.15, we known

$$ccomp\ c_1\ @\ JMP\ (size\ (ccomp\ c_2))\ \#\ ccomp\ c_2 \vdash (if\ bval\ b\ s\ then\ 0\ else\ size\ (ccomp\ c_1) + 1,\ s,\ stk) \to\hat{}m (1 + size\ (ccomp\ c_1) + size\ (ccomp\ c_2),\ t,\ stk')$$

We proceed by case distinction on $bval\ b\ s$, and use the corresponding induction hypothesis for c_1 and c_2 respectively to conclude the *IF* case.

WHILE b DO c:
: In the *WHILE* case, let w stand for the loop *WHILE b DO c*, and cs for the compiled loop $ccomp\ w$. Our induction hypothesis is

$$ccomp\ c \vdash (0,\ s,\ stk) \to\hat{}n\ (size\ (ccomp\ c),\ t,\ stk') \Longrightarrow (c,\ s) \Rightarrow t \wedge stk' = stk$$

for any s, stk, n, t, and stk'. We need to show

$$cs \vdash (0,\ s,\ stk) \to\hat{}n\ (size\ cs,\ t,\ stk') \Longrightarrow (w,\ s) \Rightarrow t \wedge stk' = stk$$

As mentioned, the induction hypothesis above is not strong enough to conclude the goal. Instead, we continue the proof with complete induction on n, that is, we still need to show the same goal, for a new arbitrary n, but we get the following additional induction hypothesis for an arbitrary s.

$$\forall m{<}n.\ cs \vdash (0,\ s,\ stk) \to\hat{}m\ (size\ cs,\ t,\ stk') \longrightarrow (w,\ s) \Rightarrow t \wedge stk' = stk$$

We can now start decomposing the machine code of the *WHILE* loop. Recall the definition of compilation for *WHILE*:

ccomp (*WHILE b DO c*) =
(*let cc* = *ccomp c*; *cb* = *bcomp b False* (*size cc* + 1)
in cb @ *cc* @ [*JMP* (− (*size cb* + *size cc* + 1))])

As in the *IF* case, we start with splitting off the execution of the boolean expression, and a case distinction on its result *bval b s*. The *False* case is easy, it directly jumps to the exit and we can conclude our goal. In the *True* case, we drop into the execution of *ccomp c*. Here, we first need to consider whether *ccomp c* = [] or not, because our decomposition lemma only applies when the program counter points *into* the code of *ccomp c*, which is not possible when that is empty (e.g. compiled from *SKIP*). If it is empty, the only thing left to do in the *WHILE* loop is to jump back to the beginning. After executing that instruction symbolically, we note that we are now in a situation where our induction hypothesis applies: we have executed at least one instruction (the jump), so m is less than n, and we can directly conclude $(w, s) \Rightarrow t \wedge stk' = stk$. In this specific situation, we could also try to prove that the loop will never terminate and that therefore our assumption that machine execution terminates is *False*, but it is easier to just apply the induction hypothesis here.

The other case, where *ccomp c* $\neq$ [], lets us apply the decomposition lemma to isolate the execution of *ccomp c* to some intermediate (s'', stk'') with $pc'' = size\ (ccomp\ c)$. With our induction hypothesis about c from the outer induction, we conclude $(c, s) \Rightarrow s''$ and $stk'' = stk$.

Symbolically executing the final jump instruction transports us to the beginning of the compiled code again, into a situation where the inner induction hypothesis applies, because we have again executed at least one instruction, so the number of remaining execution steps m is less than the original n. That is, we can conclude $(w, s'') \Rightarrow t \wedge stk' = stk$. Together with *bval b s* and $(c, s) \Rightarrow s''$, we arrive at $(w, s) \Rightarrow t \wedge stk' = stk$, which is what we needed to show.

□

Combining the above lemma with the first direction, we get our full compiler correctness theorem.

Theorem 8.19 (Compiler Correctness).
ccomp c $\vdash (0, s, stk) \rightarrow* (size\ (ccomp\ c), t, stk) \longleftrightarrow (c, s) \Rightarrow t$

Proof. Follows directly from Lemma 8.18, Lemma 8.9, and Lemma 8.11. □

It is worth pointing out that in a deterministic language like IMP, this second direction reduces to preserving termination: if the machine program

terminates, so must the source program. If that was the case, we could conclude that started in s the source must terminate in some t'. Using the first direction of the compiler proof and by determinism of the machine language, we could then conclude that $t' = t$ and that therefore the source execution was already the right one. However, showing that machine-level termination implies source-level termination is not much easier than showing the second direction of our compiler proof directly, and so we did not take this path here.

8.5 Summary and Further Reading

This section has shown the correctness of a simple, non-optimising compiler from IMP to an idealised machine language. The main technical challenge was reasoning about machine code sequences, as well as their composition and decomposition. Apart from this technical hurdle, formally proving the correctness of such a simple compiler is not as difficult as one might initially suspect.

Two related compiler correctness proofs in the literature are the compiler in the Java-like language Jinja [49] in a style that is similar to the one presented here, and more recently, the fully realistic, optimising C compiler CompCert by Leroy et al [55] which compiles to PowerPC, x86, and ARM architectures. Both proofs are naturally more complex than the one presented here, both working with the concept of intermediate languages and multiple compilation stages. This is done to simplify the argument and to concentrate on specific issues on each level.

Modelling the program as a list of instructions is another abstraction we introduce to a real low-level machine. A real CPU would implement a von-Neumann machine, which adds fetching and decoding of instructions to the execution cycle. The main difference is that our model does not admit self-modifying programs which is not necessary for IMP. It is entirely possible to model low-level machine code in a theorem prover. The CompCert project has done this as part of its compiler correctness statement, but there are also other independent machine language models available in the literature, for instance a very detailed and well-validated model of multiple ARM processor versions from Cambridge [32, 33], and a similarly detailed, but less complete model of the Intel x86 instruction set architecture by Morrisett et al [59].

For our compiler correctness, we presented a proof of semantics preservation in both directions: from source to machine and from machine to source. Jinja only presents one direction, CompCert does both, but uses a different style of argument for the more involved direction from machine to source, which involves interpreting the semantics co-inductively to include reasoning about non-terminating programs directly.

Exercises

Exercise 8.20. Modify the machine language such that instead of variable names to values, the machine state maps addresses (integers) to values. Adjust the compiler proof accordingly.

9

Types

This chapter introduces types to 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* [58]. 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 9.1, and information leakage in Section 9.2.

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 NullPointer 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 to all automatic semantic analyses and is discussed in more detail at the beginning of the next chapter.

9.1 Typed IMP thy

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 $Plus\ (Ic\ 1)$

$$\frac{}{taval\ (Ic\ i)\ s\ (Iv\ i)} \qquad \frac{}{taval\ (Rc\ r)\ s\ (Rv\ r)} \qquad \frac{}{taval\ (V\ x)\ s\ (s\ x)}$$

$$\frac{taval\ a_1\ s\ (Iv\ i_1) \qquad taval\ a_2\ s\ (Iv\ i_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Iv\ (i_1 + i_2))}$$

$$\frac{taval\ a_1\ s\ (Rv\ r_1) \qquad taval\ a_2\ s\ (Rv\ r_2)}{taval\ (Plus\ a_1\ a_2)\ s\ (Rv\ (r_1 + r_2))}$$

Fig. 9.1. Inductive definition of $taval :: aexp \Rightarrow state \Rightarrow val \Rightarrow bool$

$(Rc\ 3)$ 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 introduced in Section 2.3.1 to denote the undefined cases. It is quite possible to do so, and in later chapters we will demonstrate that variant. However, it implies that 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 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 9.1. 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 $s\ ''x'' = Iv\ 4$.

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

$$\frac{}{tbval\ (Bc\ v)\ s\ v} \qquad \frac{tbval\ b\ s\ bv}{tbval\ (Not\ b)\ s\ (\neg\ bv)}$$

$$\frac{tbval\ b_1\ s\ bv_1 \qquad tbval\ b_2\ s\ bv_2}{tbval\ (And\ b_1\ b_2)\ s\ (bv_1 \wedge bv_2)} \qquad \frac{taval\ a_1\ s\ (Iv\ i_1) \qquad taval\ a_2\ s\ (Iv\ i_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (i_1 < i_2)}$$

$$\frac{taval\ a_1\ s\ (Rv\ r_1) \qquad taval\ a_2\ s\ (Rv\ r_2)}{tbval\ (Less\ a_1\ a_2)\ s\ (r_1 < r_2)}$$

Fig. 9.2. Inductive definition of $tbval :: bexp \Rightarrow state \Rightarrow bool \Rightarrow bool$

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 9.2 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 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 Chapter 10, 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 9.3, it merely refers to the new judgements for arithmetic and boolean expressions, but does not add any new rules on its own.

$$\frac{taval\ a\ s\ v}{(x ::= a,\ s) \to (SKIP,\ s(x := v))}$$

$$\frac{}{(SKIP;;\ c,\ s) \to (c,\ s)} \qquad \frac{(c_1,\ s) \to (c1',\ s')}{(c_1;;\ c_2,\ s) \to (c1';;\ c_2,\ s')}$$

$$\frac{tbval\ b\ s\ True}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_1,\ s)}$$

$$\frac{tbval\ b\ s\ False}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_2,\ s)}$$

$$\frac{}{(WHILE\ b\ DO\ c,\ s) \to (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,\ s)}$$

Fig. 9.3. Inductive definition of $op \to :: com \times state \Rightarrow com \times state \Rightarrow bool$

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

Example 9.1. For well-behaved programs, our typed executions look as before. For instance, let s satisfy $s\ ''y'' = Iv\ 7$. Then we get the following example execution chain.

$(''x'' ::= V\ ''y'';;\ ''y'' ::= Plus\ (V\ ''x'')\ (V\ ''y''),\ s) \to$
$(''y'' ::= Plus\ (V\ ''x'')\ (V\ ''y''),\ s(''x'' := Iv\ 7)) \to$
$(SKIP,\ s(''x'' := Iv\ 7,\ ''y'' := Iv\ 14))$

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:

$(''x'' ::= V\ ''y'';;\ ''y'' ::= Plus\ (V\ ''x'')\ (Rc\ 3),\ s) \to$
$(''y'' ::= Plus\ (V\ ''x'')\ (Rc\ 3),\ s(''x'' := Iv\ 7))$

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.

9.1.1 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.

$$\frac{}{\Gamma \vdash Ic\ i : Ity} \qquad \frac{}{\Gamma \vdash Rc\ r : Rty} \qquad \frac{}{\Gamma \vdash V\ x : \Gamma\ x}$$

$$\frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Plus\ a_1\ a_2 : \tau}$$

Fig. 9.4. Inductive definition of $\_ \vdash \_ : \_ :: tyenv \Rightarrow aexp \Rightarrow ty \Rightarrow bool$

$$\frac{}{\Gamma \vdash Bc\ v} \qquad \frac{\Gamma \vdash b}{\Gamma \vdash Not\ b}$$

$$\frac{\Gamma \vdash b_1 \qquad \Gamma \vdash b_2}{\Gamma \vdash And\ b_1\ b_2} \qquad \frac{\Gamma \vdash a_1 : \tau \qquad \Gamma \vdash a_2 : \tau}{\Gamma \vdash Less\ a_1\ a_2}$$

Fig. 9.5. Inductive definition of $op \vdash :: tyenv \Rightarrow bexp \Rightarrow bool$

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 $ty = Ity \mid Rty$

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 9.4 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 9.5 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

$$\frac{}{\Gamma \vdash \mathit{SKIP}} \qquad \frac{\Gamma \vdash a : \Gamma\ x}{\Gamma \vdash x ::= a}$$

$$\frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1;;\ c_2} \qquad \frac{\Gamma \vdash b \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2} \qquad \frac{\Gamma \vdash b \quad \Gamma \vdash c}{\Gamma \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c}$$

Fig. 9.6. 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 wether 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 $\Gamma\ ''x'' = \mathit{Ity}$ as well as $\Gamma\ ''y'' = \mathit{Ity}$. Then our previous example program is well-typed, because of the following type derivation.

$$\dfrac{\dfrac{\dfrac{\Gamma\ ''y'' = \mathit{Ity}}{\Gamma \vdash V\ ''y'' : \mathit{Ity}}}{\Gamma \vdash ''x'' ::= V\ ''y''} \qquad \dfrac{\dfrac{\Gamma\ ''x'' = \mathit{Ity}}{\Gamma \vdash V\ ''x'' : \mathit{Ity}} \qquad \dfrac{\Gamma\ ''y'' = \mathit{Ity}}{\Gamma \vdash V\ ''y'' : \mathit{Ity}}}{\Gamma \vdash \mathit{Plus}\ (V\ ''x'')\ (V\ ''y'') : \mathit{Ity}}}{\Gamma \vdash ''y'' ::= \mathit{Plus}\ (V\ ''x'')\ (V\ ''y'')}}{\Gamma \vdash ''x'' ::= V\ ''y'';;\ ''y'' ::= \mathit{Plus}\ (V\ ''x'')\ (V\ ''y'')}$$

9.1.2 Well-typed Programs do Not Get Stuck

In this section we prove that the type system defined above is sound. As mentioned in the introduction to this chapter, Robert Milner coined the phrase

well-typed programs cannot go wrong [58], 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:

$$[\![(c,\ s)\ {\to}*\ (c',\ s');\ \Gamma \vdash c]\!] \Longrightarrow c' = \mathit{SKIP} \vee (\exists\, cs''.\ (c',\ s') \to cs'')$$

Given an arbitrary command c, which is well-typed $\Gamma \vdash c$, any execution $(c,\ s)\ {\to}*\ (c',\ s')$ either has terminated successfully with $c' = \mathit{SKIP}$, or can make another execution step $\exists\, cs''.\ (c',\ s') \to cs''$. 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

$$[\![(c,\ s)\ {\to}*\ (c',\ s');\ \Gamma \vdash c;\ \Gamma \vdash s;\ c' \neq \mathit{SKIP}]\!] \Longrightarrow \exists\, cs''.\ (c',\ s') \to cs''$$

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* ⇒ *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* ⊢ :: *tyenv* ⇒ *state* ⇒ *bool* **where**
Γ ⊢ *s* ⟷ (∀ *x*. *type* (*s x*) = Γ *x*)

We now have everything defined to start the soundness proof. The plan is to prove progress and preservation, 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 9.2 (Preservation for arithmetic expressions).
⟦Γ ⊢ *a* : τ; *taval a s v*; Γ ⊢ *s*⟧ ⟹ *type v* = τ

Proof. The proof is by rule induction on the type derivation Γ ⊢ *a* : τ. If we declare rule inversion on *taval* to be used automatically and unfold the definition of *styping*, Isabelle proves the rest. □

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 9.3 (Progress for arithmetic expressions).
⟦Γ ⊢ *a* : τ; Γ ⊢ *s*⟧ ⟹ ∃ *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. □

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

Lemma 9.4 (Progress for boolean expressions).
$[\![\Gamma \vdash b; \Gamma \vdash s]\!] \Longrightarrow \exists v.\ tbval\ b\ s\ v$

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. □

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 9.5 (Preservation: commands stay well-typed).
$[\![(c, s) \to (c', s'); \Gamma \vdash c]\!] \Longrightarrow \Gamma \vdash c'$

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. □

Theorem 9.6 (Preservation: states stay well-typed).
$[\![(c,\ s) \to (c',\ s');\ \Gamma \vdash c;\ \Gamma \vdash s]\!] \Longrightarrow \Gamma \vdash s'$

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. □

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 9.7 (Progress for commands).
$[\![\Gamma \vdash c;\ \Gamma \vdash s;\ c \neq SKIP]\!] \Longrightarrow \exists\, cs'.\ (c,\ s) \to cs'$

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*. □

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 9.8 (Type soundness).
$[\![(c,\ s) \to* (c',\ s');\ \Gamma \vdash c;\ \Gamma \vdash s;\ c' \neq SKIP]\!] \Longrightarrow \exists\, cs''.\ (c',\ s') \to cs''$

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. □

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.

9.1.3 Exercises

Exercise 9.9. Reformulate the inductive predicates $\Gamma \vdash a : \tau$, $\Gamma \vdash b$ and $\Gamma \vdash c$ as three functions $atype :: tyenv \Rightarrow aexp \Rightarrow ty\ option$, $bok :: tyenv \Rightarrow bexp \Rightarrow bool$ and $cok :: tyenv \Rightarrow com \Rightarrow bool$ and prove the three equivalences $(\Gamma \vdash a : \tau) = (atype\ \Gamma\ a = Some\ \tau)$, $(\Gamma \vdash b) = bok\ \Gamma\ b$ and $(\Gamma \vdash c) = cok\ \Gamma\ c$.

9.2 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.

9.2.1 Security Levels and Expressions thy

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 $sec :: aexp \Rightarrow level$ **where**
$sec\ (N\ n) = 0$
$sec\ (V\ x) = sec\ x$
$sec\ (Plus\ a_1\ a_2) = max\ (sec\ a_1)\ (sec\ a_2)$

fun $sec :: bexp \Rightarrow level$ **where**
$sec\ (Bc\ v) = 0$
$sec\ (Not\ b) = sec\ b$
$sec\ (And\ b_1\ b_2) = max\ (sec\ b_1)\ (sec\ b_2)$
$sec\ (Less\ a_1\ a_2) = max\ (sec\ a_1)\ (sec\ a_2)$

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 lightweight 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 $\leqslant$, but also the strict $<$ later on, so we define both here:

$$\frac{}{l \vdash SKIP} \qquad \frac{sec\ a \leqslant sec\ x \qquad l \leqslant sec\ x}{l \vdash x ::= a} \qquad \frac{l \vdash c_1 \qquad l \vdash c_2}{l \vdash c_1;;\ c_2}$$

$$\frac{max\ (sec\ b)\ l \vdash c_1 \qquad max\ (sec\ b)\ l \vdash c_2}{l \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2} \qquad \frac{max\ (sec\ b)\ l \vdash c}{l \vdash WHILE\ b\ DO\ c}$$

Fig. 9.7. Definition of *sec_type* :: $nat \Rightarrow com \Rightarrow bool$

$$s = s'\ (\leqslant l) \equiv \forall x.\ sec\ x \leqslant l \longrightarrow s\ x = s'\ x$$
$$s = s'\ (< l) \equiv \forall x.\ sec\ x < l \longrightarrow s\ x = s'\ x$$

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 $sec\ e$.

Lemma 9.10 (Noninterference for arithmetic expressions).
$[\![s_1 = s_2\ (\leqslant l);\ sec\ e \leqslant l]\!] \Longrightarrow aval\ e\ s_1 = aval\ e\ s_2$

Lemma 9.11 (Noninterference for boolean expressions).
$[\![s_1 = s_2\ (\leqslant l);\ sec\ b \leqslant l]\!] \Longrightarrow bval\ b\ s_1 = bval\ b\ s_2$

9.2.2 Syntax Directed Typing thy

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 $\geqslant l$.

Going through the rules of Figure 9.7 in detail, we have defined *SKIP* to be safe at any level. We have have defined assignment to be safe if the level of x is higher than or equal to the level of the information source a, but lower than or equal to l. For semicolon to conform to a level l, we just recursively demand that both parts conform to the same level l. As previously shown in the motivating example, the *IF* 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 9.12. Testing our intuition about what we have just defined, we look at four examples for various security levels.

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

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?

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

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 $2 \vdash ''x'' ::= N\ 0$. 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?

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

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?

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

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 [90], has been criticised as too restrictive. It excludes too many safe programs. This can be addressed by making the type system more refined, more flexible, and more context aware. For demonstrating the type system and its soundness proof in this book, however, we will stick to its simplest form.

9.2.3 Soundness

We introduced the correctness statement for this type system as noninterference: two executions of the same program started in related states end up in related states. The relation in our case is that the values of variables below security level l are the same. Formally, this is the following statement

$$[\![(c,\ s) \Rightarrow s';\ (c,\ t) \Rightarrow t';\ 0 \vdash c;\ s = t\ (\leqslant l)]\!] \Longrightarrow s' = t'\ (\leqslant l)$$

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 $l' \leqslant l$, then we are only admitting more flows, i.e. we are making a weaker statement.

Lemma 9.13 (Anti-monotonicity). $[\![l \vdash c;\ l' \leqslant l]\!] \Longrightarrow l' \vdash c$

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

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 $\geqslant l$.

Lemma 9.14 (Confinement). $[\![(c,\ s) \Rightarrow t;\ l \vdash c]\!] \Longrightarrow s = t\ (< l)$

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 $max\ (sec\ b)\ l \vdash c_1$. Since $l \leqslant max\ (sec\ b)\ l$, anti-monotonicity allows us to conclude $l \vdash c_1$. □

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

Theorem 9.15 (Noninterference).
$[\![(c, s) \Rightarrow s'; (c, t) \Rightarrow t'; 0 \vdash c; s = t\ (\leqslant l)]\!] \Longrightarrow s' = t'\ (\leqslant l)$

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 $s(x := aval\ a\ s)$ in the first big-step execution. We perform rule inversion for the second execution to get the same update for t. We also perform rule inversion on the typing statement to get the relationship between security levels of x and a: $sec\ a \leqslant sec\ x$. 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 $sec\ x < l$, and that all other variables y with $sec\ y < l$ still agree as before. In the first case, looking at x, we know from above that $sec\ a \leqslant sec\ x$. Hence, by transitivity, we have that $sec\ a \leqslant l$. This is enough for our noninterference result on expressions to apply, given that we also know $s = t\ (\leqslant l)$ from the premises. This means, we get $aval\ a\ s = aval\ a\ t$: the new values for x agree as required. The case for all other variables y below l follows directly from $s = t\ (\leqslant l)$.

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 $sec\ b$, since the maximum with 0 is the identity, i.e. we know $sec\ b \vdash c_1$. Then we perform a case distinction: either the level of b is $\leqslant l$ or it is not. If $sec\ b \leqslant l$, i.e. the *IF* decision is on a more public level than l, then s and t, which agree below l, also agree below $sec\ b$. That in turn means by our noninterference lemma for expressions that they evaluate to the same result, so $bval\ b\ s = True$ and $bval\ b\ t = True$. We already noted $sec\ b \vdash c_1$ 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 $l < sec\ b$, 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 $sec\ b \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2$ since we know both $max\ (sec\ b)\ 0 \vdash c_1$ and $max\ (sec\ b)\ 0 \vdash c_2$. This in

$$\frac{}{l \vdash' \mathit{SKIP}} \qquad \frac{\mathit{sec}\ a \leqslant \mathit{sec}\ x \qquad l \leqslant \mathit{sec}\ x}{l \vdash' x ::= a}$$

$$\frac{l \vdash' c_1 \qquad l \vdash' c_2}{l \vdash' c_1;; c_2} \qquad \frac{\mathit{sec}\ b \leqslant l \qquad l \vdash' c_1 \qquad l \vdash' c_2}{l \vdash' \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2}$$

$$\frac{\mathit{sec}\ b \leqslant l \qquad l \vdash' c}{l \vdash' \mathit{WHILE}\ b\ \mathit{DO}\ c} \qquad \frac{l \vdash' c \qquad l' \leqslant l}{l' \vdash' c}$$

Fig. 9.8. Definition of $\mathit{sec\_type}' :: \mathit{nat} \Rightarrow \mathit{com} \Rightarrow \mathit{bool}$

turn means we now can apply confinement: everything below $\mathit{sec}\ b$ 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 $s = t\ (\leqslant l)$, we can conclude $s' = t'\ (\leqslant l)$ 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. □

9.2.4 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 9.8 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 9.16. $l \vdash c \implies l \vdash' c$

Lemma 9.17. $l \vdash' c \implies l \vdash c$

9.2.5 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 9.9.

$$\frac{}{\vdash SKIP : l} \qquad \frac{sec\ a \leqslant sec\ x}{\vdash x ::= a : sec\ x} \qquad \frac{\vdash c_1 : l_1 \quad \vdash c_2 : l_2}{\vdash c_1;;\ c_2 : min\ l_1\ l_2}$$

$$\frac{sec\ b \leqslant min\ l_1\ l_2 \quad \vdash c_1 : l_1 \quad \vdash c_2 : l_2}{\vdash IF\ b\ THEN\ c_1\ ELSE\ c_2 : min\ l_1\ l_2} \qquad \frac{sec\ b \leqslant l \quad \vdash c : l}{\vdash WHILE\ b\ DO\ c : l}$$

Fig. 9.9. Definition of the bottom-up security type system.

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 9.18. $\vdash c : l \implies l \vdash' c$

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

$0 \vdash'\ ''x'' ::= N\ 0$, 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 9.19. $l \vdash' c \implies \exists l' \geqslant l.\ \vdash c : l'$

9.2.6 What about termination? thy

In the previous section we proved the following security theorem (Theorem 9.15):

$$[\![(c, s) \Rightarrow s';\ (c, t) \Rightarrow t';\ 0 \vdash c;\ s = t\ (\leqslant l)]\!] \Longrightarrow s' = t'\ (\leqslant l)$$

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.

$0 \vdash WHILE\ Less\ (V\ ''x'')\ (N\ 1)\ DO\ SKIP$

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 [16].

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:

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

This is already it. Figure 9.10 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

$$\frac{}{l \vdash SKIP} \qquad \frac{sec\ a \leqslant sec\ x \qquad l \leqslant sec\ x}{l \vdash x ::= a} \qquad \frac{l \vdash c_1 \qquad l \vdash c_2}{l \vdash c_1;;\ c_2}$$

$$\frac{max\ (sec\ b)\ l \vdash c_1 \qquad max\ (sec\ b)\ l \vdash c_2}{l \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2} \qquad \frac{sec\ b = 0 \qquad 0 \vdash c}{0 \vdash WHILE\ b\ DO\ c}$$

Fig. 9.10. Termination-sensitive security type system.

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.

$$[\![(c,\ s) \Rightarrow s';\ 0 \vdash c;\ s = t\ (\leqslant l)]\!] \Longrightarrow \exists t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ (\leqslant l)$$

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 9.20 (Anti-monotonicity).
$[\![l \vdash c;\ l' \leqslant l]\!] \Longrightarrow l' \vdash c$

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

Our confinement lemma is also still true.

Lemma 9.21 (Confinement).
$[\![(c,\ s) \Rightarrow t;\ l \vdash c]\!] \Longrightarrow s = t\ (< l)$

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. □

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 9.22 (Termination).
$[\![l \vdash c;\ l \neq 0]\!] \Longrightarrow \exists t.\ (c,\ s) \Rightarrow t$

[1] Note 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 $0 \neq 0$. □

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

Theorem 9.23 (Noninterference).
$[\![(c, s) \Rightarrow s'; 0 \vdash c; s = t\ (\leqslant l)]\!] \Longrightarrow \exists t'.\ (c, t) \Rightarrow t' \wedge s' = t'\ (\leqslant l)$

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. $\neg\ sec\ b \leqslant l$, 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 $sec\ b$ 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 $bval\ b\ s$, $(c, s) \Rightarrow s''$, 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 $bval\ b\ t$ 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. □

$$\frac{}{l \vdash' \mathit{SKIP}} \qquad \frac{\mathit{sec}\ a \leqslant \mathit{sec}\ x \qquad l \leqslant \mathit{sec}\ x}{l \vdash' x ::= a} \qquad \frac{l \vdash' c_1 \qquad l \vdash' c_2}{l \vdash' c_1;; c_2}$$

$$\frac{\mathit{sec}\ b \leqslant l \qquad l \vdash' c_1 \qquad l \vdash' c_2}{l \vdash' \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2} \qquad \frac{\mathit{sec}\ b = 0 \qquad 0 \vdash' c}{0 \vdash' \mathit{WHILE}\ b\ \mathit{DO}\ c}$$

$$\frac{l \vdash' c \qquad l' \leqslant l}{l' \vdash' c}$$

Fig. 9.11. Termination-sensitive security type system — standard formulation.

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 9.11 introduces the standard system.

As before, we can show equivalence with our formulation.

Lemma 9.24 (Equivalence to standard formulation).
$l \vdash c \longleftrightarrow l \vdash' c$

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. □

9.2.7 Exercises

Exercise 9.25. Reformulate the inductive predicate *sec_type* defined in Figure 9.7 as a recursive function $ok :: \mathit{level} \Rightarrow \mathit{com} \Rightarrow \mathit{bool}$ and prove the equivalence of the two formulations.

Try to reformulate the bottom-up system from Figure 9.9 as a function that computes the security level of a command. What difficulty do you face?

Exercise 9.26. Define a bottom-up termination insensitive security type system $\vdash' c : l$ with subsumption rule. Prove equivalence with the bottom-up system in Figure 9.9.

9.3 Summary and Further Reading

In this chapter 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 [56]. 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 [77]. As mentioned, the type system and the soundness statement in the sections above go back to Volpano, Irvine, and Smith [90], and the termination-sensitive analysis to Volpano and Smith [91]. 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. Our additional bottom-up type system can be seen as a simplified description of type inference for security types. Volpano and Smith [92] gave an explicit algorithm for type inference and showed that most general types exist if the program is type correct. We also mentioned that our simple security levels based on natural numbers can be generalised to arbitrary lattices. This observation goes back to Denning [27].

A popular alternative to security type systems is dynamic tracking of information flows, or so-called **taint analysis** [82]. 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 [78]. 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 [76].

The name non-interference was coined by Goguen and Meseguer [36], but the property goes back further to Ellis Cohen who called its inverse *Strong Dependency* [19]. The concept of covert information flow channels already precedes this idea [53]. 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 [62].

10

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 chapter we used type systems for error detection. In this chapter we employ what is known as *data-flow analysis*. We study three 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`.
3. Live variable analysis determines if a variable is "live" at some point, i.e. if its value can influence the subsequent execution. Assignments to variables that are not live can be eliminated: for example, the first assignment in the sequence `x := 0; x := 1` is redundant.

Throughout this chapter 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 nontrivial semantic property P has a magic analyser that

- terminates on every input program,
- only says Yes if the input program has property P (correctness), and
- 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 ∧ x^n + y^n = z^n 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.

10.1 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. [39, 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 10.2, 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.

10.1.1 Definite Initialisation thy

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

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

Similar statements exist for each 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:

$$\frac{}{D\ A\ SKIP\ A} \qquad \frac{vars\ a \subseteq A}{D\ A\ (x ::= a)\ (insert\ x\ A)}$$

$$\frac{D\ A_1\ c_1\ A_2 \qquad D\ A_2\ c_2\ A_3}{D\ A_1\ (c_1;;\ c_2)\ A_3}$$

$$\frac{vars\ b \subseteq A \qquad D\ A\ c_1\ A_1 \qquad D\ A\ c_2\ A_2}{D\ A\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ (A_1 \cap A_2)}$$

$$\frac{vars\ b \subseteq A \qquad D\ A\ c\ A'}{D\ A\ (WHILE\ b\ DO\ c)\ A}$$

Fig. 10.1. Definite initialisation $D :: vname\ set \Rightarrow com \Rightarrow vname\ set \Rightarrow bool$

fun $vars :: aexp \Rightarrow vname\ set$ **where**
$vars\ (N\ n) = \{\}$
$vars\ (V\ x) = \{x\}$
$vars\ (Plus\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

fun $vars :: bexp \Rightarrow vname\ set$ **where**
$vars\ (Bc\ v) = \{\}$
$vars\ (Not\ b) = vars\ b$
$vars\ (And\ b_1\ b_2) = vars\ b_1 \cup vars\ b_2$
$vars\ (Less\ a_1\ a_2) = vars\ a_1 \cup vars\ a_2$

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 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 wether 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:

$$D :: vname\ set \Rightarrow com \Rightarrow vname\ set \Rightarrow bool$$

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 10.1 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.

10.1.2 Initialisation Sensitive Expression Evaluation thy

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*. As mentioned in Section 2.3.1 in the Isabelle part in the beginning, Isabelle provides the *option* data type, which is useful for precisely such situations:

datatype $'a\ option = None \mid Some\ 'a$

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 $s(x := Some\ y)$, and $dom\ s = \{a.\ s\ a \neq None\}$.

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 9.1. 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 $aval :: aexp \Rightarrow state \Rightarrow val\ option$ **where**
$aval\ (N\ i)\ s = Some\ i$
$aval\ (V\ x)\ s = s\ x$
$aval\ (Plus\ a_1\ a_2)\ s = (case\ (aval\ a_1\ s, aval\ a_2\ s)\ of$
$\quad (Some\ i_1, Some\ i_2) \Rightarrow Some(i_1+i_2)$
$\quad \mid \_ \Rightarrow None)$

fun $bval :: bexp \Rightarrow state \Rightarrow bool\ option$ **where**
$bval\ (Bc\ v)\ s = Some\ v$
$bval\ (Not\ b)\ s = (case\ bval\ b\ s\ of$
$\quad None \Rightarrow None \mid Some\ bv \Rightarrow Some(\neg\ bv))$
$bval\ (And\ b_1\ b_2)\ s = (case\ (bval\ b_1\ s, bval\ b_2\ s)\ of$
$\quad (Some\ bv_1, Some\ bv_2) \Rightarrow Some(bv_1 \wedge bv_2)$
$\quad \mid \_ \Rightarrow None)$
$bval\ (Less\ a_1\ a_2)\ s = (case\ (aval\ a_1\ s, aval\ a_2\ s)\ of$
$\quad (Some\ i_1, Some\ i_2) \Rightarrow Some(i_1 < i_2)$
$\quad \mid \_ \Rightarrow None)$

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 10.1 (Initialised arithmetic expressions).
$vars\ a \subseteq dom\ s \Longrightarrow \exists i.\ aval\ a\ s = Some\ i$

Lemma 10.2 (Initialised boolean expressions).
$vars\ b \subseteq dom\ s \Longrightarrow \exists bv.\ bval\ b\ s = Some\ bv$

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

10.1.3 Initialisation Sensitive Small-Step Semantics thy

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.

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

$$\frac{}{(SKIP;;\ c,\ s) \to (c,\ s)} \qquad \frac{(c_1,\ s) \to (c_1',\ s')}{(c_1;;\ c_2,\ s) \to (c_1';;\ c_2,\ s')}$$

$$\frac{bval\ b\ s = Some\ True}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_1,\ s)}$$

$$\frac{bval\ b\ s = Some\ False}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ s) \to (c_2,\ s)}$$

$$\frac{}{(WHILE\ b\ DO\ c,\ s) \to (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP,\ s)}$$

Fig. 10.2. Small-step semantics, initialisation sensitive

In fact, the development is so standard that we only show the small-step semantics in Figure 10.2 and give one hint for the soundness proof. It needs the following lemma.

Lemma 10.3 (D is increasing). $D\ A\ c\ A' \Longrightarrow A \subseteq A'$

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

The soundness statement then is as in the type system in Section 9.1.

Theorem 10.4 (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 10.3. 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.

10.1.4 Initialisation Sensitive Big-Step Semantics thy

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.

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

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.

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

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

$$(c,\ None) \Rightarrow None$$

Consequently, in the rest of the semantics in Figure 10.3 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:

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

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

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

The remaining rules in Figure 10.3 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*.

$$[\![D\ (dom\ s)\ c\ A';\ (c,\ Some\ s) \Rightarrow s']\!] \Longrightarrow s' \neq None$$

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 A and A' need to be variables in the statement to produce suitably general induction hypotheses. Replacing $dom\ s$ 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 $A \subseteq dom\ s$, 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.

$$\frac{}{(c,\ None) \Rightarrow None} \qquad \frac{}{(SKIP,\ s) \Rightarrow s}$$

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

$$\frac{(c_1,\ s_1) \Rightarrow s_2 \qquad (c_2,\ s_2) \Rightarrow s_3}{(c_1;;\ c_2,\ s_1) \Rightarrow s_3}$$

$$\frac{bval\ b\ s = Some\ True \qquad (c_1,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = Some\ False \qquad (c_2,\ Some\ s) \Rightarrow s'}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow s'}$$

$$\frac{bval\ b\ s = None}{(IF\ b\ THEN\ c_1\ ELSE\ c_2,\ Some\ s) \Rightarrow None}$$

$$\frac{bval\ b\ s = Some\ False}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow Some\ s}$$

$$\frac{bval\ b\ s = Some\ True \qquad (c,\ Some\ s) \Rightarrow s' \qquad (WHILE\ b\ DO\ c,\ s') \Rightarrow s''}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow s''}$$

$$\frac{bval\ b\ s = None}{(WHILE\ b\ DO\ c,\ Some\ s) \Rightarrow None}$$

Fig. 10.3. Big-step semantics with error propagation

Lemma 10.5 (Soundness of D).
$[\![(c,\ Some\ s) \Rightarrow s';\ D\ A\ c\ A';\ A \subseteq dom\ s]\!]$
$\Longrightarrow \exists t.\ s' = Some\ t \wedge A' \subseteq dom\ t$

Proof. The proof is by rule induction on $(c,\ Some\ s) \Rightarrow s'$; 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 10.3. 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 $A' \subseteq dom\ t$. To apply the induction hypothesis for the rest of the *WHILE* loop, however, we need $A \subseteq dom\ t$. From Lemma 10.3 we know that $A \subseteq A'$ and can therefore proceed. □

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.

10.1.5 Exercises

Exercise 10.6. Write an executable version of the definite assignment check by providing two recursive functions: one that compute the set of assigned variables, and one that checks whether only assigned variables are used. Prove that your formulation is equivalent with the inductive formulation.

Exercise 10.7. Extend the analysis D in this section such that it takes the constant boolean expressions *True* and *False* into account. Re-prove soundness.

10.2 Constant Folding and Propagation thy

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 chapter 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 lead 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.

10.2.1 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. In the first part of this book in Section 3.1.3, after our first contact with arithmetic expressions, we already wrote such a function.

At that time we could not simplify variables, i.e. we defined

$$asimp\_const\ (V\ x) = V\ x$$

In this section, however, we are going to mix constant folding with constant propagation, and, 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 $afold :: aexp \Rightarrow tab \Rightarrow aexp$ **where**
$afold\ (N\ n)\ \_ = N\ n$
$afold\ (V\ x)\ t = (case\ t\ x\ of\ None \Rightarrow V\ x \mid Some\ x \Rightarrow N\ x)$
$afold\ (Plus\ e_1\ e_2)\ t = (case\ (afold\ e_1\ t,\ afold\ e_2\ t)\ of$
$(N\ n_1,\ N\ n_2) \Rightarrow N\ (n_1{+}n_2)$
$\mid (e_1',\ e_2') \Rightarrow Plus\ e_1'\ e_2')$

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:

$$aval\ (afold\ a\ t)\ s = aval\ a\ s$$

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 $approx\ t\ s = (\forall x\ k.\ t\ x = Some\ k \longrightarrow s\ x = k)$

With this assumption the statement is provable.

Lemma 10.8. *Correctness of afold*
$approx\ t\ s \Longrightarrow aval\ (afold\ a\ t)\ s = aval\ a\ s$

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.

10.2.2 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 $fold :: com \Rightarrow tab \Rightarrow com$. 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*:

$$fold\ (x ::= a)\ t = x ::= afold\ a\ t$$

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 10.4 shows the main definition. Auxiliary function *lvars* computes the set of variables on the left-hand side of assignments (see Appendix A). Function *merge* computes the intersection of two tables:

$$merge\ t_1\ t_2 = (\lambda m.\ if\ t_1\ m = t_2\ m\ then\ t_1\ m\ else\ None)$$

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

```
fun defs :: com ⇒ tab ⇒ tab where
defs SKIP t                          = t
defs (x ::= a) t                     = (case afold a t of
                                          N k ⇒ t(x ↦ k)
                                        | _ ⇒ t(x:=None))
defs (c1;; c2) t                     = (defs c2 ∘ defs c1) t
defs (IF b THEN c1 ELSE c2) t = merge (defs c1 t) (defs c2 t)
defs (WHILE b DO c) t                = t↑(− lvars c)
```

Fig. 10.4. Definition of *defs*.

- 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 as in Chapter 13, 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 $t{\uparrow_S}$ is defined as follows.

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

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**
$fold\ SKIP\ \_ = SKIP$
$fold\ (x ::= a)\ t = x ::= afold\ a\ t$
$fold\ (c_1;;\ c_2)\ t = fold\ c_1\ t;;\ fold\ c_2\ (defs\ c_1\ t)$
$fold\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ t = IF\ b\ THEN\ fold\ c_1\ t\ ELSE\ fold\ c_2\ t$
$fold\ (WHILE\ b\ DO\ c)\ t = WHILE\ b\ DO\ fold\ c\ (t\restriction_{(-\ lwars\ c)})$

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 this book, 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*.

10.2.3 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 7.2.4, 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
$$(P \models b <\sim> b') = (\forall s.\ P\ s \longrightarrow bval\ b\ s = bval\ b'\ s)$$

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 10.9. $((\lambda\_.\ True) \models c \sim c') = (c \sim c')$

Proof. By unfolding definitions. □

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

Lemma 10.10 (Equivalence Relation).
$P \models c \sim c$
$(P \models c \sim c') = (P \models c' \sim c)$
$[\![P \models c \sim c';\ P \models c' \sim c'']\!] \Longrightarrow P \models c \sim c''$

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:

$$[\![P \models c \sim c';\ \forall s.\ P'\ s \longrightarrow P\ s]\!] \Longrightarrow P' \models c \sim c'$$

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 10.5 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 10.1, 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 between P and Q: Q must hold in the states after execution of c, provided P held in the initial state.

$$\frac{P \models c \sim c' \qquad Q \models d \sim d' \qquad \forall s\ s'.\ (c, s) \Rightarrow s' \longrightarrow P\ s \longrightarrow Q\ s'}{P \models c;;\ d \sim c';;\ d'}$$

$$\frac{P \models b <\sim> b' \qquad P \models c \sim c' \qquad P \models d \sim d'}{P \models IF\ b\ THEN\ c\ ELSE\ d \sim IF\ b'\ THEN\ c'\ ELSE\ d'}$$

$$\frac{\begin{array}{c} P \models b <\sim> b' \\ P \models c \sim c' \qquad \forall s\ s'.\ (c, s) \Rightarrow s' \longrightarrow P\ s \longrightarrow bval\ b\ s \longrightarrow P\ s' \end{array}}{P \models WHILE\ b\ DO\ c \sim WHILE\ b'\ DO\ c'}$$

Fig. 10.5. Congruence rules for conditional semantic equivalence.

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 7.2.4, we leave the proof of the rules in Figure 10.5 as an exercise, as well as the formulation of the strengthened rules that take boolean expressions further into account.

10.2.4 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 10.2.1. 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 10.11 (*defs* approximates execution correctly).
$[\![(c, s) \Rightarrow s';\ approx\ t\ s]\!] \Longrightarrow approx\ (defs\ c\ t)\ s'$

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 10.8.
- 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 known $approx\ (defs\ c\ t{\restriction_{(-\ lvars\ c)}})\ s'$, but our

proof goal for *defs* applied to the while loop is $approx\ (t{\restriction}_{(-\ lvars\ c)})\ s'$. Lemma 10.12 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 10.12. $defs\ c\ t{\restriction}_{(-\ lvars\ c)} = t{\restriction}_{(-\ lvars\ c)}$

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:

$$[\![t_1{\restriction}_S = t{\restriction}_S;\ t_2{\restriction}_S = t{\restriction}_S]\!] \Longrightarrow merge\ t_1\ t_2{\restriction}_S = t{\restriction}_S$$

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

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 10.13.
$[\![(c,\ s) \Rightarrow s';\ approx\ (t{\restriction}_{(-\ lvars\ c)})\ s]\!] \Longrightarrow approx\ (t{\restriction}_{(-\ lvars\ c)})\ s'$

Proof. This proof is by rule induction on the big-step execution. Its cases are very similar to Lemma 10.12. □

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

Lemma 10.14 (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 10.8. Sequential composition uses the congruence rule for semicolon and Lemma 10.11. The *IF* case is automatic given the *IF* congruence rule. The *WHILE* case reduces to Lemma 10.13, the *WHILE* congruence rule, and strengthening of the equivalence condition. The strengthening uses the following property

$$[\![approx\ t_2\ s;\ t_1 \subseteq_m t_2]\!] \Longrightarrow approx\ t_1\ s$$

where $(m_1 \subseteq_m m_2) = (m_1 = m_2\ on\ dom\ m_1)$ and $t{\restriction}_S \subseteq_m t$. □

This leads us to the final result.

Theorem 10.15 (Correctness of constant folding).
fold c empty $\sim$ *c*

Proof. Follows immediately from Lemma 10.14 after observing that *approx empty* $= (\lambda\_.\ True)$. □

10.2.5 Exercises

Exercise 10.16. Extend *afold* with simplifying addition of 0. That is, for any expression e, $e + 0$ and $0 + e$ should be simplified to just e, including the case where the 0 is produced by knowledge of the content of variables. Re-prove the results in this section with the extended version.

Exercise 10.17. Prove the rules in Figure 10.5.

Exercise 10.18. Strengthen and re-prove the congruence rules for conditional semantic equivalence in Figure 10.5 to take the value of boolean expressions into account in the `IF` and `WHILE` cases.

Exercise 10.19. Define and prove correct *copy propagation*. Copy propagation is similar to constant folding, but propagates the right-hand side of assignments if these right-hand sides are just variables. For instance, the program `x := y; z := x + z` will be transformed into `x := y; z := y + z`. The assignment `x := y` can then be eliminated in a liveness analysis. Copy propagation is useful for cleaning up after other optimisation phases.

Exercise 10.20. Extend constant folding with analysing boolean expressions and eliminate dead `IF` branches as well as loops whose body is never executed. Hint: you will need to make use of stronger congruence rules for conditional semantic equivalence.

10.3 Live Variable Analysis thy

This section presents another important analysis that enables program optimisations, namely the elimination of assignments to a variables whose value is not needed afterwards. Here is a simple example:

```
x := 0; y := 1; x := y
```

The first assignment to x is redundant because x is dead at this point: it is overwritten by the second assignment to x without x having been read in between. In contrast, the assignment to y is not redundant, y is live at that point.

Semantically, variable x is live before command c if the initial value of x before execution of c can influence the final state after execution of c. A weaker but easier to check condition is the following: we call x **live** before c if there is some potential execution of c where x is read for the first time before it is overwritten. For the moment, all variables are implicitly read at the end of c. A variable is **dead** if it is not live. The phrase "potential execution" refers to the fact that we do not analyse boolean expressions.

Example 10.21.

- `x := rhs`
  Variable x is dead before this assignment unless `rhs` contains x.
  Variable y is live before this assignment if `rhs` contains y.
- `IF b THEN x := y ELSE SKIP`
  Variable y is live before this command because execution could potentially enter the `THEN` branch.
- `x := y; x := 0; y := 1`
  Variable y is live before this command (because of `x := y`) although the value of y is semantically irrelevant because the second assignment overwrites the first one. This example shows that the above definition of liveness is strictly weaker than the semantic notion. We will improve on this under the heading of "True Liveness" in Section 10.4.

Let us now formulate liveness analysis as a recursive function. This requires us to generalise the liveness notion w.r.t. a set of variables X that are implicitly read at the end of a command. The reason is that this set changes during the analysis. Therefore it needs to be a parameter of the analysis and we speak of the set of variables live before a command c relative to a set of variables X. It is computed by the function $L\ c\ X$ defined like this:

fun $L :: com \Rightarrow vname\ set \Rightarrow vname\ set$ **where**

$$
\begin{aligned}
&L\ SKIP\ X && = X\\
&L\ (x ::= a)\ X && = vars\ a \cup (X - \{x\})\\
&L\ (c_1;;\ c_2)\ X && = L\ c_1\ (L\ c_2\ X)\\
&L\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X && = vars\ b \cup L\ c_1\ X \cup L\ c_2\ X\\
&L\ (WHILE\ b\ DO\ c)\ X && = vars\ b \cup X \cup L\ c\ X
\end{aligned}
$$

In a nutshell, $L\ c\ X$ computes the set of variables that are live *before* c given the set of variables X that are live *after* c (hence the order of arguments in $L\ c\ X$).

We discuss the equations for L one by one. The one for $SKIP$ is obvious. The one for $x ::= a$ expresses that before the assignment all variables in a are live (because they are read) and that x is not live (because it is overwritten) unless it also occurs in a. The equation for $c_1;; c_2$ expresses that the computation of live variables proceeds backwards. The equation for IF expresses that the variables of b are read, that b is not analysed and both the $THEN$ and the $ELSE$ branch could be executed, and that a variable is live if it is live on some computation path leading to some point—hence the $\cup$. The situation for $WHILE$ is similar: execution could skip the loop (hence X is live) or it could execute the loop body once (hence $L\ c\ X$ is live). But what if the loop body is executed multiple times?

In the following discussion we assume this abbreviation:

$$w = WHILE\ b\ DO\ c$$

For a more intuitive understanding of the analysis of loops one should think of w as the control-flow graph in Figure 10.6. A control-flow graph is

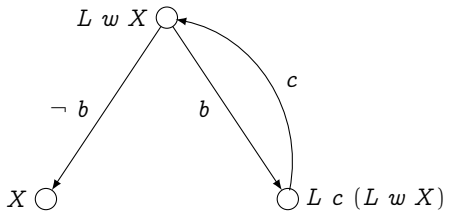


Fig. 10.6. Control-flow graph for $WHILE\ b\ DO\ c$

a graph whose nodes represent program points and whose edges are labelled with boolean expressions or commands. The operational meaning is that execution moves a state from node to node: a state s moves unchanged across an edge labelled with b provided $bval\ b\ s$, and moving across an edge labelled with c transformers s into the new state resulting from the execution of c.

In Figure 10.6 we have additionally annotated each node with the set of variables live at that node. At the exit of the loop, X should be live, and at the beginning $L\ w\ x$. Now let us pretend we had not defined $L\ w\ X$ yet but were looking for constraints that it must satisfy. An edge $Y \xrightarrow{e} Z$ (where e is a boolean expression and Y, Z are liveness annotations) should satisfy $vars\ e \subseteq Y$ and $Z \subseteq Y$ (because the variables in e are read but no variable is written). Thus the graph leads to the following three constraints:

$$\left.\begin{array}{ll} vars\ b & \subseteq L\ w\ X \\ X & \subseteq L\ w\ X \\ L\ c\ (L\ w\ X) & \subseteq L\ w\ X \end{array}\right\} \tag{10.1}$$

The first two constraints are met by our definition of L, but for the third constraint this is not clear. To facilitate proofs about L we now rephrase its definition as an instance of a general analysis.

10.3.1 Generate and Kill Analyses

This is a class of simple analyses that operate on sets. That is, each analysis in this class is a function $A :: com \Rightarrow \tau\ set \Rightarrow \tau\ set$ (for some type τ) that can be defined as

$$A\ c\ S = gen\ c \cup (S - kill\ c)$$

for suitable auxiliary functions *gen* and *kill* of type $com \Rightarrow \tau\ set$ that specify what is to be added and what is to be removed from the input set. Gen/kill analyses satisfy nice algebraic properties and many standard analyses can be expressed in this form, in particular liveness analysis. For liveness, $gen\ c$ are the variables that may be read in c before they are written and $kill\ c$ are the variables that are definitely written in c:

fun $kill :: com \Rightarrow vname\ set$ **where**
$kill\ SKIP = \{\}$
$kill\ (x ::= a) = \{x\}$
$kill\ (c_1;;\ c_2) = kill\ c_1 \cup kill\ c_2$
$kill\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) = kill\ c_1 \cap kill\ c_2$
$kill\ (WHILE\ b\ DO\ c) = \{\}$

fun $gen :: com \Rightarrow vname\ set$ **where**
$gen\ SKIP = \{\}$
$gen\ (x ::= a) = vars\ a$
$gen\ (c_1;;\ c_2) = gen\ c_1 \cup (gen\ c_2 - kill\ c_1)$
$gen\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) = vars\ b \cup gen\ c_1 \cup gen\ c_2$
$gen\ (WHILE\ b\ DO\ c) = vars\ b \cup gen\ c$

Note that *gen* uses *kill* in the only not-quite-obvious equation $gen\ (c_1;;\ c_2) = gen\ c_1 \cup (gen\ c_2 - kill\ c_1)$ where $gen\ c_2 - kill\ c_1$ expresses that variables that are read in c_2 but were written in c_1 are not live before $c_1;;\ c_2$ (unless they are also in $gen\ c_1$).

Lemma 10.22 (Liveness via gen/kill). $L\ c\ X = gen\ c \cup (X - kill\ c)$

The proof is a simple induction on c. As a consequence of this lemma we obtain

Lemma 10.23. $L\ c\ (L\ w\ X) \subseteq L\ w\ X$

Proof. $L\ c\ (L\ w\ X) = L\ c\ (vars\ b \cup gen\ c \cup X) = gen\ c \cup ((vars\ b \cup gen\ c \cup X) - kill\ c) \subseteq gen\ c \cup (X - kill\ c) = L\ c\ X$. □

Moreover, we can prove that $L\ w\ X$ is the *least* solution for the constraint system (10.1). This shows that our definition of $L\ w\ X$ is optimal: the fewer live variables, the better; from the perspective of program optimisation, the only good variable is a dead variable. To prove that $L\ w\ X$ is the least solution of (10.1), assume that P is a solution of (10.1), i.e. $vars\ b \subseteq P$, $X \subseteq P$ and $L\ c\ P \subseteq P$. Because $L\ c\ P = gen\ c \cup (P - kill\ c)$ we also have $gen\ c \subseteq P$. Thus $L\ w\ X = vars\ b \cup gen\ c \cup X \subseteq P$ by assumptions.

10.3.2 Correctness

So far we have proved that $L\ w\ X$ satisfies the informally derived constraints. Now we prove formally that L is correct w.r.t. the big-step semantics. Roughly speaking we want the following: when executing c, the final value of $x \in X$ only depends on the initial values of variables in $L\ c\ X$. Put differently:

> If two initial states of the execution of c agree on $L\ c\ X$
> then the corresponding final states agree on X.

To formalise this statement we introduce the abbreviation $f = g\ on\ X$ for two functions $f, g :: 'a \Rightarrow 'b$ being the same on a set $X :: 'a\ set$:

$$\boxed{f = g\ on\ X \equiv \forall x{\in}X.\ f\ x = g\ x}$$

With this notation we can concisely express that the value of an expression only depends of the value of the variables in the expression:

Lemma 10.24 (Coincidence).

1. $s_1 = s_2\ on\ vars\ a \Longrightarrow aval\ a\ s_1 = aval\ a\ s_2$
2. $s_1 = s_2\ on\ vars\ b \Longrightarrow bval\ b\ s_1 = bval\ b\ s_2$

The proofs are by induction on a and b.

The actual correctness statement for live variable analysis is a simulation property (see Section 8.4). The diagrammatic form of the statement is shown in Figure 10.7 where $(c,\ s) \Rightarrow t$ is displayed as $s \overset{c}{\Longrightarrow} t$. Theorem 10.25 expresses the diagram as a formula.

Theorem 10.25 (Correctness of L).
$[\![(c,\ s) \Rightarrow s';\ s = t\ on\ L\ c\ X]\!] \Longrightarrow \exists t'.\ (c,\ t) \Rightarrow t' \wedge s' = t'\ on\ X$

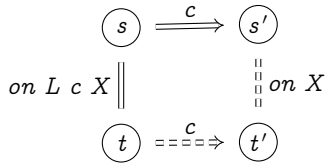
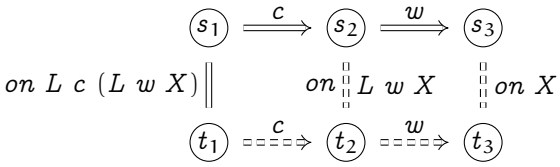


Fig. 10.7. Correctness of L

Proof. The proof is by rule induction. The only interesting cases are the assignment rule, which is correct by the Coincidence Lemma, and rule *WhileTrue*. For the correctness proof of the latter we assume its hypotheses $bval\ b\ s_1$, $(c, s_1) \Rightarrow s_2$ and $(w, s_2) \Rightarrow s_3$. Moreover we assume $s_1 = t_1\ on\ L\ w\ X$ and therefore in particular $s_1 = t_1\ on\ L\ c\ (L\ w\ X)$ because $L\ c\ (L\ w\ X) \subseteq L\ w\ X$. Thus the induction hypothesis for $(c, s_1) \Rightarrow s_2$ applies and we obtain t_2 such that $(c, s_1) \Rightarrow t_2$ and $s_2 = t_2\ on\ L\ w\ X$. The latter enables the application of the induction hypothesis for $(w, s_2) \Rightarrow s_3$ which yields t_3 such that $(w, t_2) \Rightarrow t_3$ and $s_3 = t_3\ on\ X$. By means of the Coincidence Lemma, $s_1 = t_1\ on\ L\ w\ X$ and $vars\ b \subseteq L\ w\ X$ imply $bval\ b\ s_1 = bval\ b\ t_1$. Therefore rule *WhileTrue* yields $(w, t_1) \Rightarrow t_3$ as required. A graphical view of the skeleton of this argument:



□

Note that the proofs of the loop cases (*WhileFalse* too) do not rely on the definition of L but merely on the constraints (10.1).

10.3.3 Optimisation

With the help of the analysis we can program an optimiser $bury\ c\ X$ that eliminates assignments to dead variables from c where X is of course the set of variables live at the end.

fun $bury :: com \Rightarrow vname\ set \Rightarrow com$ **where**
$bury\ SKIP\ X = SKIP$
$bury\ (x ::= a)\ X = (if\ x \in X\ then\ x ::= a\ else\ SKIP)$
$bury\ (c_1;;\ c_2)\ X = bury\ c_1\ (L\ c_2\ X);;\ bury\ c_2\ X$
$bury\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ X =$

$$\begin{array}{l} \quad \mathit{IF}\ b\ \mathit{THEN}\ \mathit{bury}\ c_1\ X\ \mathit{ELSE}\ \mathit{bury}\ c_2\ X \\ \mathit{bury}\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ X\ = \\ \quad \mathit{WHILE}\ b\ \mathit{DO}\ \mathit{bury}\ c\ (L\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ X) \end{array}$$

For simplicity assignments to dead variables are replaced with *SKIP* — eliminating such *SKIP*s in a separate pass was dealt with in Exercise 7.22.

Most of the equations for *bury* are obvious from our understanding of L. For the recursive call of *bury* in the *WHILE* case note that the variables live after c are $L\ w\ X$ (just look at Figure 10.6).

Now we need to understand in what sense this optimisation is correct and then prove it. Our correctness criterion is the big-step semantics: the transformed program must be equivalent to the original one: $\mathit{bury}\ c\ \mathit{UNIV} \sim c$. Note that *UNIV* (or at least all the variables in c) must be considered live at the end because $c' \sim c$ requires that the final states in the execution of c and c' are identical.

The proof of $\mathit{bury}\ c\ \mathit{UNIV} \sim c$ is split into two directions. We start with $(c, s) \Rightarrow s' \Longrightarrow (\mathit{bury}\ c\ \mathit{UNIV}, s) \Rightarrow s'$. For the induction to go through it needs to be generalised to look almost like the correctness statement for L. Figure 10.8 is the diagrammatic form of Lemma 10.26 and Lemma 10.27.

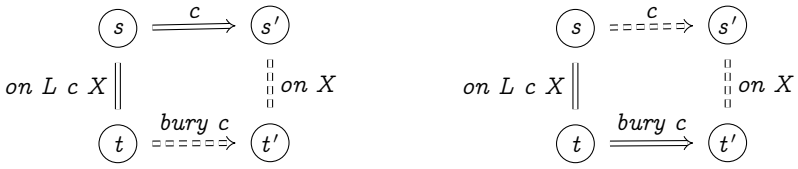


Fig. 10.8. Correctness of *bury* (both directions)

Lemma 10.26 (Correctness of *bury*, part 1).

$$[\![(c, s) \Rightarrow s';\ s = t\ \mathit{on}\ L\ c\ X]\!] \Longrightarrow \exists t'.\ (\mathit{bury}\ c\ X, t) \Rightarrow t' \wedge s' = t'\ \mathit{on}\ X$$

The proof is very similar to the proof of correctness of L. Hence there is no need to go into it.

The other direction $(\mathit{bury}\ c\ \mathit{UNIV}, s) \Rightarrow s' \Longrightarrow (c, s) \Rightarrow s'$ needs to be generalised analogously:

Lemma 10.27 (Correctness of *bury*, part 2).

$$[\![(\mathit{bury}\ c\ X, s) \Rightarrow s';\ s = t\ \mathit{on}\ L\ c\ X]\!] \Longrightarrow \exists t'.\ (c, t) \Rightarrow t' \wedge s' = t'\ \mathit{on}\ X$$

Proof. The proof is also similar to that of correctness of L but induction requires the advanced form explained in Section 5.4.6. As a result, in each case of the induction, there will be an assumption that $bury\ c\ X$ is of a particular form, e.g. $c_1';;\ c_2' = bury\ c\ X$. Now we need to infer that c must be a sequential composition too. The following property expresses this fact:

$$(c_1';;\ c_2' = bury\ c\ X) =$$
$$(\exists\, c_1\ c_2.\ c = c_1;;\ c_2 \wedge c_2' = bury\ c_2\ X \wedge c_1' = bury\ c_1\ (L\ c_2\ X))$$

Its proof is by cases on c, the rest is automatic. This property can either be proved as a separate lemma or c_1' and c_2' can be obtained locally within the case of the induction that deals with sequential composition. Because *bury* preserves the structure of the command, all the required lemmas look similar, except for *SKIP*, which can be the result of either a *SKIP* or an assignment:

$$(SKIP = bury\ c\ X) = (c = SKIP \vee (\exists\, x\ a.\ c = x ::= a \wedge x \notin X))$$

We merely need the left-to-right implication of these function inversion properties but that the formulation as an equivalence permits us to use them as simplification rules. □

Combining the previous two lemmas we obtain

Corollary 10.28 (Correctness of *bury*). $bury\ c\ UNIV \sim c$

10.3.4 Exercises

Exercise 10.29. Prove the following termination-insensitive version of the correctness of L:

$$[\![(c,\ s) \Rightarrow s';\ (c,\ t) \Rightarrow t';\ s = t\ on\ L\ c\ X]\!] \Longrightarrow s' = t'\ on\ X$$

Do not derive it as a corollary to the original correctness theorem but prove it separately. Hint: modify the original proof.

Exercise 10.30. Find a command c such that $bury\ c\ \{\} \neq bury\ (bury\ c\ \{\})\ \{\}$. For an arbitrary command, can you put a limit on the amount of burying needed until everything that is dead is also buried?

Exercise 10.31. An **available definitions** analysis determines which previous assignments `x := a` are valid equalities `x = a` at later program points. For example, after `x := y+1` the equality `x = y+1` is available, but after `x := y+1; y := 2` the equality `x = y+1` is no longer available. The motivation for the analysis is that if `x = a` is available before `v := a` then `v := a` can be replaced by `v := x`.

Define an available definitions analysis $AD :: (vname \times aexp)\ set \Rightarrow com \Rightarrow (vname \times aexp)\ set$. A call $AD\ A\ c$ should compute the available

definitions after the execution of c assuming that the definitions in A are available before the execution of c. This is a gen/kill analysis! Prove correctness of the analysis: if $(c, s) \Rightarrow s'$ and $\forall (x, a) \in A.\ s\ x = aval\ a\ s$ then $\forall (x, a) \in AD\ c\ A.\ s'\ x = aval\ a\ s'$.

10.4 True Liveness thy

In Example 10.21 we had already seen that our definition of liveness is too simplistic: in `x := y; x := 0`, variable `y` is read before it can be written, but it is read in an assignment to a dead variable. Therefore we modify the definition of $L\ (x ::= a)\ X$ to consider $vars\ a$ live only if x is live:

$$L\ (x ::= a)\ X = (if\ x \in X\ then\ vars\ a \cup (X - \{x\})\ else\ X)$$

As a result, our old analysis of loops is no longer correct.

Example 10.32. Consider for a moment the following specific w and c

$$\begin{aligned} w &= WHILE\ Less\ (N\ 0)\ (V\ x)\ DO\ c \\ c &= x ::= V\ y;;\ y ::= V\ z \end{aligned}$$

where x, y and z are distinct. Then $L\ w\ \{x\} = \{x, y\}$ but z is live too, semantically: the initial value of z can influence the final value of x. This is the computation of L: $L\ c\ \{x\} = L\ (x ::= V\ y)\ (L\ (y ::= V\ z)\ \{x\}) = L\ (x ::= V\ y)\ \{x\} = \{y\}$ and therefore $L\ w\ \{x\} = \{x\} \cup \{x\} \cup L\ c\ \{x\} = \{x, y\}$. The reason is that $L\ w\ X = \{x,y\}$ is no longer a solution of the last constraint of (10.1): $L\ c\ \{x, y\} = L\ (x ::= V\ y)\ (L\ (y ::= V\ z)\ \{x, y\}) = L\ (x ::= V\ y)\ \{x, z\} = \{y, z\} \not\subseteq \{x, y\}$.

Let us now abstract from this example and reconsider (10.1). We still want $L\ w\ X$ to be a solution of (10.1) because the proof of correctness of L depends on it. An equivalent formulation of (10.1) is

$$vars\ b \cup X \cup L\ c\ (L\ w\ X) \subseteq L\ w\ X \tag{10.2}$$

That is, $L\ w\ X$ should be some set Y such that $vars\ b \cup X \cup L\ c\ Y \subseteq Y$. For optimality we want the least such Y. We will now study abstractly under what conditions a least such Y exists and how to compute it.

10.4.1 The Knaster-Tarski Fixpoint Theorem on Sets

Definition 10.33. *A type 'a is a* ***partial order*** *if there is binary predicate* $\leqslant$ *on 'a which is*

reflexive: $x \leqslant x$,
transitive: $[\![x \leqslant y;\ y \leqslant z]\!] \Longrightarrow x \leqslant z$, *and*
antisymmetric: $[\![x \leqslant y;\ y \leqslant x]\!] \Longrightarrow x = y$.

We use "$\leqslant$ is a partial order" and "type $'a$ is a partial order" interchangeably.

Definition 10.34. *If $'a$ is a partial order and $A :: 'a$ set, then $p \in A$ is a* ***least*** *element of A if $p \leqslant q$ for all $q \in A$.*

Least elements are unique: if p_1 and p_2 are least elements of A then $p_1 \leqslant p_2$ and $p_2 \leqslant p_1$ and hence $p_1 = p_2$ by antisymmetry.

Definition 10.35. *Let τ be a type and $f :: \tau \Rightarrow \tau$. A point $p :: \tau$ is a* ***fixpoint*** *of f if $f\ p = p$.*

Definition 10.36. *Let τ be a type with a partial order $\leqslant$ and $f :: \tau \Rightarrow \tau$.*

- *Function f is* ***monotone*** *if $x \leqslant y \Longrightarrow f\ x \leqslant f\ y$ for all x and y.*
- *A point $p :: \tau$ is a* ***pre-fixpoint*** *of f if $f\ p \leqslant p$.*

This definition applies in particular to sets, where the partial order is $\subseteq$. Hence we can rephrase (10.2) as saying that $L\ w\ X$ should be a pre-fixpoint of $\lambda Y.\ vars\ b \cup X \cup L\ c\ Y$, ideally the least one. The Knaster-Tarski fixpoint theorem tells us that all we need is monotonicity.

Theorem 10.37. *If $f :: \tau\ set \Rightarrow \tau\ set$ is monotone then $\bigcap\{P.\ f\ P \subseteq P\}$ is the least pre-fixpoint of f.*

Proof. Let $M = \{P.\ f\ P \subseteq P\}$. First we show that $\bigcap M$ is a pre-fixpoint. For all $P \in M$ we have $\bigcap M \subseteq P$ (by definition of $\bigcap$) and therefore $f\ (\bigcap M) \subseteq f\ P \subseteq P$ (by monotonicity and because $P \in M$). Therefore $f\ (\bigcap M) \subseteq \bigcap M$ (by definition of $\bigcap$). Moreover, $\bigcap M$ is the least pre-fixpoint: if $f\ P \subseteq P$ then $P \in M$ and thus $\bigcap M \subseteq P$. □

Lemma 10.38. *Let f be a monotone function on a partial order $\leqslant$. Then a least pre-fixpoint of f is also a least fixpoint.*

Proof. Let P be a least pre-fixpoint of f. From $f\ P \leqslant P$ and monotonicity we have $f(f\ P) \leqslant f\ P \leqslant P$ and therefore $P \leqslant f\ P$ because P is a least pre-fixpoint. Together with $f\ P \leqslant P$ this yields $f\ P = P$ (by antisymmetry). Moreover, P is the least fixpoint because any fixpoint is a pre-fixpoint and P is the least pre-fixpoint. □

The **Knaster-Tarski fixpoint theorem** is the combination of both results: Every monotone function on sets has a least pre-fixpoint which is also a least fixpoint.

Theorem 10.39. *If* $f :: \tau\ set \Rightarrow \tau\ set$ *is monotone then*

$$\mathit{lfp}\, f\ =\ \bigcap\{P.\ f\ P \subseteq P\}$$

is the least pre-fixpoint of f*, which is also its least fixpoint.*

We have separated the two parts because the second part is a generally useful result about partial orders.

Now we boldly define $L\ w\ X$ as a least fixpoint:

$$L\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ X\ =\ \mathit{lfp}\ (\lambda Y.\ \mathit{vars}\ b \cup X \cup L\ c\ Y)$$

This is the full definition of our revised L for true liveness:

fun $L :: \mathit{com} \Rightarrow \mathit{vname\ set} \Rightarrow \mathit{vname\ set}$ **where**
$L\ \mathit{SKIP}\ X = X\ |$
$L\ (x ::= a)\ X = (\mathit{if}\ x \in X\ \mathit{then}\ \mathit{vars}\ a \cup (X - \{x\})\ \mathit{else}\ X)\ |$
$L\ (c_1;;\ c_2)\ X = L\ c_1\ (L\ c_2\ X)\ |$
$L\ (\mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2)\ X = \mathit{vars}\ b \cup L\ c_1\ X \cup L\ c_2\ X\ |$
$L\ (\mathit{WHILE}\ b\ \mathit{DO}\ c)\ X = \mathit{lfp}\ (\lambda Y.\ \mathit{vars}\ b \cup X \cup L\ c\ Y)$

Only the assignment and *WHILE* cases differ from the version in Section 10.3.

The definition of the *WHILE* case is bold for two reasons: we do not yet know that $\lambda Y.\ \mathit{vars}\ b \cup X \cup L\ c\ Y$ is monotone, i.e. that it makes sense to apply *lfp* to it, and we have no idea how to compute *lfp*.

Lemma 10.40. $L\ c$ *is monotone.*

Proof. The proof is by induction on c. All cases are essentially trivial because $\cup$ is monotone in both arguments and set difference is monotone in the first argument. In the *WHILE* case we additionally need that *lfp* is monotone in the following sense: if $f\ A \subseteq g\ A$ for all A, then $\mathit{lfp}\ f \subseteq \mathit{lfp}\ g$. This is obvious because any pre-fixpoint of f must also be a pre-fixpoint of g. □

As a corollary we obtain that $\lambda Y.\ \mathit{vars}\ b \cup X \cup L\ c\ Y$ is monotone too. Hence, by the Knaster-Tarski fixpoint theorem, it has a least (pre-)fixpoint. Hence $L\ w\ X$ is defined exactly as required, i.e. it satisfies (10.1).

Lemma 10.41 (Correctness of L).
$[\![(c, s) \Rightarrow s';\ s = t\ on\ L\ c\ X]\!] \Longrightarrow \exists t'.\ (c, t) \Rightarrow t' \wedge s' = t'\ on\ X$

Proof. This is the same correctness lemma as in Section 10.3, proved in the same way, but for a modified L. The proof of the *WHILE* case remains unchanged because it only relied on the pre-fixpoint constraints (10.1) that are still satisfied. The proof of correctness of the new definition of $L\ (x ::= a)\ X$ is just as routine as before. □

How about an optimiser *bury* like in the previous section? It turns out that we can reuse that *bury* function verbatim, with true liveness instead of liveness. What is more, even the correctness proof carries over verbatim. The reason: the proof of the *WHILE* case relies only on the constraints (10.1), which hold for both L.

10.4.2 Computing the Least Fixpoint

Under certain conditions, the least fixpoint of a function f can be computed by iterating f, starting from the least element, which in the case of sets is the empty set. By iteration we mean f^n defined as follows:

$$\begin{aligned} f^0\ x &= x \\ f^{n+1}\ x &= f\ (f^n\ x) \end{aligned}$$

In Isabelle, $f^n\ x$ is input as $(f\ \hat{}\hat{}\ n)\ x$.

Lemma 10.42. *Let $f :: \tau\ set \Rightarrow \tau\ set$ be a monotone function. If the chain $\{\} \subseteq f\ \{\} \subseteq f^2\ \{\} \subseteq \ldots$ stabilises after k steps, i.e. $f^{k+1}\ \{\} = f^k\ \{\}$, then lfp $f = f^k\ \{\}$.*

Proof. From $f^{k+1}\ \{\} = f^k\ \{\}$ it follows that $f^k\ \{\}$ is a fixpoint. It is the least fixpoint because it is a subset of any other fixpoint P of f: $f^n\ \{\} \subseteq P$ follows from monotonicity by an easy induction on n. The fact that the $f^n\ \{\}$ form a chain, i.e. $f^n\ \{\} \subseteq f^{n+1}\ \{\}$, follows by a similar induction. □

This gives us a way to compute the least fixpoint. In general this will not terminate, as the sets can grow larger and larger. However, in our application only a finite set of variables is involved, those in the program. Therefore termination is guaranteed.

Example 10.43. Recall the loop from Example 10.32:

$$\begin{aligned} w &= \mathit{WHILE}\ \mathit{Less}\ (N\ 0)\ (V\ x)\ \mathit{DO}\ c \\ c &= x ::= V\ y;;\ y ::= V\ z \end{aligned}$$

To compute $L\ w\ \{x\}$ we iterate $f = (\lambda Y.\ \{x\} \cup L\ c\ Y)$. For compactness the notation $X_2\ c_1\ X_1\ c_2\ X_0$ (where the X_i are sets of variables) abbreviates $X_1 = L\ c_2\ X_0$ and $X_2 = L\ c_1\ X_1$. The following table shows the computation of $f\ \{\}$ through $f^4\ \{\}$:

$$
\begin{array}{rlll}
\{\}\ x ::= V\ y & \{\} & y ::= V\ z & \{\} \\
 & & \Longrightarrow & f\ \{\} = \{x\} \cup \{\} = \{x\} \\
\{y\}\ x ::= V\ y & \{x\} & y ::= V\ z & \{x\} \\
 & & \Longrightarrow & f\ \{x\} = \{x\} \cup \{y\} = \{x, y\} \\
\{y, z\}\ x ::= V\ y & \{x, z\} & y ::= V\ z & \{x, y\} \\
 & & \Longrightarrow & f\ \{x, y\} = \{x\} \cup \{y, z\} = \{x, y, z\} \\
\{y, z\}\ x ::= V\ y & \{x, z\} & y ::= V\ z & \{x, y, z\} \\
 & & \Longrightarrow & f\ \{x, y, z\} = \{x\} \cup \{y, z\} = \{x, y, z\}
\end{array}
$$

The final line confirms that $\{x, y, z\}$ is a fixpoint. Of course this is obvious because $\{x, y, z\}$ cannot get any bigger: it already contains all the variables of the program.

Let us make the termination argument more precise and derive some concrete bounds. We need to compute the least fixpoint of $f = (\lambda Y.\ vars\ b \cup X \cup L\ c\ Y)$. Informally, the chain of the $f^n\ \{\}$ must stabilise because only a finite set of variables is involved—we assume that X is finite. In the following, let $rvars\ c$ be the set of variables read in c (see Appendix A). An easy induction on c shows that $L\ c\ X \subseteq rvars\ c \cup X$. Therefore f is bounded by $U = vars\ b \cup rvars\ c \cup X$ in the following sense: $Y \subseteq U \Longrightarrow f\ Y \subseteq U$. Hence $f^k\ \{\}$ is a fixpoint of f for some $k \leqslant card\ U$, the cardinality of U. More precisely, $k \leqslant card\ (rvars\ c) + 1$ because already in the first step $f\ \{\} \supseteq vars\ b \cup X$.

It remains to give an executable definition of $L\ w\ X$. Instead of programming the required function iteration ourselves we use a combinator from the library theory *While_Combinator*:

$$
\begin{array}{l}
while :: ('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \\
while\ b\ f\ x = (if\ b\ x\ then\ while\ b\ f\ (f\ x)\ else\ x)
\end{array}
$$

The equation makes *while* executable. Calling $while\ b\ f\ x$ leads to a sequence of (tail!) recursive calls $while\ b\ f\ (f^n\ x)$, $n = 0, 1, \ldots$, until $b\ (f^k\ x)$ for some k. The equation cannot be a definition because it may not terminate. It is a lemma derived from the actual definition which is a bit tricky and need not concern us here.

Theory *While_Combinator* also contains a lemma that tells us that *while* can implement *lfp* on finite sets provided termination is guaranteed:

Lemma 10.44. *If* $f :: 'a\ set \Rightarrow 'a\ set$ *is monotone and there is a finite set* U *such that* $A \subseteq U \Longrightarrow f\ A \subseteq U$ *for all* A, *then*

$$lfp\ f = while\ (\lambda A.\ f\ A \neq A)\ f\ \{\}.$$

This is a consequence of Lemma 10.42 and allows us to prove

$L\ (WHILE\ b\ DO\ c)\ X =$
$(let\ f = \lambda Y.\ vars\ b \cup X \cup L\ c\ Y\ in\ while\ (\lambda Y.\ f\ Y \neq Y)\ f\ \{\})$

for finite X. Finally, L has become executable and for our example

```
value let b = Less (N 0) (V ''x'');
          c = ''x'' ::= V ''y'';; ''y'' ::= V ''z''
      in L (WHILE b DO c) {''x''}
```

Isabelle computes the expected result $\{''x'', ''y'', ''z''\}$.

To conclude this section we compare true liveness with liveness. The motivation for true liveness was improved precision, and this was achieved: $L\ (''x'' ::= V\ ''y'';;\ ''x'' ::= N\ 0)\ \{''x''\}$ returns $\{\}$ for true liveness, as opposed to $\{''y''\}$ for liveness, but at the cost of efficiency: true liveness is no longer a gen/kill analysis that can be performed in a single pass over the program, analysis of loops now needs iteration to compute least fixpoints.

10.4.3 Exercises

Exercise 10.45. Compute $L\ w\ \{\}$ for w as in Example 10.43. The result will be nonempty. Explain why it is not strange that even if we are not interested in the value of any variable after w, some variables may still be live before w. The correctness lemma for L may be helpful.

Exercise 10.46. Find a family of commands $c_2, c_3, \ldots$, such that the computation of $L\ (WHILE\ b\ DO\ c_n)\ X$ (for suitable X and b) requires n iterations to reach the least fixpoint. Hint: generalise Example 10.43.

Exercise 10.47. Design a dependency analysis between variables. We say that y depends on x in the execution of c if the initial value of x may influence the final value of y. Define an inductive relation $influences :: vname \Rightarrow com \Rightarrow vname \Rightarrow bool$ which specifies a dependency analysis and prove its correctness: $[\![(c, s) \Rightarrow t;\ s = s'\ on\ deps\ c\ y;\ (c, s') \Rightarrow t']\!] \Longrightarrow t\ y = t'\ y$ where $deps\ c\ y$ abbreviates $\{x.\ influences\ x\ c\ y\}$. Can you strengthen the correctness theorem?

Hints: You may want to use functions *lvars* from Appendix A. When analysing a loop, you can follow the execution in the style of the big-step semantics. Do not worry about executability and termination (for now).

10.5 Summary and Further Reading

This chapter has explored three different, widely used data-flow analyses and associated program optimisations: definite initialisation analysis, con-

stant propagation, and live variable analysis. 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

- Definite initialisation analysis is a forward must analysis: variables must be assigned on all paths before they are used.
- Constant propagation is a forward must analysis: a variable must have the same constant value on all paths.
- Live variable analysis is a backward may analysis: a variable is live if it is used on some path before it is overwritten.

There are also forward may and backward must analyses.

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. [2], but in particular in the book by Muchnik [60]. The book by Nielson, Nielson and Hankin [63] provides a comprehensive and more theoretical account of program analysis.

In Chapter 13 we study "Abstract Interpretation", a powerful but also complex approach to program analysis that generalises the algorithms presented in this chapter.

11

Denotational Semantics thy

So far we have worked exclusively with operational semantics which are defined by inference rules that tell us how to execute some command. But those rules do not tell us directly what the *meaning* of a command is. This is what denotational semantics is about: mapping syntactic objects to their denotation or meaning. In fact, we are already familiar with two examples, namely the evaluation of expressions. The denotation of an arithmetic expression is a function from states to values and $aval :: aexp \Rightarrow (state \Rightarrow val)$ (note the parentheses) is the mapping from syntax to semantics. Similarly, we can think of the meaning of a command as a relation between initial states and final states and can even define

$$Big\_step\ c \equiv \{(s, t).\ (c, s) \Rightarrow t\}$$

If the language is deterministic, this relation is a partial function.

However, *Big_step* is not a true denotational semantics because all the work happens on the level of the operational semantics. A denotational semantics is characterised as follows:

> There is a type *syntax* of syntactic objects, a type *semantics* of denotations and a function $D :: syntax \Rightarrow semantics$ that is defined by primitive recursion. That is, for each syntactic construct C there is a defining equation
>
> $$D\ (C\ x_1\ \ldots\ x_n) = \ldots\ D\ x_1\ \ldots\ D\ x_n\ \ldots$$
>
> In words: the meaning of a compound object is defined as a function of the meanings of its subcomponents.

Both *aval* and *bval* are denotational definitions, but the big-step semantics is not: the meaning of $WHILE\ b\ DO\ c$ is not defined simply in terms of the meaning of b and c: rule *WhileTrue* inductively relies on the meaning of $WHILE\ b\ DO\ c$ in the premise.

One motivation for denotational definitions is that proofs can be conducted by the simple and effective proof principles of equational reasoning and structural induction over the syntax.

11.1 A Relational Denotational Semantics

Although the natural view of the meaning of a deterministic command may be a partial function from initial to final states, it is mathematically simpler to work with relations instead. For this purpose we introduce the identity relation and composition of relations:

$$Id :: ('a \times 'a)\ set$$
$$Id = \{p.\ \exists x.\ p = (x, x)\}$$

$$op \bigcirc :: ('a \times 'b)\ set \Rightarrow ('b \times 'c)\ set \Rightarrow ('a \times 'c)\ set$$
$$r \bigcirc s = \{(x, z).\ \exists y.\ (x, y) \in r \wedge (y, z) \in s\}$$

Note that $r \bigcirc s$ can be read from left to right: first r, then s. The ASCII form of $\bigcirc$ is the letter O.

The type of D and the first four equations should be self-explanatory:

$$\begin{array}{l} D :: com \Rightarrow (state \times state)set \\ D\ SKIP \quad = Id \\ D\ (x ::= a) \ = \{(s, t).\ t = s(x := aval\ a\ s)\} \\ D\ (c_1;;\ c_2) \ = D\ c_1 \bigcirc D\ c_2 \\ D\ (IF\ b\ THEN\ c_1\ ELSE\ c_2) \\ = \{(s, t).\ if\ bval\ b\ s\ then\ (s, t) \in D\ c_1\ else\ (s, t) \in D\ c_2\} \end{array}$$

Example 11.1. Let $c_1 = ''x'' ::= N\ 0$ and $c_2 = ''y'' ::= V\ ''x''$.

$D\ c_1 = \{(s_1, s_2).\ s_2 = s_1(''x'' := 0)\}$
$D\ c_2 = \{(s_2, s_3).\ s_3 = s_2(''y'' := s_2\ ''x'')\}$
$D\ (c_1;;\ c_2) = \{(s_1, s_3).\ s_3 = s_1(''x'' := 0, ''y'' := 0)\}$

The definition of $D\ w$, where $w = WHILE\ b\ DO\ c$, is trickier. Ideally we would like to write the recursion equation

$$D\ w = \{(s, t).\ if\ bval\ b\ s\ then\ (s, t) \in D\ c \bigcirc D\ w\ else\ s = t\} \qquad (*)$$

but because $D\ w$ depends on $D\ w$, this is not in denotational style. What is worse, it would not be accepted by Isabelle because it does not terminate and may be inconsistent: remember the example of the illegal 'definition' $f\ n = f\ n + 1$ where subtracting $f\ n$ on both sides would lead to $0 = 1$. The fact remains that $D\ w$ should be a solution of $(*)$, but we need to show

that a solution exists and to pick a specific one. This is entirely analogous to the problem we faced when analysing true liveness in Section 10.4, and the solution will be the same: the *lfp* operator and the Knaster-Tarski Fixpoint Theorem.

In general, a solution of an equation $t = f\ t$ such as $(*)$ is the same as a fixpoint of f. In the case of $(*)$, the function f is

$$W :: (state \Rightarrow bool) \Rightarrow com\_den \Rightarrow (com\_den \Rightarrow com\_den)$$
$$W\ db\ dc = (\lambda dw.\ \{(s, t).\ \mathit{if}\ db\ s\ \mathit{then}\ (s, t) \in dc \bigcirc dw\ \mathit{else}\ s = t\})$$

where we have abstracted the terms $bval\ b$, $D\ c$ and $D\ w$ in $(*)$ by the parameters db, dc and dw. Hence $W\ (bval\ b)\ (D\ c)\ (D\ w)$ is the right-hand side of $(*)$. Now we define $D\ w$ as a least fixpoint:

$$\boxed{D\ (WHILE\ b\ DO\ c) = lfp\ (W\ (bval\ b)\ (D\ c))}$$

Why the least? The formal justification will be an equivalence proof between denotational and big-step semantics. The following example provides some intuition why leastness is what we want.

Example 11.2. Let $w = WHILE\ Bc\ True\ DO\ SKIP$. Then

$$f = W\ (bval\ (Bc\ True))\ (D\ SKIP) = W\ (\lambda s.\ True)\ Id$$
$$= (\lambda dw.\ \{(s, t).\ (s, t) \in Id \bigcirc dw\}) = (\lambda dw.\ dw)$$

Therefore any relation is a fixpoint of f, but our intuition tells us that because w never terminates, its semantics should be the empty relation, which is precisely the least fixpoint of f.

We still have to prove that $W\ db\ dc$ is monotone. Only then can we appeal to the Knaster-Tarski Fixpoint Theorem to conclude that $lfp\ (W\ db\ dc)$ really is the least fixpoint.

Lemma 11.3. *$W\ db\ dc$ is monotone.*

Proof. For Isabelle, the proof is automatic. The core of the argument rests on the fact that relation composition is monotone in both arguments: if $r \subseteq r'$ and $s \subseteq s'$, then $r \bigcirc s \subseteq r' \bigcirc s'$, which can be seen easily from the definition of $\bigcirc$. If $dw \subseteq dw'$ then $W\ db\ dc\ dw \subseteq W\ db\ dc\ dw'$ because $dc \bigcirc dw \subseteq dc \bigcirc dw'$. □

We could already define D without this monotonicity lemma, but we would not be able to deduce anything about $D\ w$ without it. Now we can derive $(*)$. This is a completely mechanical consequence of the way we defined $D\ w$ and W. Given $t = lfp\ f$ for some monotone f we obtain

$$t = lfp\ f = f\ (lfp\ f) = f\ t$$

where the step $lfp\ f\ =\ f\ (lfp\ f)$ is the consequence of the Knaster-Tarski Fixpoint Theorem because f is monotone. Setting $t\ =\ D\ w$ and $f\ =\ W\ (bval\ b)\ (D\ c)$ in $t\ =\ f\ t$ results in $(*)$ by definition of W.

An immediate consequence is

$$D\ (WHILE\ b\ DO\ c)\ =$$
$$D\ (IF\ b\ THEN\ c;;\ WHILE\ b\ DO\ c\ ELSE\ SKIP)$$

Just expand the definition of D for IF and ;; and you obtain $(*)$. This is an example of the simplicity with which program equivalences can be derived with the help of denotational semantics.

Discussion

Why can't we just define $(*)$ as it is but have to go through the indirection of *lfp* and prove monotonicity of W? None of this was required for the operational semantics! The reason for this discrepancy is that inductive definitions require a very fixed format to be admissible. For this fixed format, it is not hard to prove that the inductively defined predicate actually exists. Isabelle does this by converting the inductive definition internally into a function on sets, proving its monotonicity and defining the inductive predicate as the least fixpoint of that function. The monotonicity proof is automatic provided we stick to the fixed format. Once you step outside the format, in particular when using negation, it will be rejected by Isabelle because least fixpoints may cease to exist and the inductive definition may be plain contradictory:

$$P\ x\ \Longrightarrow\ \neg\ P\ x$$
$$\neg\ P\ x\ \Longrightarrow\ P\ x$$

The analogous recursive 'definition' is $P\ x\ =\ (\neg\ P\ x)$, which is also rejected by Isabelle, because it does not terminate.

To avoid the manual monotonicity proof required for our denotational semantics one could put together a collection of basic functions that are all monotone or preserve monotonicity. One would end up with a little programming language where all functions are monotone and this could be proved automatically. In fact, one could then even automate the translation of recursion equations like $(*)$ into *lfp* format. Creating such a programming language is at the heart of denotational semantics, but we do not go into it in our brief introduction into the subject.

In summary: Although our treatment of denotational semantics appears more complicated than operational semantics because of the explicit *lfp*, operational semantics is defined as a least fixpoint too, but this is hidden inside **inductive**. With more effort one can hide the *lfp* in denotational semantics too and allow direct recursive definitions. This is what Isabelle's **partial_function** command does [52, 93].

11.1.1 Equivalence of Denotational and Big-step Semantics

We show that the denotational semantics is logically equivalent with our gold standard, the big-step semantics. The equivalence is proved as two separate lemmas. Both proofs are almost automatic because the denotational semantics is relational and thus close to the operational one. Even the treatment of *WHILE* is the same: $D\ w$ is defined explicitly as a least fixpoint and the operational semantics is an inductive definition which is internally defined as a least fixpoint (see the Discussion above).

Lemma 11.4. $(c,\ s) \Rightarrow t \implies (s,\ t) \in D\ c$

Proof. By rule induction. All cases are automatic. We just look at *WhileTrue* where we may assume $bval\ b\ s_1$ and the IHs $(s_1,\ s_2) \in D\ c$ and $(s_2,\ s_3) \in D\ (WHILE\ b\ DO\ c)$. We have to show $(s_1,\ s_3) \in D\ (WHILE\ b\ DO\ c)$, which follows immediately from $(*)$. □

The other direction is expressed by means of the abbreviation *Big_step* introduced at the beginning of this chapter. The reason is purely technical.

Lemma 11.5. $(s,\ t) \in D\ c \implies (s,\ t) \in Big\_step\ c$

Proof. By induction on c. All cases are proved automatically except $w = WHILE\ b\ DO\ c$, which we look at in detail. Let $B = Big\_step\ w$ and $f = W\ (bval\ b)\ (D\ c)$. By definition of W it follows that B is a pre-fixpoint of f, i.e. $f\ B \subseteq B$: given $(s,\ t) \in f\ B$, either $bval\ b\ s$ and there is some s' such that $(s,\ s') \in D\ c$ (hence $(c,\ s) \Rightarrow s'$ by IH) and $(w,\ s') \Rightarrow t$, or $\neg\ bval\ b\ s$ and $s = t$; in either case $(w,\ s) \Rightarrow t$, i.e. $(s,\ t) \in B$. Because $D\ w$ is the least fixpoint and also the least pre-fixpoint of f (see Knaster-Tarski), $D\ w \subseteq B$ and hence $(s,\ t) \in D\ w \implies (s,\ t) \in B$ as claimed. □

The combination of the previous two lemma yields the equivalence:

Theorem 11.6 (Equivalence of denotational and big-step semantics). $(s,\ t) \in D\ c \longleftrightarrow (c,\ s) \Rightarrow t$

As a nice corollary we obtain that the program equivalence $\sim$ defined in Section 7.2.4 is the same as denotational equality: if you replace $(c_i,\ s) \Rightarrow t$ in the definition of $c_1 \sim c_2$ by $(s,\ t) \in D\ c_i$ this yields $\forall\ s\ t.\ (s,\ t) \in D\ c_1 \longleftrightarrow (s,\ t) \in D\ c_2$, which is equivalent with $D\ c_1 = D\ c_2$ because two sets are equal iff they contain the same elements.

Corollary 11.7. $c_1 \sim c_2 \longleftrightarrow D\ c_1 = D\ c_2$

11.1.2 Continuity

Denotational semantics is usually not based on relations but on (partial) functions. The difficulty with functions is that the Knaster-Tarski Fixpoint Theorem does not apply. Abstractly speaking, functions do not form a complete lattice (see Chapter 13) because the union of two functions (as relations) is not necessarily again a function. Thus one needs some other means of obtaining least fixpoints. It turns out that the functions one is interested in satisfy a stronger property called **continuity** which guarantees least fixpoints also in the space of partial functions. Although we do not need to worry about existence of fixpoints in our setting, the notion of continuity is interesting in its own right because it allows a much more intuitive characterisation of least fixpoints than Knaster-Tarski. This in turn gives rise to a new induction principle for least fixpoints.

Definition 11.8. *A **chain** is a sequence of sets* $S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$*:*

$$\begin{array}{l} chain :: (nat \Rightarrow 'a\ set) \Rightarrow bool \\ chain\ S = (\forall i.\ S\ i \subseteq S\ (Suc\ i)) \end{array}$$

Our specific kinds of chains are often called **ω-chains**.

In the following we make use of the notation $\bigcup_n A\ n$, the union of all sets $A\ n$ (see Section 4.2). If $n :: nat$ this is $A\ 0 \cup A\ 1 \cup \ldots$.

Definition 11.9. *A function f on sets is called **continuous** if it commutes with* $\bigcup$ *for all chains:*

$$\begin{array}{l} cont :: ('a\ set \Rightarrow 'b\ set) \Rightarrow bool \\ cont\ f = (\forall S.\ chain\ S \longrightarrow f\ (\bigcup_n S\ n) = (\bigcup_n f\ (S\ n))) \end{array}$$

That is, $f(S_0 \cup S_1 \cup \ldots) = f\ S_0 \cup f\ S_1 \cup \ldots$ *for all chains S.*

To understand why these notions are relevant for us, think in terms of relations between states, or, for simplicity, input and output of some computation. For example, the input-output behaviour of a function *sum* that sums up the first n numbers can be expressed as this infinite relation $Sum = \{(0,0), (1,1), (2,3), (3,6), (4,10), \ldots\}$ on *nat*. We can compute this relation gradually by starting from $\{\}$ and adding more pairs in each step: $\{\} \subseteq \{(0,0)\} \subseteq \{(0,0), (1,1)\} \subseteq \ldots$. This is why chains are relevant. Each element in the chain is only a finite approximation of the full semantics of the summation function which is the infinite set $\bigcup_n S\ n$.

To understand the computational meaning of monotonicity, consider a second summation function sum_2 with semantics $Sum_2 = \{(0, 0), (1, 1), (2, 3)\}$, i.e. sum_2 behaves like *sum* for inputs $\leqslant 2$ and does not terminate otherwise. Let P[.] be a program where we can slot in different subcomponents.

Monotonicity of the semantics of P means that because $Sum_2 \subseteq Sum$, any result that P[sum_2] can deliver, P[sum] can deliver too. This makes computational sense: if P[sum_2] delivers an output, it can only have called sum_2 with arguments $\leqslant 2$ (otherwise the call and thus the whole computation would not have terminated), in which case P[sum] can follow the same computation path in P and deliver the same result.

But why continuity? We give an example where non-computability means non-continuity. Consider the non-computable function $T :: (nat \times nat)\ set \Rightarrow bool\ set$ whose output tells us if its input is (the semantics of) an everywhere terminating computation:

$$T\ r = (if\ \forall m.\ \exists n.\ (m, n) \in r\ then\ \{True\}\ else\ \{\})$$

This function is monotone but not continuous. For example, let S be the chain of finite approximations of Sum above. Then $T\ (\bigcup_n\ S\ n) = \{True\}$ but $(\bigcup_n\ T\ (S\ n)) = \{\}$. Going back to the P[.] scenario above, continuity means that a terminating computation of P[f] should only need a finite part of the semantics of f, which means it can only run f for a finite amount of time. Again, this makes computational sense.

Now we study chains and continuity formally. It is obvious (and provable by induction) that in a chain S we have $i \leqslant j \Longrightarrow S\ i \subseteq S\ j$. But because $\leqslant$ is total on nat ($i \leqslant j \vee j \leqslant i$), $\subseteq$ must be total on S too:

Lemma 11.10. *chain* $S \Longrightarrow S\ i \subseteq S\ j \vee S\ j \subseteq S\ i$

Continuity implies monotonicity:

Lemma 11.11. *Every continuous function is monotone.*

Proof. Let f be continuous, assume $A \subseteq B$ and consider the chain $A \subseteq B \subseteq B \subseteq \ldots$, i.e. $S = (\lambda i.\ if\ i = 0\ then\ A\ else\ B)$. Then $f\ B = f\ (\bigcup_n\ S\ n) = (\bigcup_n f\ (S\ n)) = f\ A \cup f\ B$ and hence $f\ A \subseteq f\ B$. □

Our main theorem about continuous function is that their least fixpoints can be obtained by iteration starting from $\{\}$.

Theorem 11.12 (Kleene fixpoint theorem).
cont $f \Longrightarrow lfp\ f = (\bigcup_n f^n\ \{\})$

Proof. Let $U = (\bigcup_n f^n\ \{\})$. First we show $lfp\ f \subseteq U$, then $U \subseteq lfp\ f$.

Because f is continuous, it is also monotone. Hence $lfp\ f \subseteq U$ follows by Knaster-Tarski if U is a fixpoint of f. Therefore we prove $f\ U = U$. Observe that by Lemma 10.42 the $f^n\ \{\}$ form a chain.

$$\begin{aligned} f\ U &= \bigcup_n f^{n+1}\ \{\} && \text{by continuity} \\ &= f^0\ \{\} \cup (\bigcup_n f^{n+1}\ \{\}) && \text{because } f^0\ \{\} = \{\} \\ &= U \end{aligned}$$

For the opposite direction $U \subseteq \mathit{lfp}\ f$ it suffices to prove $f^n\ \{\} \subseteq \mathit{lfp}\ f$. The proof is by induction on n. The base case is trivial. Assuming $f^n\ \{\} \subseteq \mathit{lfp}\ f$ we have by monotonicity of f and Knaster-Tarski that $f^{n+1}\ \{\} = f\ (f^n\ \{\}) \subseteq f\ (\mathit{lfp}\ f) = \mathit{lfp}\ f$. □

This is a generalisation of Lemma 10.42 that allowed us to compute *lfp* in the true liveness analysis in Section 10.4.2. At the time we could expect the chain of iterates of f to stabilise, now we have to go to the limit.

The Kleene fixpoint theorem is applicable to W:

Lemma 11.13. *Function $W\ b\ r$ is continuous.*

Proof. Although the Isabelle proof is automatic, we explain the details because they may not be obvious. Let $R :: \mathit{nat} \Rightarrow \mathit{com\_den}$ be any sequence of state relations — it does not even need to be a chain. We show that $(s, t) \in W\ b\ r\ (\bigcup_n R\ n)$ iff $(s, t) \in (\bigcup_n W\ b\ r\ (R\ n))$. If $\neg\ b\ s$ then (s, t) is in both sets iff $s = t$. Now assume $b\ s$. Then

$$\begin{aligned}
(s, t) \in W\ b\ r\ (\textstyle\bigcup_n R\ n) &\longleftrightarrow (s, t) \in r \bigcirc (\textstyle\bigcup_n R\ n)\\
&\longleftrightarrow \exists s'.\ (s, s') \in r \wedge (s', t) \in (\textstyle\bigcup_n R\ n)\\
&\longleftrightarrow \exists s'\ n.\ (s, s') \in r \wedge (s', t) \in R\ n\\
&\longleftrightarrow \exists n.\ (s, t) \in r \bigcirc R\ n\\
&\longleftrightarrow (s, t) \in (\textstyle\bigcup_n r \bigcirc R\ n)\\
&\longleftrightarrow (s, t) \in (\textstyle\bigcup_n W\ b\ r\ (R\ n))
\end{aligned}$$

Warning: such $\longleftrightarrow$ chains are an abuse of notation: $A \longleftrightarrow B \longleftrightarrow C$ really means the logically not equivalent $(A \longleftrightarrow B) \wedge (B \longleftrightarrow C)$ which implies $A \longleftrightarrow C$. □

Example 11.14. In this example we show concretely how iterating W creates a chain of relations that approximate the semantics of some loop and whose union is the full semantics. The loop is

$$\begin{aligned}
&\mathit{WHILE}\ b\ \mathit{DO}\ c\\
&\text{where } b = \mathit{Not}(\mathit{Eq}\ (V\ ''x'')\ (N\ 0))\\
&\text{and } \quad c = ''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ -1).
\end{aligned}$$

Function *Eq* compares its arguments for equality (see Exercise 3.7). Intuitively, the semantics of this loop is the following relation:

$$S = \{(s, t).\ 0 \leqslant s\ ''x'' \wedge t = s(''x'' := 0)\}$$

Function F expresses the semantics of one loop iteration:

$$\begin{aligned}
&F = W\ db\ dc\\
&\text{where } db = \mathit{bval}\ b = (\lambda s.\ s\ ''x'' \neq 0)\\
&\text{and } \quad dc = D\ c \quad = \{(s, t).\ t = s(''x'' := s\ ''x'' - 1)\}.
\end{aligned}$$

This is what happens when we iterate F starting from $\{\}$:

$$\begin{aligned} F^1\ \{\} &= \{(s,\ t).\ \mathit{if}\ s\ ''x'' \neq 0\ \mathit{then}\ (s,\ t) \in \mathit{dc} \bigcirc \{\}\ \mathit{else}\ s = t\} \\ &= \{(s,\ t).\ s\ ''x'' = 0 \wedge s = t\} \end{aligned}$$

$$\begin{aligned} F^2\ \{\} &= \{(s,\ t).\ \mathit{if}\ s\ ''x'' \neq 0\ \mathit{then}\ (s,\ t) \in \mathit{dc} \bigcirc F\ \{\}\ \mathit{else}\ s = t\} \\ &= \{(s,\ t).\ 0 \leqslant s\ ''x'' \wedge s\ ''x'' < 2 \wedge t = s(''x'' := 0)\} \end{aligned}$$

$$\begin{aligned} F^3\ \{\} &= \{(s,\ t).\ \mathit{if}\ s\ ''x'' \neq 0\ \mathit{then}\ (s,\ t) \in \mathit{dc} \bigcirc F^2\ \{\}\ \mathit{else}\ s = t\} \\ &= \{(s,\ t).\ 0 \leqslant s\ ''x'' \wedge s\ ''x'' < 3 \wedge t = s(''x'' := 0)\} \end{aligned}$$

Note that the first equality $F^n\ \{\} = R$ follows by definition of W whereas the second equality $R = R'$ requires a few case distinctions. The advantage of the form R' is that we can see a pattern emerging:

$$F^n\ \{\} = \{(s,\ t).\ 0 \leqslant s\ ''x'' \wedge s\ ''x'' < \mathit{int}\ n \wedge t = s(''x'' := 0)\}$$

where function *int* coerces a *nat* to an *int*. This formulation shows clearly that $F^n\ \{\}$ is the semantics of our loop restricted to at most n iterations. The $F^n\ \{\}$ form a chain and the union is the full semantics: $(\bigcup_n F^n\ \{\}) = S$. The latter equality follows because the condition $s\ ''x'' < \mathit{int}\ n$ in $\bigcup_n F^n\ \{\}$ is satisfied by all large enough n and hence $\bigcup_n F^n\ \{\}$ collapses to S.

As an application of the Kleene fixpoint theorem we show that our denotational semantics is deterministic, i.e. the command denotations returned by D are **single-valued**, a predefined notion:

$$\mathit{single\_valued}\ r = (\forall x\ y\ z.\ (x,\ y) \in r \wedge (x,\ z) \in r \longrightarrow y = z)$$

The only difficult part of the proof is the *WHILE* case. Here we can now argue that because W preserves single-valuedness, then its least fixpoint is single-valued because it is a union of a chain of single-valued relations:

Lemma 11.15. *If $f :: \mathit{com\_den} \Rightarrow \mathit{com\_den}$ is continuous and preserves single-valuedness then $\mathit{lfp}\ f$ is single-valued.*

Proof. Because $\mathit{cont}\ f$ we have $\mathit{lfp}\ f = (\bigcup_n f^n\ \{\})$. By Lemma 10.42 (because f is also monotone) the $f^n\ \{\}$ form a chain. All its elements are single-valued because $\{\}$ is single-valued and f preserves single-valuedness. The union of a chain of single-valued relations is obviously single-valued too. □

Lemma 11.16. $\mathit{single\_valued}\ (D\ c)$

Proof. A straightforward induction on c. The *WHILE* case follows by the previous lemma because $W\ b\ r$ is continuous and preserves single-valuedness (as can be checked easily, assuming by IH that r is single-valued). □

11.2 Summary and Further Reading

A denotational semantics is a compositional function from syntactic objects to their meaning. Compositional means that the meaning of a compound construct is a function of the meaning of its subconstructs. The meaning of iterative or recursive constructs is given as a least fixpoint. In a relational context, the existence of a least fixpoint can be guaranteed by monotonicity via the Knaster-Tarski Fixpoint Theorem. For computational reasons the semantics should not just be monotone but even continuous, in which case the least fixpoint is the union of all finite iterates $f^n\ \{\}$. Thus we can argue about least fixpoints of continuous functions by induction on n.

Denotational semantics has its roots in the work by Dana Scott and Christopher Strachey [83, 84]. This developed into a rich and mathematically sophisticated theory. We have only presented a simplified set-theoretic version of the foundations. For the real deal the reader is encouraged to consult textbooks [40, 80] and handbook articles [1, 87] dedicated to denotational semantics. Some of the foundations of denotational semantics have been formalised in theorem provers [12, 44, 61].

Exercises

Exercise 11.17. Building on Exercise 7.23, extend the denotational semantics and the equivalence proof with the big-step semantics with $REPEAT\ c\ UNTIL\ b$.

Exercise 11.18. Building on Exercise 7.24, extend the denotational semantics and the equivalence proof with the big-step semantics with nondeterministic choice.

Exercise 11.19. Consider Example 11.14 but with the loop condition $b = Less\ (N\ 0)\ (V\ ''x'')$. Find a closed expression for $F^n\ \{\}$.

Exercise 11.20. Consider Example 11.14 and prove the equation $F^n\ \{\} = \{(s, t).\ 0 \leqslant s\ ''x'' \wedge s\ ''x'' < int\ n \wedge t = s(''x'' := 0)\}$ by induction.

Exercise 11.21. Let the 'thin' part of a relation be its single-valued subset:

$$Thin\ R = \{(a, b).\ (a, b) \in R \wedge (\forall c.\ (a, c) \in R \longrightarrow c = b)\}$$

Prove that if $F :: ('a * 'a)\ set \Rightarrow ('a * 'a)\ set$ is monotone and for all R, $F\ (Thin\ R) \subseteq Thin\ (F\ R)$, then $single\_valued\ (lfp\ F)$.

Exercise 11.22. Generalise our set-theoretic treatment of continuity and least fixpoints to chain-complete partial orders (cpos), i.e. partial orders $\leqslant$

where every chain $x_0 \leqslant x_1 \leqslant \ldots$ has a least upper bound. Prove Kleene's theorem for cpos instead of sets. It is recommended to formalise cpos as a type class (see Section 13.4.1).

12

Hoare Logic

So far we have proved properties of IMP, like type soundness, or properties of tools for IMP, like compiler correctness, but almost never properties of individual IMP programs. The Isabelle part of the book has taught us how to prove properties of functional programs, but not of imperative ones.

Hoare logic (due to Sir Tony Hoare), also known as **axiomatic semantics**, is a logic for proving properties of imperative programs. The formulas of Hoare logic are so-called **Hoare triples**

$$\{P\}\ c\ \{Q\}$$

which should be read as saying that if formula P is true before the execution of command c then formula Q is true after the execution of c. This is a simple example of a Hoare triple:

```
{x = y} y := y+1 {x < y}
```

12.1 Proof via Operational Semantics thy

Before introducing the details of Hoare logic we show that in principle we can prove properties of programs via their operational semantics. Hoare logic can be viewed as the structured essence of such proofs.

As an example, we prove that the program

```
y := 0;
WHILE 0 < x DO (y := y+x; x := x-1)
```

sums up the numbers 1 to x in y. Formally let

$$wsum = \mathit{WHILE}\ \mathit{Less}\ (N\ 0)\ (V\ ''x'')\ \mathit{DO}\ \mathit{csum}$$

$$\begin{aligned} \mathit{csum} = {} & ''y'' ::= \mathit{Plus}\ (V\ ''y'')\ (V\ ''x'');; \\ & ''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ -1) \end{aligned}$$

The summation property can be expressed as a theorem about the program's big-step semantics:

$$(''y'' ::= N\ 0;;\ wsum,\ s) \Rightarrow t \implies t\ ''y'' = sum\ (s\ ''x'') \qquad (*)$$

where

fun $sum :: int \Rightarrow int$ **where**
$sum\ i = (if\ i \leqslant 0\ then\ 0\ else\ sum\ (i - 1) + i)$

We prove $(*)$ in two steps. First we show that $wsum$ does the right thing. This will be an induction, and as usual we have to generalise what we want to prove from the special situation where y is 0.

$$(wsum,\ s) \Rightarrow t \implies t\ ''y'' = s\ ''y'' + sum\ (s\ ''x'') \qquad (**)$$

This is proved by an induction on the premise. There are two cases.

If the loop condition is false, i.e. if $s\ ''x'' \leqslant 0$, then we have to prove $s\ ''y'' = s\ ''y'' + sum\ (s\ ''x'')$, which follows because $s\ ''x'' < 1$ implies $sum\ (s\ ''x'') = 0$.

If the loop condition is true, i.e. if $0 < s\ ''x''$, then we may assume $(csum,\ s) \Rightarrow u$ and the IH $t\ ''y'' = u\ ''y'' + sum\ (u\ ''x'')$ and we have to prove the conclusion of $(**)$. From $(csum,\ s) \Rightarrow u$ it follows by inversion of the rules for ;; and ::= (Section 7.2.3) that $u = s(''y'' := s\ ''y'' + s\ ''x'',\ ''x'' := s\ ''x'' - 1)$. Substituting this into the IH yields $t\ ''y'' = s\ ''y'' + s\ ''x'' + sum\ (s\ ''x'' - 1)$. This is equivalent with the conclusion of $(**)$ because $0 < s\ ''x''$.

Having proved $(**)$, $(*)$ follows easily: From $(''y'' ::= N\ 0;;\ wsum,\ s) \Rightarrow t$ rule inversion for ;; and ::= shows that after the assignment the intermediate state must have been $s(''y'' := 0)$ and therefore $(wsum,\ s(''y'' := 0)) \Rightarrow t$. Now $(**)$ implies $t\ ''y'' = sum\ (s\ ''x'')$, thus concluding the proof of $(*)$.

Hoare logic can be viewed as the structured essence of such operational proofs. The rules of Hoare logic are (almost) syntax directed and automate all those aspects of the proof that are concerened with program execution. However, there is no free lunch: you still need to be creative to find generalisations of formulas when proving properties of loops, and proofs about arithmetic formulas are still up to you (and Isabelle).

We will now move on to the actual topic of this chapter, Hoare logic.

12.2 Hoare Logic for Partial Correctness

The formulas of Hoare logic are the Hoare triples $\{P\}\ c\ \{Q\}$, where P is called the **precondition** and Q the **postcondition**. We call $\{P\}\ c\ \{Q\}$ **valid** if the following implication holds:

If P is true before the execution of c and c terminates
then Q is true afterwards.

Validity is defined in terms of execution, i.e. the operational semantics, our ultimate point of reference. This notion of validity is called **partial correctness** because the postcondition is only required to be true if c terminates. There is also the concept of **total correctness**:

If P is true before the execution of c
then c terminates and Q is true afterwards.

In a nutshell:

Total correctness = partial correctness + termination

Except for the final section of this chapter, we will always work with the partial correctness interpretation, because it is easier.

Pre and postconditions come from some set of logical formulas that we call **assertions**. There are two approaches to the language of assertions:

Syntactic: Assertions are concrete syntactic objects, just like type *bexp*.
Semantic: Assertions are predicates on states, i.e. of type $state \Rightarrow bool$.

We follow the syntactic approach in the introductory subsection, because it is nicely intuitive and one can sweep some complications under the carpet. But for proofs about Hoare logic, the semantic approach is much simpler and we switch to it in the rest of the chapter.

12.2.1 Syntactic Assertions

Assertions are ordinary logical formulas and include all the boolean expressions of IMP. We are deliberately vague because the exact nature of assertions is not important for understanding how Hoare logic works. Just like the simplified notation for IMP we write concrete assertions and Hoare triples in typewriter font. Here are some examples of valid Hoare triples:

`{x = 5} x := x+5 {x = 10}`

`{True} x := 10 {x = 10}`

`{x = y} x := x+1 {x` $\neq$ `y}`

Note that the precondition `True` is always true, hence the second triple merely says that from whatever state you start, after `x := 10` the postcondition `x = 10` is true.

More interesting are the following somewhat extreme examples:

`{True}` c_1 `{True}`

{True} c_2 {False}

{False} c_3 {Q}

Which c_i make these triples valid? Think about it before you read on. Remember that we work with partial correctness in this section. Therefore every c_1 works because the postcondition `True` is always true. In the second triple, c_2 must not terminate, otherwise `False` would have to be true, which it certainly is not. In the final triple, any c_3 and Q work because the meaning of the triple is that "if `False` is true ...", but `False` is not true. Note that for the first two triples, the answer is different under a total correctness interpretation.

Proof System

So far we have spoken of Hoare triples being valid. Now we will present a set of inference rules or **proof system** for deriving Hoare triples. This is a new mechanism and here we speak of Hoare triples being **derivable**. Of course being valid and being derivable should have something to do with each other. When we look at the proof rules in a moment they will all feel very natural (well, except for one) precisely because they follow our informal understanding of when a triple is valid. Nevertheless it is essential not to confuse the notions of validity (which employs the operational semantics) and derivability (which employs an independent set of proof rules).

The proof rules for Hoare logic are shown in Figure 12.1. We go through them one by one.

The *SKIP* rule is obvious, but the assignment rule needs some explanation. It uses the substitution notation

$P[a/x] \equiv P$ with a substituted for x.

For example, `(x = 5)[5/x]` is `5 = 5` and `(x = x)[5+x/x]` is `5+x = 5+x`. The latter example shows that all occurrences of x in P are simultaneously replaced by a, but that this happens only once: if x occurs in a, those occurrences are not replaced, otherwise the substitution process would go on forever. Here are some instances of the assignment rule:

```
          {5 = 5} x := 5        {x = 5}
       {x+5 = 5} x := x+5       {x = 5}
{2*(x+5) > 20} x := 2*(x+5) {x > 20}
```

Simplifying the preconditions that were obtained by blind substitution yields the more readable triples

```
 {True} x := 5        {x = 5}
{x = 0} x := x+5      {x = 5}
{x > 5} x := 2*(x+5) {x > 20}
```

$$\frac{}{\{P\}\ SKIP\ \{P\}}$$

$$\frac{}{\{P[a/x]\}\ x ::= a\ \{P\}}$$

$$\frac{\{P_1\}\ c_1\ \{P_2\} \quad \{P_2\}\ c_2\ \{P_3\}}{\{P_1\}\ c_1;;\ c_2\ \{P_3\}}$$

$$\frac{\{P \wedge b\}\ c_1\ \{Q\} \quad \{P \wedge \neg\ b\}\ c_2\ \{Q\}}{\{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}}$$

$$\frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ WHILE\ b\ DO\ c\ \{P \wedge \neg\ b\}}$$

$$\frac{P' \longrightarrow P \quad \{P\}\ c\ \{Q\} \quad Q \longrightarrow Q'}{\{P'\}\ c\ \{Q'\}}$$

Fig. 12.1. Hoare logic for partial correctness (syntactic assertions)

The assignment rule may still puzzle you because it seems to go in the wrong direction by modifying the precondition rather than the postcondition. After all, the operational semantics modifies the post-state, not the pre-state. Correctness of the assignment rule can be explained as follows: if initially $P[a/x]$ is true, then after the assignment x will have the value of a, and hence no substitution is necessary anymore, i.e. P itself is true afterwards. A forward version of this rule exists but is more complicated.

The ;; rule strongly resembles its big-step counterpart. Reading it backward it decomposes the proof of $c_1;;\ c_2$ into two proofs involving c_1 and c_2 and a new intermediate assertion P_2.

The *IF* rule is pretty obvious: you need to prove that both branches lead from P to Q, were in each proof the appropriate b or $\neg\ b$ can be conjoined to P. That is, each sub-proof additionally assumes the branch condition.

Now we consider the *WHILE* rule. Its premise says that if P and b are true before the execution of the loop body c, then P is true again afterwards (if the body terminates). Such a P is called an **invariant** of the loop: if you start in a state where P is true then no matter how often the loop body is iterated, as long as b is true before each iteration, P stays true too. This explains the conclusion: if P is true initially, then it must be true at the end because it is invariant. Moreover, if the loop terminates, then $\neg\ b$ must be

true too. Hence $P \wedge \neg\ b$ at the end (if we get there). The *WHILE* rule can be viewed as an induction rule where the invariance proof is the step.

The final rule in Figure 12.1 is called the **consequence** rule. It is independent of any particular IMP construct. Its purpose is to adjust the precondition and postcondition. Going from $\{P\}\ c\ \{Q\}$ to $\{P'\}\ c\ \{Q'\}$ under the given premises permits us to

- strengthen the precondition: $P' \longrightarrow P$
- weaken the postcondition: $Q \longrightarrow Q'$

where A is called stronger than B if $A \longrightarrow B$. For example, from $\{x \geqslant 0\}\ c\ \{x \geqslant 1\}$ we can prove $\{x = 5\}\ c\ \{x \geqslant 0\}$. The latter is strictly weaker than the former because it tells us less about the behaviour of c. Note that the consequence rule is the only rule where some premises are not Hoare triples but assertions. We do not have a formal proof system for assertions and rely on our informal understanding of their meaning to check, for example, that $x = 5 \longrightarrow x \geqslant 0$. This informality will be overcome once we consider assertions as predicates on states.

This completes the discussion of the basic proof rules. Although these rules are sufficient for all proofs, i.e. the system is complete (which we show later), the rules for *SKIP*, $::=$ and *WHILE* are inconvenient: they can only be applied backwards if the pre or postcondition are of a special form. For example for *SKIP*, they need to be identical. Therefore we derive new rules for those constructs that can be applied backwards irrespective of the pre and postcondition of the given triple. The new rules are shown in Figure 12.2. They are easily derived by combining the old rules with the consequence rule.

$$\frac{P \longrightarrow Q}{\{P\}\ SKIP\ \{Q\}}$$

$$\frac{P \longrightarrow Q[a/x]}{\{P\}\ x ::= a\ \{Q\}}$$

$$\frac{\{P \wedge b\}\ c\ \{P\} \qquad P \wedge \neg\ b \longrightarrow Q}{\{P\}\ WHILE\ b\ DO\ c\ \{Q\}}$$

Fig. 12.2. Derived rules (syntactic assertions)

Here is one of the derivations:

$$\frac{P \longrightarrow Q[a/x] \qquad \overline{\{Q[a/x]\}\ x ::= a\ \{Q\}} \qquad \overline{Q \longrightarrow Q}}{\{P\}\ x ::= a\ \{Q\}}$$

Two of the three premises are overlined because they have been proved, namely with the original assignment rule and with the trivial logical fact that anything implies itself.

Examples

We return to the summation program from Section 12.1. This time we prove it correct by means of Hoare logic rather than operational semantics. In Hoare logic, we want to prove the triple

$$\texttt{\{x = i\} y := 0; wsum \{y = sum i\}}$$

We cannot write `y = sum x` in the postcondition because `x` is 0 at that point. Unfortunately the postcondition cannot refer directly to the initial state. Instead, the precondition `x = i` allows us to refer to the unchanged `i` and therefore to the initial value of `x` in the postcondition. This is a general trick for remembering values of variables that are modified.

The central part of the proof is to find and prove the invariant `I` of the loop. Note that we have three constraints that must be satisfied:

1. It should be an invariant: $\texttt{\{I} \wedge \texttt{0 < x\} csum \{I\}}$
2. It should imply the postcondition: $\texttt{I} \wedge \neg\, \texttt{0 < x} \longrightarrow \texttt{y = sum i}$
3. The invariant should be true initially: $\texttt{x = i} \wedge \texttt{y = 0} \longrightarrow \texttt{I}$

In fact, this is a general design principle for invariants. As usual, it is a case of generalising the desired postcondition. During the iteration, `y = sum i` is not quite true yet because the first `x` numbers are still missing from `y`. Hence we try the following assertion:

$$\texttt{I} = (\texttt{y + sum x = sum i})$$

It is easy to check that the constraints 2 and 3 are true. Moreover, `I` is indeed invariant as the following proof tree shows:

$$\dfrac{\dfrac{\overline{\texttt{I} \wedge \texttt{0 < x} \longrightarrow \texttt{I[x-1/x][y+x/y]}}}{\texttt{\{I} \wedge \texttt{0 < x\} y := y+x \{I[x-1/x]\}}} \qquad \overline{\texttt{\{I[x-1/x]\} x := x-1 \{I\}}}}{\texttt{\{I} \wedge \texttt{0 < x\} csum \{I\}}}$$

Although we have not given the proof rules names, it is easy to see at any point in the proof tree which one is used. In the above tree, the left assignment is proved with the derived rule, the right assignment with the basic rule.

In case you are wondering why $\texttt{I} \wedge \texttt{0 < x} \longrightarrow \texttt{I[x-1/x][y+x/y]}$ is true, expand the definition of `I` and carry out the substitutions and you arrive at the following easy arithmetic truth:

$$\texttt{y + sum x = sum i} \wedge \texttt{0 < x} \longrightarrow \texttt{y + x + sum(x-1) = sum i} \quad (12.1)$$

With the loop rule and some more arithmetic we derive

$$\frac{\texttt{\{I} \wedge \texttt{0 < x\} csum \{I\}} \qquad \overline{\texttt{I} \wedge \neg\ \texttt{0 < x} \longrightarrow \texttt{y = sum i}}}{\texttt{\{I\} wsum \{y = sum i\}}}$$

Now we only need to connect this result with the initialisation to obtain the correctness proof for `y := 0; wsum`:

$$\frac{\dfrac{\overline{\texttt{x = i} \longrightarrow \texttt{I[0/y]}}}{\texttt{\{x = i\} y := 0 \{I\}}} \qquad \texttt{\{I\} wsum \{y = sum i\}}}{\texttt{\{x = i\} y := 0; wsum \{y = sum i\}}}$$

The summation program is special in that it always terminates. Hence it does not demonstrate that the proof system can prove anything about nonterminating computations, like in the following example:

$$\frac{\overline{\texttt{\{True\} SKIP \{True\}}} \qquad \overline{\texttt{True} \wedge \neg\ \texttt{True} \longrightarrow Q}}{\texttt{\{True\} WHILE True DO SKIP \{}Q\texttt{\}}}$$

We have proved that if the loop terminates, any assertion Q is true. It sounds like magic but is merely the consequence of nontermination. The proof is straightforward: the invarint is `True` and Q is trivially implied by the negation of the loop condition.

As a final example consider swapping two variables:

$$\texttt{\{P\} h := x; x := y; y := h \{Q\}}$$

where $\texttt{P} = (\texttt{x = a} \wedge \texttt{y = b})$ and $\texttt{Q} = (\texttt{x = b} \wedge \texttt{y = a})$. Drawing the full proof tree for this triple is tedious and unnecessary. A compact form of the tree can be given by annotating the intermediate program points with the correct assertions:

$$\texttt{\{P\} h := x; \{Q[h/y][y/x]\} x := y; \{Q[h/y]\} y := h \{Q\}}$$

Both `Q[h/y]` and `Q[h/y][y/x]` are simply the result of the basic assignment rule. All that is left to check is the first assignment with the derived assignment rule, i.e. check $\texttt{P} \longrightarrow \texttt{Q[h/y][y/x][x/h]}$. This is true because $\texttt{Q[h/y][y/x][x/h]} = (\texttt{y = b} \wedge \texttt{x = a})$.

It should be clear that this proof procedure works for any sequence of assignments, thus reducing the proof to pulling back the postcondition (which is completely mechanical) and checking an implication.

The Method

If we look at the proof rules and the examples it becomes apparent that there is a method in this madness: the backward construction of Hoare logic proofs is partly mechanical. Here are the key points:

- We only need the original rules for ;; and *IF* together with the derived rules for *SKIP*, ::= and *WHILE*. This is a syntax directed proof system and each backward rule application creates new subgoals for the subcommands. Thus the shape of the proof tree exactly mirrors the shape of the command in the Hoare triple we want to prove. The construction of the skeleton of this proof tree is completely automatic.
  The consequence rule is built into the derived rules and is not required anymore. This is crucial: the consequence rule destroys syntax directedness because it can be applied at any point.
- When applying the ;; rule backwards we need to provide the intermediate assertion P_2 that occurs in the premises but not the conclusion. It turns out that we can compute P_2 by pulling the final assertion P_3 back through c_2. The variable swapping example illustrates this principle.
- There are two aspects that cannot be fully automated (or progam verification would be completely automatic, which is impossible): invariants must be supplied explicitly, and the implications between assertions in the premises of the derived rules must be proved somehow.

In a nutshell, Hoare logic can be reduced to finding invariants and proving assertions. We will carry out this program in full detail in Section 12.4. But first we need to formalise our informal notion of assertions.

12.2.2 Assertions as Functions thy

Our introduction to Hoare logic so far was informal with an emphasis on intuition. Now we formalise assertions as predicates on states:

type_synonym *assn* = *state* ⇒ *bool*

As an example of the simplicity of this approach we define validity formally:

$$\models \{P\}\ c\ \{Q\} \longleftrightarrow (\forall s\ t.\ P\ s \wedge (c, s) \Rightarrow t \longrightarrow Q\ t)$$

We pronounce $\models \{P\}\ c\ \{Q\}$ as "$\{P\}\ c\ \{Q\}$ is valid".

Hoare logic with functional assertions is defined as an inductive predicate with the syntax $\vdash \{P\}\ c\ \{Q\}$ which is read as "$\{P\}\ c\ \{Q\}$ is derivable/provable" (in Hoare logic). The rules of the inductive definition are shown in Figure 12.3, two derived rules in Figure 12.4. These rules are a direct translation of the

$$\frac{}{\vdash \{P\}\ SKIP\ \{P\}}\ Skip$$

$$\frac{}{\vdash \{\lambda s.\ P\ (s[a/x])\}\ x ::= a\ \{P\}}\ Assign$$

$$\frac{\vdash \{P\}\ c_1\ \{Q\} \qquad \vdash \{Q\}\ c_2\ \{R\}}{\vdash \{P\}\ c_1;;\ c_2\ \{R\}}\ Seq$$

$$\frac{\vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c_1\ \{Q\} \qquad \vdash \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}\ c_2\ \{Q\}}{\vdash \{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}}\ If$$

$$\frac{\vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c\ \{P\}}{\vdash \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}}\ While$$

$$\frac{\forall s.\ P'\ s \longrightarrow P\ s \qquad \vdash \{P\}\ c\ \{Q\} \qquad \forall s.\ Q\ s \longrightarrow Q'\ s}{\vdash \{P'\}\ c\ \{Q'\}}\ conseq$$

Fig. 12.3. Hoare logic for partial correctness (functional assertions)

syntactic ones, taking into account that assertions are predicates on states. Only rule *Assign* requires some explanation. The notation $s[a/x]$ is merely an abbreviation that mimics syntactic substitution into assertions:

$$s[a/x] \equiv s(x := aval\ a\ s)$$

What does our earlier `P[a/x]` have to do with $P(s[a/x])$? We have not formalised the syntax of assertions, but we can explain what is going on at the level of their close relatives, boolean expressions. Assume we have a substitution function *bsubst* such that *bsubst b a x* corresponds to `b[a/x]`, i.e. substitutes a for x in b. Then we can prove

Lemma 12.1 (Substitution lemma).
bval (*bsubst b a x*) s = *bval b* $(s[a/x])$

It expresses that as far as evaluation is concerened, it does not matter if you substitute into the expression or into the state.

12.2.3 Example Proof thy

We can now perform Hoare logic proofs in Isabelle. For that purpose we go back to the **apply**-style because it allows us to perform such proofs without having to type in the myriad of intermediate assertions. Instead they are

$$\frac{\forall s.\ P\ s \longrightarrow Q\ (s[a/x])}{\vdash \{P\}\ x ::= a\ \{Q\}}\ Assign'$$

$$\frac{\vdash \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c\ \{P\} \qquad \forall s.\ P\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s}{\vdash \{P\}\ WHILE\ b\ DO\ c\ \{Q\}}\ While'$$

Fig. 12.4. Derived rules (functional assertions)

computed by rule application, except for the invariants. As an example we verify *wsum* (Section 12.1) once more:

lemma $\vdash \{\lambda s.\ s\ ''x'' = i\}\ ''y'' ::= N\ 0;;\ wsum\ \{\lambda s.\ s\ ''y'' = sum\ i\}$

Rule *Seq* creates two subgoals:

apply(*rule Seq*)

1. $\vdash \{\lambda s.\ s\ ''x'' = i\}\ ''y'' ::= N\ 0\ \{?Q\}$
2. $\vdash \{?Q\}\ wsum\ \{\lambda s.\ s\ ''y'' = sum\ i\}$

As outlined in The Method above, we left the intermediate assertion open and will instantiate it by working on the second subgoal first (**prefer** 2). Since that is a loop, we have to provide it anyway, because it is the invariant:

prefer 2
apply(*rule While'*[**where** $P = \lambda s.\ (s\ ''y'' = sum\ i - sum\ (s\ ''x''))$])

We have rearranged the earlier invariant `y + sum x = sum i` slightly to please the simplifier.

The first subgoal of *While'* is preservation of the invariant:

1. $\vdash \{\lambda s.\ s\ ''y'' = sum\ i - sum\ (s\ ''x'') \wedge$
   $bval\ (Less\ (N\ 0)\ (V\ ''x''))\ s\}$
   $csum\ \{\lambda s.\ s\ ''y'' = sum\ i - sum\ (s\ ''x'')\}$

A total of 3 *subgoals...*

Because *csum* stands for a sequential composition we proceed as above:

apply(*rule Seq*)
prefer 2

1. $\vdash \{?Q6\}\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ -1)$
   $\{\lambda s.\ s\ ''y'' = sum\ i - sum\ (s\ ''x'')\}$
2. $\vdash \{\lambda s.\ s\ ''y'' = sum\ i - sum\ (s\ ''x'') \wedge$
   $bval\ (Less\ (N\ 0)\ (V\ ''x''))\ s\}$
   $''y'' ::= Plus\ (V\ ''y'')\ (V\ ''x'')\ \{?Q6\}$

A total of 4 *subgoals...*

Now the two assignment rules (basic and derived) do their job.

apply(*rule Assign*)
apply(*rule Assign'*)

The resulting subgoal is large and hard to read because of the substitutions; therefore we do not show it. It corresponds to (12.1) and can be proved by *simp* (not shown). We move on to the second premise of *While'*, the proof that at the exit of the loop the required postcondition is true:

1. $\forall s.\ s\ ''y'' = sum\ i - sum\ (s\ ''x'') \wedge$
   $\neg\ bval\ (Less\ (N\ 0)\ (V\ ''x''))\ s \longrightarrow$
   $s\ ''y'' = sum\ i$

A total of 2 subgoals...

This is proved by *simp* and all that is left is the initialisation.

1. $\vdash \{\lambda s.\ s\ ''x'' = i\}\ ''y'' ::= N\ 0$
   $\{\lambda s.\ s\ ''y'' = sum\ i - sum\ (s\ ''x'')\}$

apply(*rule Assign'*)

The resulting subgoal shows a simple example of substitution into the state:

1. $\forall s.\ s\ ''x'' = i \longrightarrow$
   $(s[N\ 0/''y''])\ ''y'' = sum\ i - sum\ ((s[N\ 0/''y''])\ ''x'')$

The proof is again a plain *simp*.

Functional assertions lead to more verbose statements. For the verification of larger programs one would add some Isabelle syntax magic to make functional assertions look more like syntactic ones. We have refrained from that because our emphasis is on explaining Hoare logic rather than verifying concrete programs.

12.2.4 Exercises

Exercise 12.2. For The Method to work as sketched above we need to generalise rule *While'* because it requires the precondition P to be the invariant, which need not be the case. Derive the following rule:

$$\frac{\forall s.\ P\ s \longrightarrow I\ s \qquad \vdash \{\lambda s.\ I\ s \wedge bval\ b\ s\}\ c\ \{I\} \qquad \forall s.\ I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s}{\vdash \{P\}\ WHILE\ b\ DO\ c\ \{Q\}}$$

Rule *If* needs to be generalised to allow us to compute the precondition from the preconditions of the two branches. Derive the following rule:

$$\frac{\vdash \{P_1\}\ c_1\ \{Q\} \qquad \vdash \{P_2\}\ c_2\ \{Q\} \qquad \forall s.\ P\ s \longrightarrow (bval\ b\ s \longrightarrow P_1\ s) \wedge (\neg\ bval\ b\ s \longrightarrow P_2\ s)}{\vdash \{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}}$$

Exercise 12.3. Let $wsum2 = WHILE\ Not(Eq\ (V\ ''x'')\ (N\ 0))\ DO\ csum$ where $bval\ (Eq\ a_1\ a_2)\ s = (aval\ a_1 = aval\ a_2)$ (see Exercise 3.7). Prove $\vdash \{\lambda s.\ s\ ''x'' = i \wedge 0 \leqslant i\}\ ''y'' ::= N\ 0;;\ wsum2\ \{\lambda s.\ s\ ''y'' = sum\ i\}$.

Exercise 12.4. Let $wminusp = WHILE\ Less\ (N\ 0)\ (V\ ''x'')\ DO\ (''x'' ::= Plus\ (V\ ''x'')\ (N\ -1);;\ ''y'' ::= Plus\ (V\ ''y'')\ (N\ -1))$. Prove $\vdash \{\lambda s.\ s\ ''x'' = x \wedge s\ ''y'' = y \wedge 0 \leqslant x\}\ wminusp\ \{\lambda s.\ s\ ''y'' = y - x\}$.

Exercise 12.5. Prove $\vdash \{\lambda s.\ s\ ''x'' = x \wedge s\ ''y'' = y\}\ iminus\ \{\lambda s.\ s\ ''y'' = y - x\}$ for a suitably defined command *iminus*.

Exercise 12.6. Prove $\vdash \{P\}\ c\ \{\lambda s.\ True\}$ (by induction on c).

Exercise 12.7. Give a concrete counterexample to this naive version of the assignment rule: $\{P\}\ x ::= a\ \{P[a/x]\}$.

Exercise 12.8. Define *bsubst* and prove ths Substitution Lemma 12.1. This may require a similar definition and proof for *aexp*.

12.3 Soundness and Completeness thy

So far we have motivated the rules of Hoare logic by operational semantics considerations but we have not proved a precise link between the two. We will now prove

Soundness of the logic w.r.t. the operational semantics:
if a triple is derivable, it is also valid.

Completeness of the logic w.r.t. the operational semantics:
if a triple is valid, it is also derivable.

Recall the definition of validity:

$$\models \{P\}\ c\ \{Q\} \iff (\forall s\ t.\ P\ s \wedge (c, s) \Rightarrow t \longrightarrow Q\ t)$$

Soundness is straightforward:

Lemma 12.9 (Soundness of Hoare logic for partial correctness).
$\vdash \{P\}\ c\ \{Q\} \implies \models \{P\}\ c\ \{Q\}$

Proof. By induction on the derivation of $\vdash \{P\}\ c\ \{Q\}$: we must show that every rule of the logic preserves validity. This is automatic for all rules except *While*, because these rules resemble their big-step counterparts. Even assignment is easy: To prove $\models \{\lambda s.\ P\ (s[a/x])\}\ x ::= a\ \{P\}$ we may assume $P\ (s[a/x])$ and $(x ::= a,\ s) \Rightarrow t$. Therefore $t = s[a/x]$ and thus $P\ t$ as required.

The only other rule we consider is *While*. We may assume the IH $\models \{\lambda s.\ P\ s \wedge bval\ b\ s\}\ c\ \{P\}$. First we prove for arbitrary s and t that if $(WHILE\ b\ DO\ c,\ s) \Rightarrow t$ then $P\ s \Longrightarrow P\ t \wedge \neg\ bval\ b\ t$ by induction on the assumption. There are two cases. If $\neg\ bval\ b\ s$ and $s = t$ then $P\ s \Longrightarrow P\ t \wedge \neg\ bval\ b\ t$ is trivial. If $bval\ b\ s$, $(c,\ s) \Rightarrow s'$ and IH $P\ s' \Longrightarrow P\ t \wedge \neg\ bval\ b\ t$, then we assume $P\ s$ and prove $P\ t \wedge \neg\ bval\ b\ t$. Because $P\ s$, $bval\ b\ s$ and $(c,\ s) \Rightarrow s'$, the outer IH yields $P\ s'$ and then the inner IH yields $P\ t \wedge \neg\ bval\ b\ t$, thus finishing the inner induction. Returning to the *While* rule we need to prove $\models \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}$. Assuming $P\ s$ and $(WHILE\ b\ DO\ c,\ s) \Rightarrow t$, the lemma we just proved locally yields the required $P\ t \wedge \neg\ bval\ b\ t$, thus finishing the *While* rule. □

Soundness was straightforward, as usual: one merely has to plough through the rules one by one. Completness requires new ideas.

12.3.1 Completeness

Completeness, i.e. $\models \{P\}\ c\ \{Q\} \Longrightarrow \vdash \{P\}\ c\ \{Q\}$, is proved with the help of the notion of the **weakest precondition** (in the literature often called the **weakest liberal precondition**):

definition $wp :: com \Rightarrow assn \Rightarrow assn$ **where**
$wp\ c\ Q = (\lambda s.\ \forall t.\ (c,\ s) \Rightarrow t \longrightarrow Q\ t)$

Thinking of assertions as sets of states, this says that the weakest precondition of a command c and a postcondition Q is the set of all pre-states such that if c terminates one ends up in Q.

It is easy to see that $wp\ c\ Q$ is indeed the weakest precondition:

Fact 12.10.
1. $\models \{wp\ c\ Q\}\ c\ \{Q\}$
2. $\models \{P\}\ c\ \{Q\} \implies \forall s.\ P\ s \longrightarrow wp\ c\ Q\ s$

The weakest preconditions is a central concept in Hoare logic because it formalizes the idea of pulling a postcondition back through a command to obtain the corresponding precondition. This idea is central to the construction of Hoare logic proofs and we have already alluded to it multiple times.

The following nice recursion equations hold for *wp*. They can be used to compute the weakest precondition, except for *WHILE*, which would lead to nontermination.

$$
\begin{aligned}
&wp\ SKIP\ Q \quad\ = Q\\
&wp\ (x ::= a)\ Q = (\lambda s.\ Q\ (s[a/x]))\\
&wp\ (c_1;;\ c_2)\ Q = wp\ c_1\ (wp\ c_2\ Q)\\
&wp\ (IF\ b\ THEN\ c_1\ ELSE\ c_2)\ Q\\
&= (\lambda s.\ \mathit{if}\ bval\ b\ s\ \mathit{then}\ wp\ c_1\ Q\ s\ \mathit{else}\ wp\ c_2\ Q\ s)\\
&wp\ (WHILE\ b\ DO\ c)\ Q\\
&= (\lambda s.\ \mathit{if}\ bval\ b\ s\ \mathit{then}\ wp\ (c;;\ WHILE\ b\ DO\ c)\ Q\ s\ \mathit{else}\ Q\ s)
\end{aligned}
$$

Proof. All equations are easily proved from the definition of *wp* once you realise that they are equations between functions. Such equations can be proved with the help of **extensionality**, one of the basic rules of HOL:

$$\frac{\bigwedge x.\ f\ x = g\ x}{f = g}\ ext$$

It expresses that two functions are equal if they are equal for all arguments. For example, we can prove $wp\ SKIP\ Q = Q$ by proving $wp\ SKIP\ Q\ s = Q\ s$ for an arbitrary s. Expanding the definition of *wp* we have to prove $(\forall t.\ (SKIP, s) \Rightarrow t \longrightarrow Q\ t) = Q\ s$ which follows from the inversion of the big-step rule for *SKIP*. The proof of the other equations is similar. □

The key property of *wp* is that is also a precondition w.r.t. provability:

Lemma 12.11. $\vdash \{wp\ c\ Q\}\ c\ \{Q\}$

Proof. By induction on c. We consider only the *WHILE* case, the other cases are automatic (with the help of the *wp* equations). Let $w = WHILE\ b\ DO\ c$. We show $\vdash \{wp\ w\ Q\}\ w\ \{Q\}$ by an application of rule *While′*. Its first premise is $\vdash \{\lambda s.\ wp\ w\ Q\ s \wedge bval\ b\ s\}\ c\ \{wp\ w\ Q\}$. It follows from the IH $\vdash \{wp\ c\ R\}\ c\ \{R\}$ (for any R) where we set $R = wp\ w\ Q$, by precondition strengthening: the implication $wp\ w\ Q\ s \wedge bval\ b\ s \longrightarrow wp\ c\ (wp\ w\ Q)\ s$ follows from the *wp* equations for *WHILE* and ;;. The second premise we need to prove is $wp\ w\ Q\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$; it follows from the *wp* equation for *WHILE*. □

The completeness theorem is an easy consequence:

Theorem 12.12 (Completeness of $\vdash$ w.r.t. $\models$).
$\models \{P\}\ c\ \{Q\} \implies \vdash \{P\}\ c\ \{Q\}$

Proof. Because *wp* is the weakest precondition (Fact 12.10), $\models \{P\}\ c\ \{Q\}$ implies $\forall s.\ P\ s \longrightarrow wp\ c\ Q\ s$. Therefore we can strengthen the precondition of $\vdash \{wp\ c\ Q\}\ c\ \{Q\}$ and infer $\vdash \{P\}\ c\ \{Q\}$. □

Putting soundness and completeness together we obtain that a triple is provable in Hoare logic iff it is provable via the operational semantics:

Corollary 12.13. $\vdash \{P\}\ c\ \{Q\} \iff \models \{P\}\ c\ \{Q\}$

Thus one can also view Hoare logic as a reformulation of operational semantics aimed at proving rather than executing—or the other way around.

12.3.2 Incompleteness

Having just proved completeness we will now explain a related incompleteness result. This section requires some background in recursion theory and logic. It can be skipped on first reading.

Recall from Section 12.2.1 the triple `{True}` c `{False}`. We argued that this triple is valid iff c terminates for no start state. It is well-known from recursion theory that the set of such never terminating programs is not recursively enumerable (r.e.) (see, for example, [43]). Therefore the set of valid Hoare triples is not r.e. either: an enumeration of all valid Hoare triples could easily be filtered to yield an enumeration of all valid `{True}` c `{False}` and thus an enumeration of all never terminating programs.

Therefore there is no sound and complete Hoare logic whose provable triples are r.e.. This is strange because we have just proved that our Hoare logic is sound and complete, and its inference rules together with the inference rules for HOL (see, for example, [38]) provide an enumeration mechanism for all provable Hoare triples. What is wrong here? Nothing. Both our soundness and completeness results and the impossibility of having a sound and complete Hoare logic are correct but they refer to subtly different notions of validity of Hoare triples. The notion of validity used in our soundness and completeness proofs is defined in HOL and we have shown that we can prove a triple valid iff we can prove it in the Hoare logic. Thus we have related provability in HOL of two different predicates. On the other hand, the impossibility result is based on an abstract mathematical notion of validity and termination independent of any proof system. This abstract notion of validity is stronger than our HOL-based definition. That is, there are Hoare triples that are valid in this abstract sense but whose validity cannot be proved in HOL. Otherwise the equivalence of $\models$ and $\vdash$ we proved in HOL would contradict the impossibility of having a sound and complete Hoare logic (assuming that HOL is consistent).

What we have just seen is an instance of the incompleteness of HOL, not Hoare logic: there are sentences in HOL that are true but not provable. Gödel [35] showed that this is necessarily the case in any sufficiently strong, consistent and r.e. logic. Cook [21] was the first to analyse this incompleteness of Hoare logic and to show that, because it is due to the incompleteness of the assertion language, one can still prove what he called **relative completeness**

of Hoare logic. The details are beyond the scope of this book. Winskel [94] provides a readable account.

12.3.3 Exercises

Exercise 12.14. Prove Fact 12.10.

Exercise 12.15. Design and prove a forward assignment rule of the form $\vdash \{P\}\ x ::= a\ \{?\}$ where $?$ is a suitable postcondition. Hint: use completeness.

Exercise 12.16. Add a *REPEAT c UNTIL b* command to IMP (see Exercise 7.23, big-step semantics only). Extend Hoare logic and the soundness and completeness proofs accordingly.

Exercise 12.17. Replace the assignment command with a new command *Do f* where $f :: state \Rightarrow state$ can be an arbitrary state transformer. Update the big-step semantics, Hoare logic and the soundness and completeness proofs.

Exercise 12.18. Based on Exercise 7.24 extend Hoare logic and the soundness and completeness proofs with nondeterministic choice.

Exercise 12.19. Instead of the weakest precondition one can also work with the **strongest postcondition**:

$$sp\ P\ c = (\lambda t.\ \exists s.\ P\ s \wedge (c, s) \Rightarrow t)$$

1. Show that this indeed defines the strongest postcondition: prove $\models \{P\}\ c\ \{sp\ P\ c\}$ and $\models \{P\}\ c\ \{Q\} \Longrightarrow \forall s.\ sp\ P\ c\ s \longrightarrow Q\ s$.
2. Prove recursion equations for *sp* analogous to those for *wp*. Hint: the one for assignment requires an existential quantifier.
3. Redo the completeness proof for Hoare logic with *sp* instead of *wp*.

12.4 Verification Condition Generation thy

This section shows what we have already hinted at: Hoare logic can be automated. That is, we reduce provability in Hoare logic to provability in the assertion language, i.e. HOL in our case. Given a triple $\{P\}\ c\ \{Q\}$ that we want to prove, we show how to compute an assertion A from it such that $\vdash \{P\}\ c\ \{Q\}$ is provable iff A is provable.

We call A a **verification condition** and the function that computes A a **verification condition generator** or **VCG**. The advantage of working

with a VCG is that no knowledge of Hoare logic is required by the person or machine that attempts to prove the generated verification conditions. Most systems for the verification of imperative programs are based on VCGs.

Our VCG works like The Method for Hoare logic we sketched above: it simulates the backward application of Hoare logic rules and gathers up the implications between assertions that arise in the process. Of course there is the problem of loop invariants: where do they come from? We take the easy way out and let the user provide them. In general this is the only feasible solution because we cannot expect the machine to come up with clever invariants in all situations. In Chapter 13 we will present a method for computing invariants in simple situations.

Invariants are supplied to the VCG as annotations of *WHILE* loops. For that purpose we introduce a type *acom* of **annotated commands** with the same syntax as that of type *com*, except that *WHILE* is annotated with an assertion *Inv*:

$\{Inv\}\ WHILE\ b\ DO\ C$

To distinguish variables of type *com* and *acom*, the latter are capitalised. Function $strip :: acom \Rightarrow com$ removes all annotations from an annotated command, thus turning it into an ordinary command.

Verification condition generation is based on two functions: *pre* is similar to *wp*, *vc* is the actual VCG.

fun $pre :: acom \Rightarrow assn \Rightarrow assn$ **where**
$pre\ SKIP\ Q = Q$
$pre\ (x ::= a)\ Q = (\lambda s.\ Q\ (s[a/x]))$
$pre\ (C_1;;\ C_2)\ Q = pre\ C_1\ (pre\ C_2\ Q)$
$pre\ (IF\ b\ THEN\ C_1\ ELSE\ C_2)\ Q$
$= (\lambda s.\ if\ bval\ b\ s\ then\ pre\ C_1\ Q\ s\ else\ pre\ C_2\ Q\ s)$
$pre\ (\{I\}\ WHILE\ b\ DO\ C)\ Q = I$

Function *pre* follows the recursion equations for *wp* except in the *WHILE* case where the annotation is returned. If the annotation is an invariant then it must also hold before the loop and thus it makes sense for *pre* to return it.

fun $vc :: acom \Rightarrow assn \Rightarrow assn$ **where**
$vc\ SKIP\ Q = (\lambda s.\ True)$
$vc\ (x ::= a)\ Q = (\lambda s.\ True)$
$vc\ (C_1;;\ C_2)\ Q = (\lambda s.\ vc\ C_1\ (pre\ C_2\ Q)\ s \wedge vc\ C_2\ Q\ s)$
$vc\ (IF\ b\ THEN\ C_1\ ELSE\ C_2)\ Q = (\lambda s.\ vc\ C_1\ Q\ s \wedge vc\ C_2\ Q\ s)$
$vc\ (\{I\}\ WHILE\ b\ DO\ C)\ Q =$
$(\lambda s.\ (I\ s \wedge bval\ b\ s \longrightarrow pre\ C\ I\ s) \wedge$
$\quad (I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s) \wedge vc\ C\ I\ s)$

Function *vc* essentially just goes through the command and produces the following two verification conditions for each $\{I\}$ *WHILE b DO C*:

- $I\ s \wedge bval\ b\ s \longrightarrow pre\ C\ I\ s$
  It expresses that I and b together imply the precondition that I holds again after C, i.e. I is an invariant.
- $I\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$
  It expresses that at the end of the loop the postcondition holds.

The recursive invocation *vc C I s* merely generates the verification conditions for any loops inside the body *C*.

For the other constructs only trivial verification conditions (*True*) are generated or the results of subcomputations are combined with $\wedge$.

In examples we revert to syntactic assertions for compactness and readability. We only need to drop the state parameter *s* and perform substitution on assertions instead of states.

Example 12.20. We return to our familiar summation program and define the following abbreviations:

```
W  =  {I} WHILE 0 < x DO C     C  =  y := y+x;; x := x-1
I  =  (y + sum x = sum i)      Q  =  (y = sum i)
```

The following equations show the computation of `vc W Q`:

```
vc W Q = ((I ∧ 0 < x ⟶ pre C I) ∧ (I ∧ ¬ 0 < x ⟶ Q)
          ∧ vc C I)
pre C I = pre (y := y+x) (pre (x := x-1) I)
        = pre (y := y+x) (I[x-1/x])
        = I[x-1/x][y+x/y] = (y+x + sum (x-1) = sum i)
vc C I = (vc (y := y+x) (pre (x := x-1) I) ∧ vc (x := x-1) I)
       = (True ∧ True)
```

Therefore `vc W Q` boils down to

```
(I ∧ 0 < x ⟶ I[x-1/x][y+x/y]) ∧ (I ∧ x ⩽ 0 ⟶ Q)
```

The exact same conditions arose in the first Hoare logic proof of this example and we satisfied ourselves that they are true.

The example has demonstrated the computation of *vc* and *pre*. The generated verification condition turned out to be true, but it remains unclear what exactly that tells us. We need to show that our VCG is sound w.r.t. Hoare logic. This should allow us to reduce the question if $\vdash \{P\}\ c\ \{Q\}$ to the question if the verification condition is true. More precisely we will obtain the following result:

Corollary 12.21.
$[\![\forall s.\ vc\ C\ Q\ s;\ \forall s.\ P\ s \longrightarrow pre\ C\ Q\ s]\!] \Longrightarrow \vdash \{P\}\ strip\ C\ \{Q\}$

This can be read as a procedure for proving $\vdash \{P\}\ c\ \{Q\}$:

1. Annotate c with invariants, yielding C such that $strip\ C = c$.
2. Prove the verification condition $vc\ C\ Q$ and that P implies $pre\ C\ Q$.

The actual soundness lemma is a bit more compact than its above corollary which follows from it by precondition strengthening.

Lemma 12.22 (Soundness of *pre* and *vc* w.r.t. $\vdash$).
$\forall s.\ vc\ C\ Q\ s \implies \vdash \{pre\ C\ Q\}\ strip\ C\ \{Q\}$

Proof. By induction on c. The *WHILE* case is routine, the other cases are automatic. □

How about completeness? Can we just reverse this implication? Certainly not: if C is badly annotated, $\vdash \{pre\ C\ Q\}\ strip\ C\ \{Q\}$ may be provable but not $\forall s.\ vc\ C\ Q\ s$.

Example 12.23. The triple `{x=1} WHILE True DO x := 0 {False}` is provable with the help of the invariant `True` and precondition strengthening:

$$\dfrac{\dfrac{\dfrac{}{\texttt{True}\wedge\texttt{True} \longrightarrow \texttt{True}}}{\{\texttt{True}\wedge\texttt{True}\}\ \texttt{x:=0}\ \{\texttt{True}\}} \quad \dfrac{}{\texttt{True}\wedge\neg\texttt{True} \longrightarrow \texttt{False}}}{\{\texttt{True}\}\ \texttt{WHILE True DO x:=0}\ \{\texttt{False}\}} \quad \dfrac{}{\texttt{x=1}\longrightarrow\texttt{True}}}{\{\texttt{x=1}\}\ \texttt{WHILE True DO x := 0}\ \{\texttt{False}\}}$$

But starting from the badly annotated `W = {x=1} WHILE True DO x := 0` one of the verification conditions will be that `x=1` is an invariant, which it is not. Hence `vc W False` is not true.

However there always is an annotation that works:

Lemma 12.24 (Completeness of *pre* and *vc* w.r.t. $\vdash$).
$\vdash \{P\}\ c\ \{Q\} \implies$
$\exists C.\ strip\ C = c \wedge (\forall s.\ vc\ C\ Q\ s) \wedge (\forall s.\ P\ s \longrightarrow pre\ C\ Q\ s)$

Proof. The proof requires two little monotonicity lemmas:

$$[\![\forall s.\ P\ s \longrightarrow P'\ s;\ pre\ C\ P\ s]\!] \implies pre\ C\ P'\ s$$
$$[\![\forall s.\ P\ s \longrightarrow P'\ s;\ vc\ C\ P\ s]\!] \implies vc\ C\ P'\ s$$

Both are proved by induction on c; each case is automatic.

In the rest of the proof the formula $\forall s.\ P\ s$ is abbreviated to P and $\forall s.\ P\ s \longrightarrow Q\ s$ is abbreviated to $P \to Q$.

The proof of the completeness lemma is by rule induction on $\vdash \{P\}\ c\ \{Q\}$. We only consider the sequential composition rule in detail:

$$\frac{\vdash \{P_1\}\ c_1\ \{P_2\} \qquad \vdash \{P_2\}\ c_2\ \{P_3\}}{\vdash \{P_1\}\ c_1;;\ c_2\ \{P_3\}}$$

From the IHs we obtain C_1 and C_2 such that *strip* $C_1 = c_1$, *vc* C_1 P_2, $P_1 \to$ *pre* C_1 P_2, *strip* $C_2 = c_2$, *vc* C_2 P_3, $P_2 \to$ *pre* C_2 P_3. We claim that $C' = C_1;;\ C_2$ is the required annotated command. Clearly *strip* $C' = c_1;;\ c_2$. From *vc* C_1 P_2 and $P_2 \to$ *pre* C_2 P_3 it follows by monotonicity that *vc* C_1 (*pre* C_2 P_3); together with *vc* C_2 P_3 this implies *vc* C' P_3. From $P_2 \to$ *pre* C_2 P_3 it follows by monotonicity of *pre* that *pre* C_1 $P_2 \to$ *pre* C_1 (*pre* C_2 P_3); because $P_1 \to$ *pre* C_1 P_2 we obtain the required $P_1 \to$ *pre* C' P_3, thus concluding the case of the sequential composition rule.

The *WHILE* rule is special because we need to synthesise the loop annotation. This is easy: just take the invariant P from the *WHILE* rule.

The remaining rules are straightforward. □

12.4.1 Exercises

Exercise 12.25. Let *asum* i be the annotated command `W` in Example 12.20. Prove $\vdash \{\lambda s.\ s\ ''x'' = i\}$ *strip* (*asum* i) $\{\lambda s.\ s\ ''y'' = sum\ i\}$ with the help of theorem *vc_sound'*, i.e. Corollary 12.21.

Exercise 12.26. Write an annotated command that multiplies two variables and verify it like the summantion command in the previous exercise. You may assume that one of the two input variables is non-negative. The list of theorems *algebra_simps* can be *add*ed to the simplifier to improve automation.

Exercise 12.27. Having two separate functions *pre* and *vc* is inefficient. When computing *vc* one often needs to compute *pre* too, leading to multiple traversals of many subcommands. Define an optimised function *prevc* :: *acom* $\Rightarrow$ *assn* $\times$ *assn* that traverses the command only once and prove *prevc* c Q = (*pre* c Q, *vc* c Q).

12.5 Hore Logic for Total Correctness thy

Recall the informal definition of total correctness of a triple $\{P\}\ c\ \{Q\}$:

> If P is true before the execution of c
> then c terminates and Q is true afterwards.

Formally, **validity for total correctness** is defined like this:

$$\models_t \{P\}\ c\ \{Q\} \ \longleftrightarrow\ (\forall s.\ P\ s \longrightarrow (\exists t.\ (c, s) \Rightarrow t \wedge Q\ t))$$

In this section we always refer to this definition when we speak of validity.

Note that this definition assumes that the language is deterministic. Otherwise $\models_t \{P\}\ c\ \{Q\}$ may hold although only some computations starting from P terminate in Q while others may show any behaviour whatsoever.

Hoare logic for total corrections is defined like for partial correctness, except that we write $\vdash_t$ and that the *WHILE* rule is augmented with a relation $T :: state \Rightarrow nat \Rightarrow bool$ that guarantees termination:

$$\frac{\bigwedge n.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge T\ s\ n\}\ c\ \{\lambda s.\ P\ s \wedge (\exists n'{<}n.\ T\ s\ n')\}}{\vdash_t \{\lambda s.\ P\ s \wedge (\exists n.\ T\ s\ n)\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}}$$

The purpose of the universally quantified $n :: nat$ in the premise is to remember the value of T in the precondition to express that it has decreased in the postcondition. The name of the rule is again *While*.

Although this formulation with a relation T has a technical advantage, the following derived rule formulated with a measure function $f :: state \Rightarrow nat$ is more intuitive. We call this rule *While_fun*:

$$\frac{\bigwedge n.\ \vdash_t \{\lambda s.\ P\ s \wedge bval\ b\ s \wedge n = f\ s\}\ c\ \{\lambda s.\ P\ s \wedge f\ s < n\}}{\vdash_t \{P\}\ WHILE\ b\ DO\ c\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}}$$

This is like the partial correctness rule except that it also requires a measure function that decreases with each iteration. In case you wonder how to derive the functional version: set $T = (\lambda s\ n.\ f\ s = n)$ in rule *While*.

Example 12.28. We redo the proof of *wsum* from Section 12.2.2. The only difference is that when applying rule *While_fun* (combined with postcondition strengthening as in rule *While'*) we need not only instantiate P as previously but also f: $f = (\lambda s.\ nat\ (s\ ''x''))$ where the predefined function *nat* coerces an *int* into a *nat*, coercing all negative numbers to 0. The resulting invariance subgoal now looks like this:

$$\begin{array}{l}\bigwedge n.\ \vdash_t \{\lambda s.\ s\ ''y'' = sum\ i - sum\ (s\ ''x'') \wedge \\ \qquad\qquad bval\ (Less\ (N\ 0)\ (V\ ''x''))\ s \wedge n = nat\ (s\ ''x'')\} \\ \quad ''y'' ::= Plus\ (V\ ''y'')\ (V\ ''x'');;\ ''x'' ::= Plus\ (V\ ''x'')\ (N\ {-}1) \\ \quad \{\lambda s.\ s\ ''y'' = sum\ i - sum\ (s\ ''x'') \wedge nat\ (s\ ''x'') < n\}\end{array}$$

The rest of the proof steps are again identical to the partial correctness proof. After pulling back the postcondition the additional conjunct in the goal is $nat\ (s\ ''x'' - 1) < n$ which follows automatically from the assumptions $0 < s\ ''x''$ and $n = nat\ (s\ ''x'')$.

Lemma 12.29 (Soundness of $\vdash_t$ w.r.t. $\models_t$).
$\vdash_t \{P\}\ c\ \{Q\} \implies \models_t \{P\}\ c\ \{Q\}$

Proof. By rule induction. All cases are automatic except rule *While* which we look at in detail. Let $w = \mathit{WHILE}\ b\ \mathit{DO}\ c$. By a (nested) induction on n we show for arbitrary n and s

$$[\![P\ s;\ T\ s\ n]\!] \Longrightarrow \exists t.\ (w,\ s) \Rightarrow t \wedge P\ t \wedge \neg\ \mathit{bval}\ b\ t \qquad (*)$$

from which $\models_t \{\lambda s.\ P\ s \wedge (\exists n.\ T\ s\ n)\}\ w\ \{\lambda s.\ P\ s \wedge \neg\ \mathit{bval}\ b\ s\}$, the goal in the *While* case, follows immediately. The inductive proof assumes that $(*)$ holds for all smaller n. For n itself we argue by cases. If $\neg\ \mathit{bval}\ b\ s$ then $(w,\ s) \Rightarrow t$ is equivalent with $t = s$ and $(*)$ follows. If $\mathit{bval}\ b\ s$ then we assume $P\ s$ and $T\ s\ n$. The outer IH $\models_t \{\lambda s.\ P\ s \wedge \mathit{bval}\ b\ s \wedge T\ s\ n\}\ c\ \{\lambda s'.\ P\ s' \wedge (\exists n'.\ T\ s'\ n' \wedge n' < n)\}$ yields s' and n' such that $(c,\ s) \Rightarrow s'$, $P\ s'$, $T\ s'\ n'$ and $n' < n$. Because $n' < n$ the inner IH yields the required t such that $(w,\ s') \Rightarrow t$, $P\ t$ and $\neg\ \mathit{bval}\ b\ t$. With rule *WhileTrue* the conclusion of $(*)$ follows. □

The completeness proof proceeds like the one for partial correctness. The weakest precondition is now defined in correspondence to $\models_t$:

$$\mathit{wp}_t\ c\ Q = (\lambda s.\ \exists t.\ (c,\ s) \Rightarrow t \wedge Q\ t)$$

The same recursion equations as for wp can also be proved for wp_t. The crucial lemma is again this one:

Lemma 12.30. $\vdash_t \{\mathit{wp}_t\ c\ Q\}\ c\ \{Q\}$

Proof. By induction on c. We focus on the *WHILE* case because the others are automatic, thanks to the wp_t equations. Let $w = \mathit{WHILE}\ b\ \mathit{DO}\ c$. The termination relation T counts the number of iterations of w and is defined inductively by the two rules

$$\frac{\neg\ \mathit{bval}\ b\ s}{T\ s\ 0} \qquad \frac{\mathit{bval}\ b\ s \qquad (c,\ s) \Rightarrow s' \qquad T\ s'\ n}{T\ s\ (n + 1)}$$

Because IMP is deterministic, T is a functional relation:

$$[\![T\ s\ n;\ T\ s\ n']\!] \Longrightarrow n = n'$$

This is easily proved by induction on the first premise.

Moreover, T is 'defined' for any state from which w terminates:

$$(w,\ s) \Rightarrow t \ \Longrightarrow\ \exists n.\ T\ s\ n \qquad (*)$$

The proof is an easy rule induction on the premise.

Now we come to the actual proof. The IH is $\vdash_t \{\mathit{wp}_t\ c\ R\}\ c\ \{R\}$ for any R, and we need to prove $\vdash_t \{\mathit{wp}_t\ w\ Q\}\ w\ \{Q\}$. In order to apply the *WHILE* rule we use the consequence rule to turn the precondition into $P = (\lambda s.\ \mathit{wp}_t\ w\ Q\ s \wedge (\exists n.\ T\ s\ n))$ and the postcondition into $\lambda s.\ P\ s \wedge \neg\ \mathit{bval}\ b\ s$. Thus we have to prove the following three goals:

$\forall s.\ wp_t\ w\ Q\ s \longrightarrow P\ s$
$\vdash_t \{P\}\ w\ \{\lambda s.\ P\ s \wedge \neg\ bval\ b\ s\}$
$\forall s.\ P\ s \wedge \neg\ bval\ b\ s \longrightarrow Q\ s$

The third goal follows because $\neg\ bval\ b\ s$ implies $wp_t\ w\ Q\ s = Q\ s$ and therefore $P\ s$ implies $Q\ s$. The first goal follows directly from $(*)$ by definition of wp_t. Applying the *WHILE* rule backwards to the second goal leaves us with $\vdash_t \{P'\}\ c\ \{R\}$ where $P' = (\lambda s.\ P\ s \wedge bval\ b\ s \wedge T\ s\ n)$ and $R = (\lambda s'.\ P\ s' \wedge (\exists n'{<}n.\ T\ s'\ n'))$. By IH we have $\vdash_t \{wp_t\ c\ R\}\ c\ \{R\}$ and we can obtain $\vdash_t \{P\}\ c\ \{R\}$ by precondition strengthening because $\forall s.\ P'\ s \longrightarrow wp_t\ c\ R\ s$. To prove the latter we assume $P'\ s$ and show $wp_t\ c\ R\ s$. From $P\ s$ we obtain by definition of wp_t some t such that $(w,\ s) \Rightarrow t$ and $Q\ t$. Because $bval\ b\ s$, rule inversion yields a state s' such that $(c,\ s) \Rightarrow s'$ and $(w,\ s') \Rightarrow t$. From $(w,\ s') \Rightarrow t$ we obtain a number n' such that $T\ s'\ n'$. By definition of T, $T\ s\ (n' + 1)$ follows. Because T is functional we have $n = n' + 1$. Together with $(c,\ s) \Rightarrow s'$, $(w,\ s') \Rightarrow t$, $Q\ t$ and $T\ s'n'$ this implies $wp_t\ c\ R\ s$ by definition of wp_t. □

The completeness theorem is an easy consequence:

Theorem 12.31 (Completeness of $\vdash_t$ w.r.t. $\models_t$).
$\models_t \{P\}\ c\ \{Q\} \implies \vdash_t \{P\}\ c\ \{Q\}$

The proof is the same as for Theorem 12.12.

12.5.1 Exercises

Exercise 12.32. Prove total correctness of the program in Exercise 12.3.

Exercise 12.33. Solve Exercise 12.16 for total instead of partial correctness.

Exercise 12.34. Extend the VCG in Section 12.4 to cover total correctness.

12.6 Summary and Further Reading

This chapter was dedicated to Hoare logic and the verification of IMP programs. We covered three main topics:

- A Hoare logic for partial correctness and its soundness and completeness w.r.t. the big-step semantics.
- A verification condition generator that reduces the task of verifying a program by means of Hoare logic to the task of annotating all loops in that program with invariants and proving that these annotations are indeed invariants and imply the necessary postconditions.

- A Hoare logic for total correctness and its soundness and completeness.

Hoare logic is a huge subject area and we have only scratched the surface. Therefore we provide further references for the theory and applications of Hoare logic.

12.6.1 Theory

A precursor of Hoare logic is Floyd's method of annotated flowcharts [31]. Hoare [42] transferred Floyd's idea to inference rules for structured programs and Hoare logic (as we now know it) was born. Hence it is sometimes called Floyd-Hoare logic.

Soundness and completeness was first proved by Cook [21]. An early overview of the foundations of various Hoare logics is due to Apt [6, 7]. An excellent modern introduction to the many variants of Hoare logic is the book by Apt, de Boer and Olderog [8] which covers procedures, objects and concurrency.

Weakest preconditions are due to Dijkstra [29]. He reformulated Hoare logic as a weakest precondition calculus to facilitate the verification of concrete programs.

All of the above references follow the syntactic approach to assertions. The book by Nielson and Nielson [64] is an exception in that its chapters on program verification follow the functional approach.

Formalisations of Hoare logic in theorem provers have all followed the functional approach. The first such formalisation is due to Gordon [37] who showed the way and proved soundness. Nipkow [66] also proved completeness. Schreiber [81] covered procedures, which was extended with nondeterminism by Nipkow [68, 67]. Nipkow and Prensa [70] formalised a Hoare logic for concurrency.

Formalising Hoare logics is not a frivolous pastime. Apt observes "various proofs given in the literature are awkward, incomplete or even incorrect" [6]. In fact, Apt himself presents a proof system for total correctness of procedures that was later found to be unsound by America and de Boer [5] who presented a correct version. The formalised system by Schreiber [81] is arguably slicker.

Finally we mention work on verifying programs that manipulate data structures on the heap. An early approach by Burstall [17] was formalised by Bornat [14] and Mehta and Nipkow [57]. A recent and very influential approach is **separation logic** [75]. Its formulas provide special connectives for talking about the heap. They serve as assertions for heap-manipulating programs.

12.6.2 Applications

Our emphasis on foundational issues and toy examples is bound to give the reader the impression that Hoare logic is not practically useful. Luckily, that is not the case.

Among the early practical program verification tools are the KeY [11] and KIV [74] systems. The KeY system can be used to reason about Java programs. One of the recent larger applications of the KIV system is for instance the verification of file-system code in operating systems [30].

Hoare-logic based tools also exist for concurrent programs. The VCC tool [20], for instance, is integrated with Microsoft Visual studio and can be used to verify concurrent C. Cohen et al have used VCC to demonstrate the verification of a small operating system hypervisor [4]. Instead of interactive proof, VCC aims for automation and uses the SMT solver Z3 [26] as back-end. The interaction with the tool consists of annotating invariants and function pre/post conditions in such a way that they are simple enough for the prover to succeed automatically. The automation is stronger than in interactive proof assistants like Isabelle, but it forces specifications to be phrased in first-order logic. The Dafny tool [54] uses the same infrastructure. It does not support concurrency, but is well suited for learning this paradigm of program verification.

Of course, Isabelle does also support practical program verification. As mentioned in Chapter 7, the Simpl [79] verification framework for Isabelle develops Hoare-logic from its foundations to a degree that can directly be used for the verification of C programs [89]. While it provides less automation than tools like VCC, the user is rewarded with the full power and flexibility of higher-order logic. Two of the largest formal software verifications to date used Isabelle, both in the area of operating system verification. The Verisoft project [3] looked at constructing a verified software stack from verified hardware up to verified applications, an aspiration of the field that goes back to the 1980s [13]. The seL4 project verified a commercially viable operating system microkernel consisting of roughly 10,000 lines of code [48] in Isabelle. The project made full use of the flexibility of higher-order logic and later extend the verification to include higher-level security properties such as integrity and information-flow noninterference [62], as well as verification down to the compiled binary of the kernel [85].

13

Abstract Interpretation

In Chapter 10 we had seen a number of automatic program analyses, and each one was hand-crafted. Abstract Interpretation is a generic approach to automatic program analysis. In principle, it covers all of our earlier analyses. In this chapter we ignore the optimising program transformations that accompany certain analyses.

The specific form of abstract interpretation we consider aims to compute the possible values of all variables at each program point. In order to infer this information the program is interpreted with abstract values that represent sets of concrete values, for example, with intervals instead of integers.

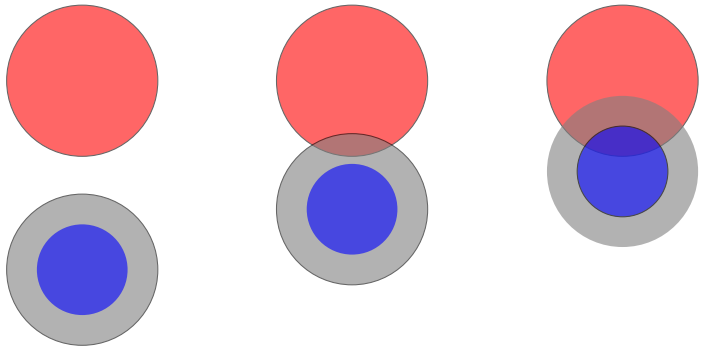
Program analyses are necessarily imprecise because they are automatic. We already discussed this in the beginning of Chapter 10. For analyses that compute if some program point is reachable, which the analyses in this chapter do, this is particularly obvious: the end of the program is reachable iff the program terminates for some input, but termination is undecidable.

Therefore a program analysis should *overapproximate* to be on the safe side: it should compute a superset of the possible values that the variables can take on at each program point.

- If the analysis says that some value cannot arise, this is definitely the case.
- But if the analysis says that some value can arise, this is only potentially the case.

That is, if the analysis says that the variables have no possible value, which means that the program point is unreachable, then it really is unreachable, and the code at that point can safely be removed. But the analysis may claim some points are reachable which in fact are not, thus missing opportunities for code removal. Similarly for constant folding: if the analysis says that `x` always has value 2 at some point, we can safely replace `x` by 2 at that point. On the other hand, if the analysis says that `x` could also have 3 at that point, constant folding is out, although `x` may in fact only have one possible value.

If we switch from an optimization to a debugging or verification point of view, we think of certain states as erroneous (e.g., where `x = 0`) because they lead to a runtime error (e.g., division by `x`) or violate some given specification (e.g., `x > 0`). The analysis is meant to find if there is any program point where the reachable states include erroneous states. The following figure shows three possibilities how the analysis can behave. The red circle is the set erroneous states, the blue circle the set of reachable states, and the gray halo is the superset of the reachable states computed by the analysis, all for one fixed program point.



No Alarm False Alarm True Alarm

The term "alarm" describes the situation where the analysis finds an erroneous state, in which case it raises an alarm, typically by flagging the program point in question. In the No Alarm situation, the analysis does not find any erroneous states and everybody is happy. In the True Alarm situation, the analysis finds erroneous states and some reachable states are indeed erroneous. But due to overapproximation, there are also so-called **false alarms**: the analysis finds an erroneous state that cannot arise at this program point. False alarms are the bane of all program analyses. They force the programmer to convince himself that the potential error found is not a real error but just a weakness of the analysis.

13.1 Informal Introduction

At the center of our approach to abstract interpretation are **annotated commands**. These are commands interspersed with annotations containing semantic information. This is reminiscent of Hoare logic and we also borrow the $\{\ldots\}$ syntax for annotations. However, annotations may now be placed at all intermediate program points, not just in front of loops as invariants. Loops will be annotated like this:

$\{I\}$
$WHILE\ b\ DO\ \{P\}\ c$
$\{Q\}$

where I (as in "invariant") annotates the loop head, P annotates the point before the loop body c, and Q annotates the exit of the whole loop. The annotation points are best visualized by means of the control-flow graph in Figure 13.1. We introduced control-flow graphs in Section 10.3. In fact, Figure 13.1 is the same as Figure 10.6, except that we now label nodes not with sets of live variables but with the annotations at the corresponding program points. Edges labelled by compound commands stand for whole subgraphs.

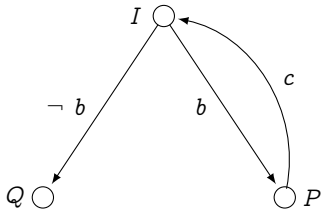


Fig. 13.1. Control-flow graph for $\{I\}\ WHILE\ b\ DO\ \{P\}\ c\ \{Q\}$

13.1.1 Collecting Semantics

Before we come to the actual program analysis we need a semantics to justify it against. This semantics needs to express for each program point the set of states that can arise at this point during an execution. It is known as a **collecting semantics** because it collects together all the states that can arise at each point. Neither the big nor the small-step semantics express this information directly. Here is an example of a program annotated with its collecting semantics:

```
x := 0  {{<x := 0>}} ;
{{<x := 0>, <x := 2>, <x := 4>}}
WHILE x < 3
DO  {{<x := 0>, <x := 2>}}
    x := x+2  {{<x := 2>, <x := 4>}}
{{<x := 4>}}
```

Annotations are of the form $\{\{\ldots\}\}$ because the object inside the outer annotation braces is a set. Computing the annotations is an iterative process that we explain later.

Annotations can also be infinite sets of states:

$\{\{\ldots, <x := -1>, <x := 0>, <x := 1>, \ldots\}\}$
`WHILE x < 3`
`DO` $\{\{\ldots, <x := 1>, <x := 2>\}\}$
`x := x+2` $\{\{\ldots, <x := 3>, <x := 4>\}\}$
$\{\{<x := 3>, <x := 4>, \ldots\}\}$

And programs can have multiple variables:

`x := 0; y := 0` $\{\{<x := 0, y := 0>\}\}$;
$\{\{<x := 0, y := 0>, <x := 2, y := 1>, <x := 4, y := 2>\}\}$
`WHILE x < 3`
`DO` $\{\{<x := 0, y := 0>, <x := 2, y := 1>\}\}$
`x := x+2; y := y+1` $\{\{<x := 2, y := 1>, <x := 4, y := 2>\}\}$
$\{\{<x := 4, y := 2>\}\}$

13.1.2 Abstract Interpretation

Now we move from the semantics to abstract interpretation in two steps. First we replace sets of states with single states that map variables to sets of values:

$$(vname \Rightarrow val)\ set \quad \text{becomes} \quad vname \Rightarrow val\ set.$$

Our first example above now looks much simpler:

`x := 0` $\{<x := \{0\}>\}$;
$\{x := \{0, 2, 4\}>\}$
`WHILE x < 3`
`DO` $\{<x := \{0, 2\}>\}$
`x := x+2` $\{<x := \{2, 4\}>\}$
$\{<x := \{4\}>\}$

However, this simplification comes at a price: it is an overapproximation that loses relationships between variables. For example, $\{<x := 0, y := 0>, <x := 1, y := 1>\}$ is overapproximated by $<x := \{0, 1\}, y := \{0, 1\}>$. The latter also subsumes $<x := 0, y := 1>$ and $<x := 1, y := 0>$.

In the second step we replace sets of values by "abstract values". This step is domain specific. For example, we can approximate sets of intergers by intervals. For the above example we obtain the following consistent annotation with integer intervals (written $[low, high]$):

`x := 0` $\{<x := [0, 0]>\}$;
$\{<x := [0, 4]>\}$
`WHILE x < 3`
`DO` $\{<x := [0, 2]>\}$

```
    x := x+2 {<x := [2, 4]>}
{<x := [3, 4]>}
```

Clearly, we have lost some precision in this step, but the annotations have become finitely representable: we have replaced arbitrary and potentially infinite sets by intervals, which are simply pairs of numbers.

How are the annotations actually computed? We start from an unannotated program and iterate abstract execution of the program (on intervals) until the annotations stabilize. Each execution step is a simultaneous execution of all edges of the control-flow graph. You can also think of it as a synchronous circuit where in each step simultaneously all units of the circuit process their input and make it their new output.

To demonstrate this iteration process we take the example above and give the annotations names:

```
x := 0 {A0} ;
{A1}
WHILE x < 3
DO {A2} x := x+2 {A3}
{A4}
```

In a separate table we can see how the annotations change with each step.

	0	1	2	3	4	5	6	7	8	9
A_0	$\bot$	[0, 0]								[0, 0]
A_1	$\bot$		[0, 0]			[0, 2]			[0, 4]	[0, 4]
A_2	$\bot$			[0, 0]			[0, 2]			[0, 2]
A_3	$\bot$				[2, 2]			[2, 4]		[2, 4]
A_4	$\bot$									[3, 4]

Instead of the full annotation $<x := ivl>$, the table merely shows *ivl*. Column 0 shows the initial annotations where $\bot$ represents the empty interval. Unchanged entries are left blank. In steps 1–3, [0, 0] is merely propagated around. In step 4, 2 is added to it, making it [2, 2]. The crucial step is from 4 to 5: [0, 0], the invariant A_1 in the making, is combined with the annotation [2, 2] at the end of the loop body, telling us that the value of x at A_1 can in fact be any value from the union [0, 0] and [2, 2]. This is overapproximated by the interval [0, 2]. The same thing happens again in step 8, where the invariant becomes [0, 4]. In step 9, the final annotation A_4 is reached at last: the invariant is now [0, 4] and intersecting it with the negation of `x < 3` lets [3, 4] reach the end of the loop. The annotations reached in step 9 (which are displayed in full) are stable: performing another step leaves them unchanged.

This is the end of our informal introduction and we become formal again. First we define the type of annotated commands, then the collecting semantics, and finally abstract interpretation. Most of the chapter is dedicated to the stepwise development of a generic abstract interpreter whose precision is gradually increased.

The rest of this chapter builds on Section 10.4.1.

13.2 Annotated Commands thy

The type of commands annotated with values of type *'a* is called *'a acom*. Just like *com*, *'a acom* is defined as a datatype together with concrete syntax. The concrete syntax is described by the following grammar:

```
'a acom ::= SKIP  { 'a }
          | string ::= aexp  { 'a }
          | 'a acom ;; 'a acom
          | IF bexp THEN { 'a } 'a acom ELSE { 'a } 'a acom
            { 'a }
          | { 'a }
            WHILE bexp DO  { 'a } 'a acom
            { 'a }
```

We exclusively use the concrete syntax and omit to show the actual datatype. Having both kinds of commands around at the same time introduces a minor ambiguity: we need to write *com.SKIP* to refer to the *SKIP* of type *com*.

Variables C, C_1, C' etc will henceforth stand for annotated commands.

The layout of *IF* and *WHILE* is suggestive of the intended meaning of the annotations but has no logical significance. We have already discussed the annotatations of *WHILE* but not yet of *IF*:

$IF\ b\ THEN\ \{P_1\}\ C_1\ ELSE\ \{P_2\}\ C_2$
$\{Q\}$

Annotation P_i refers to the state before the execution of C_i. Annotation $\{Q\}$ is placed on a separate line to emphasize that it refers to the state after the execution of the whole conditional, not just the *ELSE* branch. The corresponding annotated control-flow graph is shown in Figure 13.2.

Our annotated commands are polymorphic because, as we have already seen, we will want to annotate programs with different objects: with sets of states for the collecting semantics and with abstract states, for example, involving intervals, for abstract interpretation.

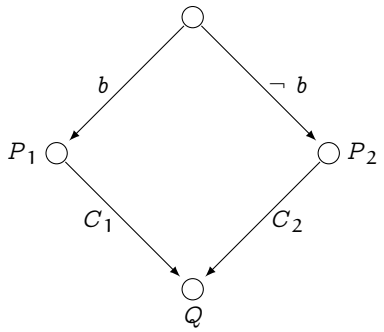


Fig. 13.2. Control-flow graph for *IF b THEN* $\{P_1\}$ C_1 *ELSE* $\{P_2\}$ C_2 $\{Q\}$

We now introduce a number of simple and repeatedly used auxiliary functions. Their functionality is straightforward and explained in words. In case of doubt, you can consult their full definitions in Appendix A.

strip :: *'a acom* ⇒ *com*
strips all annotations from an annotated command.

annos :: *'a acom* ⇒ *'a list*
extracts the list of all annotations from an annotated command, in left-to-right order.

anno :: *'a acom* ⇒ *nat* ⇒ *'a*
yields the nth annotation of an annotated command, starting at 0:

anno C p = *annos C* ! *p*

The standard infix operator ! indexes into a list with a natural number.

post :: *'a acom* ⇒ *'a*
returns the rightmost/last/post annotation of an annotated command:

post C = *last* (*annos C*)

annotate :: (*nat* ⇒ *'a*) ⇒ *com* ⇒ *'a acom*
annotates a command: annotation number *p* (as counted by *anno*) is set to *f p*. The characteristic lemma is

p < *asize c* ⟹ *anno* (*annotate f c*) *p* = *f p*

where *asize* counts the number of annotation positions in a command.

map_acom :: (*'a* ⇒ *'b*) ⇒ *'a acom* ⇒ *'b acom*
applies its first argument to every annotation of its second argument.

13.3 Collecting Semantics thy

The aim is to annotate commands with the set of states that can occur at each annotation point. The annotations are generated iteratively by a function *step* that maps annotated commands to annotated commands, updating the annotations. Each *step* executes all atomic commands (*SKIP*s and assignments) simultaneously and propagates the effects on the annotations forward. You can think of an annotated command as a synchronous circuit where with each clock tick (*step*) the information stored in each node (annotation point) is transformed by the outgoing edges and propagated to the successor nodes. Because we have placed (almost) no annotation points in front of commands, function *step* takes an additional argument, the set of states that are fed into the command. The full type of *step* and its recursive definition is shown in Figure 13.3. We will discuss the different cases one by one.

```
fun step :: state set ⇒ state set acom ⇒ state set acom where
step S (SKIP {Q})    = SKIP {S}
step S (x ::= e {Q})  = x ::= e {{s(x := aval e s) |s. s ∈ S}}
step S (C1;; C2)     = step S C1;; step (post C1) C2
step S (IF b THEN {P1} C1 ELSE {P2} C2 {Q})
  = IF b THEN {{s ∈ S. bval b s}} step P1 C1
    ELSE {{s ∈ S. ¬ bval b s}} step P2 C2
    {post C1 ∪ post C2}
step S ({I} WHILE b DO {P} C {Q})
  = {S ∪ post C}
    WHILE b
    DO {{s ∈ I. bval b s}}
       step P C
    {{s ∈ I. ¬ bval b s}}
```

Fig. 13.3. Definition of *step*

In the *SKIP* and the assigment case, the input set S is transformed and replaces the old post-annotation Q: for *SKIP* the transformation is the identity, $x ::= e$ transforms S by updating all of its elements (remember the set comprehension syntax explained in Section 4.2).

In the following let *S27* be the (somewhat arbitrary) state set $\{<x := 2>, <x := 7>\}$. Its is merely used to illustrate the behaviour of *step* on the various constructs. Here is an example for assignment:

step S27 $(x ::=$ *Plus* $(V\ x)\ (N\ 1)\ \{Q\}) =$

$x ::= Plus\ (V\ x)\ (N\ 1)\ \{\{<x := 3>, <x := 8>\}\}$

When applied to $C_1;;\ C_2$, *step* executes C_1 and C_2 simultaneously: the input to the execution of C_2 is the post-annotation of C_1, not of *step* $S\ C_1$. For example:

```
step S27
 (x ::= Plus (V x) (N 1) {{<x := 0>}};;
  x ::= Plus (V x) (N 2) {Q}) =
x ::= Plus (V x) (N 1) {{<x := 3>, <x := 8>}};;
x ::= Plus (V x) (N 2) {{<x := 2>}}
```

On *IF b THEN* $\{P_1\}\ C_1$ *ELSE* $\{P_2\}\ C_2\ \{\_\}$, *step* S does the following: it pushes filtered versions of S into P_1 and P_2 (this corresponds to the upper two edges in Figure 13.2), it executes C_1 and C_2 simultaneously (starting with the old annotations in front of them) and updates the post-annotation with *post* $C_1 \cup$ *post* C_2 (this corresponds to the lower two edges in Figure 13.2). Here is an example:

```
step S27
 (IF Less (V x) (N 5)
  THEN {{<x := 0>}} x ::= Plus (V x) (N 1) {{<x := 3>}}
  ELSE {{<x := 5>}} x ::= Plus (V x) (N 2) {{<x := 9>}}
  {Q}) =
IF Less (V x) (N 5)
THEN {{<x := 2>}} x ::= Plus (V x) (N 1) {{<x := 1>}}
ELSE {{<x := 7>}} x ::= Plus (V x) (N 1) {{<x := 7>}}
{{<x := 3>, <x := 9>}}
```

Finally we look at the *WHILE* case. It is similar to the conditional but feeds the post-annotation of the body of the loop back into the head of the loop. Here is an example:

```
step {<x := 7>}
 ({{<x := 2>, <x := 5>}}
  WHILE Less (V x) (N 5)
  DO {{<x := 1>}}
       x ::= Plus (V x) (N 2) {{<x := 4>}}
  {Q}) =
{{<x := 4>, <x := 7>}}
WHILE Less (V x) (N 5)
DO {{<x := 2>}}
     x ::= Plus (V x) (N 2) {{<x := 3>}}
{{<x := 7>}}
```

The collecting semantics is now defined by iterating *step S* where S is some fixed set of initial states, typically all possible states, i.e. $UNIV$. The iteration starts with a command that is annotated everywhere with the empty set of states because no state has reached an annotation point yet. We illustrate the process with the example program from Section 13.1:

```
x := 0 {A0} ;
{A1}
WHILE x < 3
DO {A2} x := x+2 {A3}
{A4}
```

In a separate table we can see how the annotations change with each iteration of *step S* (where S is irrelevant as long as it is not empty).

	0	1	2	3	4	5	6	7	8	9
A_0	{}	{0}								{0}
A_1	{}		{0}			{0,2}			{0,2,4}	{0,2,4}
A_2	{}			{0}			{0,2}			{0,2}
A_3	{}				{2}			{2,4}		{2,4}
A_4	{}									{4}

Instead of the full annotation $\{<x := i_1>,<x:=i_2>,\ldots\}$, the table merely shows $\{i_1,i_2,\ldots\}$. Unchanged entries are left blank. In steps 1–3, {0} is merely propagated around. In step 4, 2 is added to it, making it {2}. The crucial step is from 4 to 5: {0}, the invariant A_1 in the making, is combined with the annotation {2} at the end of the loop body, yielding {0,2}. The same thing happens again in step 8, where the invariant becomes {0,2,4}. In step 9, the final annotation A_4 is reached at last: intersecting the invariant {0,2,4} with the negation of x < 3 lets {4} reach the exit of the loop. The annotations reached in step 9 (which are displayed in full) are stable: performing another step leaves them unchanged.

In contrast to the interval analysis in Section 13.1, the semantics is and has to be exact. As a result, it is not computable in general. The above example is particularly simple in a number of respects that are all interrelated: the initial set S plays no role because x is initialized, all annotations are finite, and we reach a fixpoint after a finite number of steps. Most of the time we will not be so lucky.

13.3.1 Executing *step* in Isabelle

If we only deal with finite sets of states, we can let Isabelle execute *step* for us. Of course we again have to print states explicitly because they are functions. This is encapsulated in the function

$$show\_acom :: state\ set\ acom \Rightarrow (vname \times val)set\ set\ acom$$

that turns a state into a set of variable-value pairs. We reconsider the example program above, but now in full Isabelle syntax:

```
definition cex :: com where cex =
  ''x'' ::= N 0;;
  WHILE Less (V ''x'') (N 3)
  DO ''x'' ::= Plus (V ''x'') (N 2)

definition Cex :: state set acom where Cex = annotate (λp. {}) cex

value show_acom (step {<>} Cex)
```

yields

```
''x'' ::= N 0 {{{(''x'', 0)}}};;
{{}}
WHILE Less (V ''x'') (N 3)
DO {{}}
    ''x'' ::= Plus (V ''x'') (N 2) {{}}
{{}}
```

The triply-nested braces are the combination of the outer annotation braces with annotations that are sets of sets of variable-value pairs, the result of converting sets of states into printable form.

You can iterate *step* by means of the function iteration operator $f\ \hat{}\hat{}\ n$ (pretty-printed as f^n). For example, executing 4 steps

```
value show_acom ((step {<>} ^^ 4) Cex)
```

yields

```
''x'' ::= N 0 {{{(''x'', 0)}}};;
{{{(''x'', 0)}}}
WHILE Less (V ''x'') (N 3)
DO {{{(''x'', 0)}}}
    ''x'' ::= Plus (V ''x'') (N 2) {{{(''x'', 2)}}}
{{}}
```

Iterating *step* $\{<>\}$ 9 times reaches the fixpoint shown in the table above:

$''x''$ ::= $N\ 0$ {{{($''x''$, 0)}}};;
{{{($''x''$, 0)}, {($''x''$, 2)}, {($''x''$, 4)}}}
WHILE Less (V $''x''$) (N 3)
DO {{{($''x''$, 0)}, {($''x''$, 2)}}}
  $''x''$ ::= *Plus* (V $''x''$) (N 2) {{{($''x''$, 2)}, {($''x''$, 4)}}}
{{{($''x''$, 4)}}}

13.3.2 Collecting Semantics as Least Fixpoint

The collecting semantics is a fixpoint of *step* because a fixpoint describes an annotated command where the annotations are consistent with the execution via *step*. This is a fixpoint

x ::= $N\ 0$ {{<x := 0>}};;
{{<x := 0>, <x := 2>, <x := 4>}}
WHILE Less ($V\ x$) (N 3)
DO {{<x := 0>, <x := 2>}}
  x ::= *Plus* ($V\ x$) (N 2) {{<x := 2>, <x := 4>}}
{{<x := 4>}}

because (in control-flow graph terminology) each node is annotated with the transformed annotation of the predecessor node. For example, the assignment x ::= *Plus* ($V\ x$) (N 2) transforms {<x := 0>, <x := 2>} into {<x := 2>, <x := 4>}. In case there are multiple predecessors, the annotation is the union of the transformed annotations of all predecessor nodes: {<x := 0>, <x := 2>, <x := 4>} = {<x := 0>} ∪ {<x := 2>, <x := 4>} (there is no transformation).

We can also view a fixpoint as a solution of not just the single equation *step* $S\ C = C$ but of a system of equations, one for each annotation. If C = {I} *WHILE* b *DO* {P} C_0 {Q} then the equation system is

$I = S \cup post\ C_0$
$P = \{s \in I.\ bval\ b\ s\}$
$Q = \{s \in I.\ \neg\ bval\ b\ s\}$

together with the equations arising from C_0 = *step* $P\ C_0$. Iterating *step* is one way of solving this equation system. It corresponds to Jacobi iteration in linear algebra, where the new values for the unknowns are computed simultaneously from the old values. In principle, any method for solving an equation system can be used.

In general, *step* can have multiple fixpoints. For example

{I}
WHILE Bc True

DO {*I*}
    SKIP {*I*}
{{}}

is a fixpoint of *step* {} for every *I*. But only $I = \{\}$ makes computational sense: *step* {} means that the empty set of states is fed into the execution, and hence no state can ever reach any point in the program. This happens to be the least fixpoint (w.r.t. $\subseteq$). We will now show that *step* always has a least fixpoint. At the end of this section we prove that the least fixpoint is consistent with the big-step semantics.

13.3.3 Complete Lattices and Least Fixpoints thy

The Knaster-Tarski fixpoint theorem (Theorem 10.39) told us that monotone functions on sets have least pre-fixpoints, which are least fixpoints. Unfortunately *step* acts on annotated commands, not sets. Fortunately the theorem easily generalises from sets to complete lattices, and annotated commands form complete lattices.

Definition 13.1. *A type* $'a$ *with a partial order* $\leqslant$ *is a* ***complete lattice*** *if every set* $S :: 'a$ *set has a* ***greatest lower bound*** $\bigsqcap S :: 'a$*:*

- $\forall s \in S.\ \bigsqcap S \leqslant s$
- *If* $\forall s{\in}S.\ l' \leqslant s$ *then* $l' \leqslant \bigsqcap S$

The greatest lower bound of S *is often called the* ***infimum****.*

Note that in a complete lattice, the ordering determines the infimum uniquely: if l_1 and l_2 are infima of S then $l_1 \leqslant l_2$ and $l_2 \leqslant l_1$ and thus $l_1 = l_2$.

The archetypical complete lattice is the powerset lattice: for any type $'a$, its powerset, type $'a$ *set*, is a complete lattice where $\leqslant$ is $\subseteq$ and $\bigsqcap$ is $\bigcap$. This works because $\bigcap M$ is the greatest set below (w.r.t. $\subseteq$) all sets in M.

Lemma 13.2. *In a complete lattice, every set* S *of elements also has a* ***least upper bound (supremum)*** $\bigsqcup S$*:*

- $\forall s \in S.\ s \leqslant \bigsqcup S$
- *If* $\forall s{\in}S.\ s \leqslant u$ *then* $\bigsqcup S \leqslant u$

The least upper bound is the greatest lower bound of all upper bounds: $\bigsqcup S = \bigsqcap\{u.\ \forall s \in S.\ s \leqslant u\}$.

The proof is left as an exercise. Thus complete lattices can be defined via the existence of all infima or all suprema or both.

It follows that a complete lattice does not just have a least element $\bigsqcap UNIV$, the infimum of all elements, but also a greatest element $\bigsqcup UNIV$.

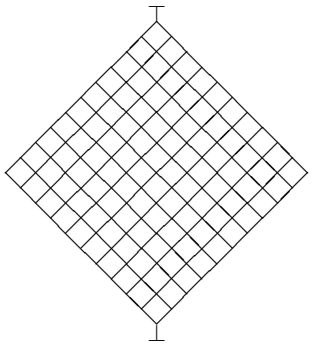


Fig. 13.4. Typical complete lattice

They are denoted by $\bot$ and $\top$ (pronounced "bottom" and "top"). Figure 13.4 shows a typical complete lattice and motivates the name "lattice".

The generalization of the Knaster-Tarski fixpoint theorem (Theorem 10.39) from sets to complete lattices is straightforward:

Theorem 13.3 (Knaster-Tarski). *Every monotone function f on a complete lattice has the least (pre-)fixpoint* $\bigsqcap\{p.\ f\ p \sqsubseteq p\}$.

The proof is the same as for the set version but with $\sqsubseteq$ and $\bigsqcap$ instead of $\subseteq$ and $\bigcap$. This works because $\bigcap M$ is the greatest lower bound of M w.r.t. $\subseteq$.

13.3.4 Annotated Commands as a Complete Lattice

In order to apply Knaster-Tarski to *step* we need to turn annotated commands into a complete lattice. Given an ordering $\sqsubseteq$ on $'a$, it can be lifted to $'a\ acom$ in a pointwise manner:

$$\begin{array}{l}(C_1 \sqsubseteq C_2) = \\ (strip\ C_1 = strip\ C_2 \wedge \\ \ (\forall p{<}length\ (annos\ C_1).\ anno\ C_1\ p \sqsubseteq anno\ C_2\ p))\end{array}$$

Two annotated commands C_1 and C_2 are only comparable if they are **structurally equal**, i.e. if $strip\ C_1 = strip\ C_2$. For example, $x ::= e\ \{\{a\}\} \sqsubseteq x ::= e\ \{\{a, b\}\}$ is *True* and both $x ::= e\ \{\{a\}\} \sqsubseteq x ::= e\ \{\{\}\}$ and $x ::= N\ 0\ \{S\} \sqsubseteq x ::= N\ 1\ \{S\}$ are *False*.

Lemma 13.4. *If $'a$ is a partial order, so is $'a\ acom$.*

Proof. The required properties carry over pointwise from $'a$ to $'a\ acom$. □

Because (in Isabelle) $\subseteq$ is just special syntax for $\sqsubseteq$ on sets, the above definition lifts $\subseteq$ to a partial order $\sqsubseteq$ on *state set acom*. Unfortunately this order

does not turn *state set acom* into a complete lattice: although $SKIP\ \{S\}$ and $SKIP\ \{T\}$ have the infimum $SKIP\ \{S \cap T\}$, the two commands $SKIP\ \{S\}$ and $x ::= e\ \{T\}$ have no lower bound, let alone an infimum. The fact is:

Only structurally equal annotated commands have an infimum.

Below we show that for each $c :: com$ the set $\{C :: 'a\ acom.\ strip\ C = c\}$ of all annotated commands structurally equal to c is a complete lattice, assuming that $'a$ is one. Thus we need to generalize complete lattices from types to sets.

Definition 13.5. *Let $'a$ be a partially ordered type. A set $L :: 'a\ set$ is a* ***complete lattice*** *if every $M \subseteq L$ has a greatest lower bound $\bigsqcap M \in L$.*

The type-based definition corresponds to the special case $L = UNIV$.

In the following we write $f \in A \to B$ as a shorthand for $\forall\, a{\in}A.\ f\ a \in B$. This notation is a set-based analogue of $f :: \tau_1 \Rightarrow \tau_2$.

Theorem 13.6 (Knaster-Tarski). *Let $L :: 'a\ set$ be a complete lattice and $f \in L \to L$ a monotone function. On L, f has the least (pre-)fixpoint*

$$lfp\ f = \bigsqcap\{p \in L.\ f\ p \leqslant p\}$$

The proof is the same as before, but we have to keep track of the fact that we always stay within L, which is ensured by $f \in L \to L$ and $M \subseteq L \Longrightarrow \bigsqcap M \in L$.

To apply Knaster-Tarski to *state set acom* we need to show that *acom* transforms complete lattices into complete lattices as follows:

Theorem 13.7. *Let $'a$ be a complete lattice and $c :: com$. Then the set $L = \{C :: 'a\ acom.\ strip\ C = c\}$ is a complete lattice.*

Proof. The infimum of a set of annotated commands is computed pointwise for each annotation:

$$Inf\_acom\ c\ M = annotate\ (\lambda p.\ \bigsqcap\ \{anno\ C\ p\ |C.\ C \in M)\ c$$

We have made explicit the dependence on the command c from the statement of the theorem. The infimum properties of *Inf_acom* follow easily from the corresponding properties of $\bigsqcap$. The proofs are automatic. □

13.3.5 Collecting Semantics

Because type *state set* is a complete lattice, Theorem 13.7 implies that $L = \{C :: state\ set\ acom.\ strip\ C = c\}$ is a complete lattice too, for any c. It is easy to prove (Lemma 13.35) that $step\ S \in L \to L$ is monotone: $C_1 \leqslant C_2 \Longrightarrow step\ S\ C_1 \leqslant step\ S\ C_2$. Therefore Kaster-Tarski tells us that *lfp* returns the least fixpoint and we can define the collecting semantics CS at last:

$$CS :: com \Rightarrow state\ set\ acom$$
$$CS\ c = lfp\ c\ (step\ UNIV)$$

Remarks:

- The extra argument c of *lfp* comes from the fact that *lfp* is defined in Theorem 13.6 in the context of a set L and our concrete L depends on c.
- The *UNIV* argument of *step* expresses that execution may start with any initial state. Other choices are possible too.

Function *CS* is not executable because the resulting annoations are usually infinite state sets. But just as for true liveness in Section 10.4.2, we can approximate (and sometimes reach) the *lfp* by iterating *step*. We already showed how that works in Section 13.3.1.

In contrast to previous operational semantics that were "obviously right", the collecting semantics is more complicated and less intuitive. In case you still have some nagging doubts that it is defined correctly, the following lemma should help. Remember that *post* extracts the final annotation (Appendix A).

Lemma 13.8. $(c, s) \Rightarrow t \Longrightarrow t \in post\ (CS\ c)$

Proof. By definition of *CS* the claim follows directly from $[\![(c, s) \Rightarrow t; s \in S]\!] \Longrightarrow t \in post(lfp\ c\ (step\ S))$ which in turn follows easily from two auxiliary propositions:

$$post(lfp\ c\ f) = \bigcap\{post\ C\ |C.\ strip\ C = c \wedge f\ C \leqslant C\}$$
$$[\![(c, s) \Rightarrow t; strip\ C = c; s \in S; step\ S\ C \leqslant C]\!] \Longrightarrow t \in post\ C$$

The first proposition is a direct consequence of $\forall C \in M.\ strip\ C = c \Longrightarrow post\ (Inf\_acom\ c\ M) = \bigcap (post\ `\ M)$, which follows easily from the definition of *post* and *Inf_acom*. The second propositions is the key. It is proved by rule induction. Most cases are automatic, but *WhileTrue* needs some work. □

Thus we know that the collecting semantics overapproximates the big-step semantics. Later we show that the abstract interpreter overapproximates the collecting semantics. Therefore we can view the collecting semantics merely as a stepping stone for proving that the abstract interpreter overapproximates the big-step semantics, our standard point of reference.

One can in fact show that both semantics are equivalent. One can also refine the lemma: it only talks about the *post* annotation but one would like to know that all annotations are correct w.r.t. the big-step sematics. We do not pursue this further but move on to the main topic of this chapter, abstract interpretation.

13.3.6 Exercises

The exercises below are conceptual and should be done on paper, also because manny of them require Isabelle material that will only be introduced later.

Exercise 13.9. Show the iterative computation of the collecting semantics of the following program in a table like the one in the beginning of Section 13.3.

```
x := 0; y := 2 {A0} ;
{A1}
WHILE 0 < y
DO {A2} ( x := x+y; y := y - 1 {A3} )
{A4}
```

Note that two annotations have been suppressed to make the task less tedious. You do not need to show steps where only the suppressed annotations change.

Exercise 13.10. Extend type *acom* and function *step* with a construct *OR* for the nondeterministic choice between two commands (see Exercise 7.24). Hint: think of *OR* as a nondeterministic conditional without a test.

Exercise 13.11. Prove that in a complete lattice $\bigsqcup S = \bigsqcap\{u.\ \forall s{\in}S.\ s \leqslant u\}$ is the least upper bound of S.

Exercise 13.12. Where is the mistake in the following argument? The natural numbers form a complete lattice because any set of natural numbers has an infimum, its least element.

Exercise 13.13. Show that the integers extended with ∞ and $-\infty$ form a complete lattice.

Exercise 13.14. Prove the following slightly generalized form of the Knaster-Tarski pre-fixpoint theorem: If P is a set of pre-fixpoints of a monoton function on a complete lattice, then $\bigsqcap P$ is a pre-fixpoint too. In other words, the set of pre-fixpoints of a monotone function on a complete lattice is a complete lattice.

Exercise 13.15. According to Lemma 10.38, least pre-fixpoints of monotone functions are also least fixpoints.

1. Show that leastness matters: find a (small!) partial order with a monotone function that has a pre-fixpoint that is not a fixpoint.
2. Show that the reverse implication does not hold: find a partial order with a monotone function that has a least fixpoint that is not a least pre-fixpoint.

Exercise 13.16. The term *collecting* semantics suggests that the reachable states are collected in the following sense: *step* should not transform $\{S\}$ into $\{S'\}$ but into $\{S \cup S'\}$. Show that this makes no difference. That is, prove that if f is a monotone function on sets, then f has the same least fixpoint as $\lambda S.\ S \cup f\ S$.

13.4 Abstract Values thy

The topic of this section is the abstraction from the state sets in the collecting semantics to some type of abstract values. In Section 13.1.2 we had already discussed a first abstraction of state sets

$$(vname \Rightarrow val)\ set \quad \rightsquigarrow \quad vname \Rightarrow val\ set$$

and that it constitutes a first overapproximation. There are so-called relational analyses that avoid this abstraction, but they are more complicated. In a second step, sets of values are abstracted to **abstract values**:

$$val\ set \quad \rightsquigarrow \quad \text{abstract domain}$$

where the **abstract domain** is some type of abstract values that we can compute on. What exactly that type is depends on the analysis. Interval analysis is one example, parity analysis is another. Parity analysis determines if the value of a variable at some point is always even, always odd, or can be either. It is based on the following abstract domain

datatype $parity = Even \mid Odd \mid Either$

that will serve as our running example.

Abstract values represent sets of concerete values *val* and need to come with an ordering that is an abstraction of the subset ordering. Figure 13.5 shows the abstract values on the left and the concrete ones on the right, where $2\mathbb{Z}$ and $2\mathbb{Z}+1$ are the even and the odd integers. The solid lines represent the orderings on the two types. The dashed arrows represent the concretisation function:

> A **concretisation function** maps an abstract value to a set of concrete values.

Concretisation functions will always be named γ, possibly with a suffix. This is the Isabelle definition of the one for *parity*:

fun $\gamma\_parity :: parity \Rightarrow val\ set$ **where**
$\gamma\_parity\ Even \;=\; \{i.\ i\ mod\ 2 = 0\}$
$\gamma\_parity\ Odd \;=\; \{i.\ i\ mod\ 2 = 1\}$
$\gamma\_parity\ Either \;=\; UNIV$

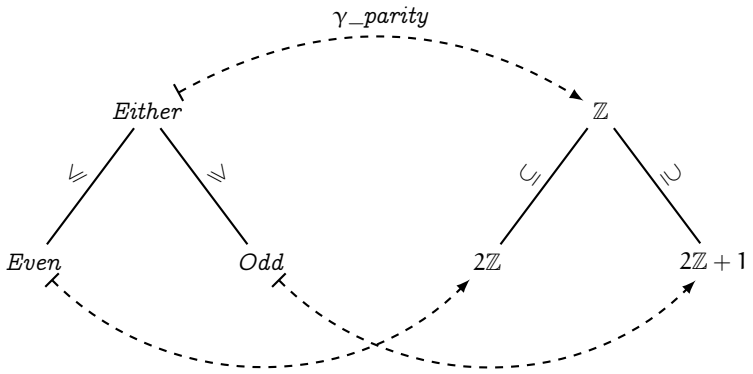


Fig. 13.5. Parity domain and its concretisation

The definition of $\leqslant$ on *parity* is even simpler:

$$(x \leqslant y) = (y = \mathit{Either} \vee x = y)$$

Of course there should be a relationship between the two:

Bigger abstract values represent bigger sets of concrete values.

That is, concretisation functions should be monotone. This is obviously the case in Figure 13.5. Moreover, $\leqslant$ should be a partial order. It is easily proved that $\leqslant$ on *parity* is a partial order.

We want $\leqslant$ to be a partial oder because it abstracts the partial order $\subseteq$. But we will need more. The collecting semantics uses $\cup$ to join the state sets where two computation paths meet, e.g., after an $\mathit{IF{-}THEN{-}ELSE}$. Therefore we require that the abstract domain also provides an operation $\sqcup$ that behaves like $\cup$. That makes it a semilattice:

Definition 13.17. *A type* $'a$ *is a* ***semilattice*** *if it is a partial order and there is a* ***supremum*** *operation* $\sqcup$ *of type* $'a \Rightarrow 'a \Rightarrow 'a$ *that returns the least upper bound of its arguments:*

- *Upper bound:* $x \leqslant x \sqcup y$ *and* $y \leqslant x \sqcup y$
- *Least:* $[\![x \leqslant z;\ y \leqslant z]\!] \Longrightarrow x \sqcup y \leqslant z$

The supremum is also called the **join** operation.

A very useful consequence of the semilattice axioms is the following:

Lemma 13.18. $x \sqcup y \leqslant z \longleftrightarrow x \leqslant z \wedge y \leqslant z$

The proof is left as an exercise.

In addition, we will need an abstract value that corresponds to UNIV.

Definition 13.19. *A partial order has a* ***top element*** $\top$ *if* $x \leqslant \top$ *for all* x.

Thus we will require our abstract domain to be a semilattice with top element, or semilattice with $\top$ for short.

Of course, type *parity* is a semilattice with $\top$:

$x \sqcup y = (\mathit{if}\ x = y\ \mathit{then}\ x\ \mathit{else}\ \mathit{Either})$

$\top = \mathit{Either}$

The proof that the semilattice and top axioms are satisfied is routine.

13.4.1 Type Classes

We will now sketch how abstract concepts like semilattices are modelled with the help of Isabelle's type classes. A **type class** has a name and is defined by

- a set of required functions, the interface, and
- a set of axioms about those functions.

For example, a partial order requires that there is a function $\leqslant$ with certain properties. The type classes introduced above are called *order*, *semilattice_sup*, *top*, and *semilattice_sup_top*.

To show that a type τ belongs to some class C we have to

- define all functions in the interface of C on type τ
- and prove that they satisfy the axioms of C.

This process is called **instantiating** C with τ. For example, above we have instantiated *semilattice_sup_top* with *parity*. Informally we just say that *parity* is a semilattice with $\top$.

Note that the function definitions made when instantiating a class are unlike normal function definitions because we define an already existing but overloaded function (e.g. $\leqslant$) for a new type (e.g. *parity*).

The instantiation of *semilattice_sup_top* with *parity* was unconditional. There is also a conditional version exemplified by the following proposition:

Lemma 13.20. *If* $'a$ *and* $'b$ *are partial orders, so is* $'a \times 'b$.

Proof. The ordering on $'a \times 'b$ is defined in terms of the orderings on $'a$ and $'b$ in the componentwise manner:

$$(x_1,x_2) \leqslant (y_1,y_2) \longleftrightarrow x_1 \leqslant y_1 \wedge x_2 \leqslant y_2$$

The proof that this yields a partial order is straightforward. □

It may be necessary to restrict a type variable $'a$ to be in a particular class C. The notation is $'a :: C$. Here are two examples:

definition *is_true* :: *'a*::*order* ⇒ *bool* **where** *is_true x* = (*x* ⩽ *x*)
lemma *is_true* (*x* :: *'a*::*order*)

If we drop the class constraints in the above definition and lemma statements, they fail because *'a* comes without a class constraint but the given formulas require one. If we drop the type annotations altogether, both the definition and the lemma statement work fine but in the definition the implicitly generated type variable *'a* will be of class *ord*, where *ord* is a predefined superclass of *order* that merely requires an operation ⩽ without any axioms. This means that the lemma will not be provable.

Note that it suffices to constrain one occurrence of a type variable in a given type. The class constraint automatically propagates to the other occurrences of that type variable.

We do not describe the precise syntax for defining and instantiating classes. The semi-formal language used so far is perfectly adequate for our purpose. The interested reader is referred to the Isabelle theories (in particular *Abs_Int*1*_parity* to see how classes are instantiated) and to the tutorial on type classes [41]. If you are familiar with the programming language Haskell you will recognize that Isabelle provides Haskell-style type classes extended with axioms.

13.4.2 From Abstract Values to Abstract States thy

Let *'a* be some abstract domain type. So far we had planned to abstract (*vname* ⇒ *val*) *set* by the **abstract state** type

type_synonym *'a st* = *vname* ⇒ *'a*

but there is a complication: the empty set of states has not meaningful abstraction. The empty state set is important because it arises at unreachable program points, and identifying the latter is of great interest for program oprimisation. Hence we abstract *state set* by

'a st option

where *None* is the abstraction of {}. That is, the abstract interpretation will compute with and annotate programs with values of type *'a st option* instead of *state set*. We will now lift the semilattice structure from *'a* to *'a st option*. All the proofs in this subsection are routine and we leave most of them as exercises whose solutions can be found in the Isabelle theories.

Because states are functions we show as a first step that function spaces preserve semilattices:

Lemma 13.21. *If 'a is a semilattice with* ⊤*, so is 'b* ⇒ *'a, for every type 'b.*

Proof. This class instantiation is already part of theory *Main*. It lifts $\leqslant$, $\sqcup$ and $\top$ from $'a$ to $'b \Rightarrow 'a$ in the canonical pointwise manner:

$$\begin{aligned} f \leqslant g &= \forall x.\ f\ x \leqslant g\ x \\ f \sqcup g &= \lambda x.\ f\ x \sqcup g\ x \\ \top &= \lambda x.\ \top \end{aligned}$$

It is easily shown that the result is a semilattice with $\top$ if $'a$ is one. For example, $f \leqslant f \sqcup g$ holds because $f\ x \leqslant f\ x \sqcup g\ x = (f \sqcup g)\ x$ for all x. □

Similarly, but for $'a\ option$ instead of $'b \Rightarrow 'a$:

Lemma 13.22. *If $'a$ is a semilattice with $\top$, so is $'a\ option$.*

Proof. Here is how $\leqslant$, $\sqcup$ and $\top$ are defined on $'a\ option$:

$$\begin{aligned} &Some\ x \leqslant Some\ y \longleftrightarrow x \leqslant y \\ &None \leqslant \_ \longleftrightarrow True \\ &Some\ \_ \leqslant None \longleftrightarrow False \end{aligned}$$

$$\begin{aligned} &Some\ x \sqcup Some\ y = Some\ (x \sqcup y) \\ &None \sqcup y = y \\ &x \sqcup None = x \end{aligned}$$

$$\top = Some\ \top$$

Figure 13.6 shows how an example of how the ordering is transformed by going from from $'a$ to $'a\ option$. The elements of $'a$ are wrapped up in

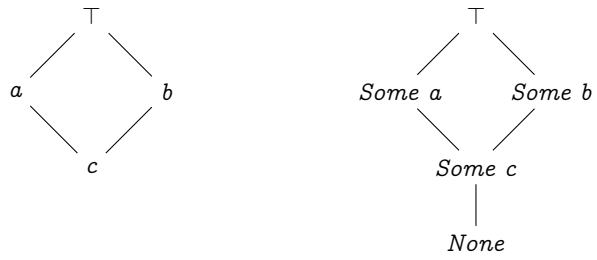


Fig. 13.6. From $'a$ to $'a\ option$

Some without modifying the ordering between them and *None* is adjoined as the least element.

The semilattice properties of $'a\ option$ are proved by exhaustive case analyses. As an example consider the proof of $x \leqslant x \sqcup y$. In each of the cases $x = None$, $y = None$ and $x = Some\ a \wedge y = Some\ b$ it holds by definition and because $a \leqslant a \sqcup b$ on type $'a$. □

Together with Lemma 13.21 we have:

Corollary 13.23. *If* $'a$ *is a semilattice with* $\top$*, so is* $'a$ *st option.*

Now we lift the concretisation function γ from $'a$ to $'a$ *st option*, again separately for $\Rightarrow$ and *option*. We can turn any concretisation function $\gamma :: 'a \Rightarrow 'c\ set$ into one from $'b \Rightarrow 'a$ to $('b \Rightarrow 'c)\ set$:

definition $\gamma\_fun :: ('a \Rightarrow 'c\ set) \Rightarrow ('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'c) set$ **where**
$\gamma\_fun\ \gamma\ F = \{f.\ \forall x.\ f\ x \in \gamma\ (F\ x)\}$

For example, $\gamma\_fun\ \gamma\_parity$ concretises $\lambda x::'b.\ Even$ to the set of all functions of type $'b \Rightarrow int$ whose ranges are even integers.

Lemma 13.24. *If* γ *is monotone, so is* $\gamma\_fun\ \gamma$*.*

Lifting of γ to *option* achieves what we initially set out to do, namely to be able to abstract the empty set:

fun $\gamma\_option :: ('a \Rightarrow 'c\ set) \Rightarrow 'a\ option \Rightarrow 'c\ set$ **where**
$\gamma\_option\ \gamma\ None\ \ = \{\}$
$\gamma\_option\ \gamma\ (Some\ a) = \gamma\ a$

Lemma 13.25. *If* γ *is monotone, so is* $\gamma\_option\ \gamma$*.*

Applying monotone functions to all annotations of a command is also monotone:

Lemma 13.26. *If* γ *is monotone, so is* $map\_acom\ \gamma$*.*

A useful property of monotone functions is the following:

Lemma 13.27. *If* $\varphi :: 'a \Rightarrow 'b$ *is a monotone function between two semilattices then* $\varphi\ a_1 \sqcup \varphi\ a_2 \leqslant \varphi\ (a_1 \sqcup a_2)$*.*

The proof is trivial: $\varphi\ a_1 \sqcup \varphi\ a_2 \leqslant \varphi\ (a_1 \sqcup a_2)$ iff $\varphi\ a_1 \leqslant \varphi\ (a_1 \sqcup a_2)$ for $i = 1,2$ (by Lemma 13.18), and the latter two follow by monotonicity from $a_i \leqslant a_1 \sqcup a_2$.

13.4.3 Exercises

Exercise 13.28. Prove Lemma 13.18 on paper.

13.5 Generic Abstract Interpreter thy

In this section we define a first abstract interpreter which we refine in later sections. It is generic because it is parameterized with the abstract domain. It works like the collecting semantics but operates on the abstract domain instead of state sets. To bring out this similarity we first abstract the collecting semantics' *step* function.

13.5.1 Abstract *Step* Function thy

The *step* function of the collecting semantics (see Figure 13.3) can be generalised as follows.

- Replace *state set* by an arbitrary type $'a$ which must have a $\sqcup$ operation (no semilattice required yet).
- Parameterize *step* with two functions
  $asem :: vname \Rightarrow aexp \Rightarrow 'a \Rightarrow 'a$
  $bsem :: bexp \Rightarrow 'a \Rightarrow 'a$
  that factor out the actions of assignment and boolean expression on $'a$.

The result is the function *Step* displayed in Figure 13.7. Note that we have

```
fun Step :: 'a ⇒ 'a acom ⇒ 'a acom where
Step a (SKIP {_})    = SKIP {a}
Step a (x ::= e {_}) = x ::= e {asem x e a}
Step a (C1;; C2)     = Step a C1;; Step (post C1) C2
Step a (IF b THEN {P1} C1 ELSE {P2} C2 {Q})
  = IF b THEN {bsem b a} Step P1 C1 ELSE {bsem (Not b) a} Step P2 C2
    {post C1 ⊔ post C2}
Step a ({I} WHILE b DO {P} C {Q})
  = {a ⊔ post C}
    WHILE b
    DO {bsem b I}
       Step P C
    {bsem (Not b) I}
```

Fig. 13.7. Definition of *Step*

supressed the parameters *asem* and *bsem* of *Step* in the figure for better readability. In reality, they are there and *step* is defined like this:

$$step = Step\ (\lambda x\ e\ S.\ \{s(x := aval\ e\ s)\ |s.\ s \in S\})\ (\lambda b\ S.\ \{s \in S.\ bval\ b\ s\})$$

(This works because in Isabelle $\cup$ is just nice syntax for $\sqcup$ on sets.) The equations in Figure 13.7 are merely derived from this definition. This way we avoided having to explain the more abstract *Step* early on.

13.5.2 Abstract Interpreter Interface

The abstract interpreter is defined as a parameterized module (a **locale** in Isabelle; see [9] for details). Its parameters are the abstract domain and to-

gether with abstractions of all operations on the concerete type $val = int$. Ther result of the module is an abstract interpreter that operates on the given abstract domain.

In detail, the parameters and their required properties are the following:

Abstract domain A type $'av$ of abstract values.
Must be a semilattice with $\top$.

Concretisation function $\gamma :: 'av \Rightarrow val\ set$
Must be monotone: $a_1 \leqslant a_2 \Longrightarrow \gamma\ a_1 \subseteq \gamma\ a_2$
Must preserve $\top$: $\gamma\ \top = UNIV$

Abstract arithmetic $num' :: val \Rightarrow 'av$
$plus' :: 'av \Rightarrow 'av \Rightarrow 'av$
Must establish and preserve the $i \in \gamma\ a$ relationship:
$i \in \gamma\ (num'\ i)$ (num')
$[\![i_1 \in \gamma\ a_1;\ i_2 \in \gamma\ a_2]\!] \Longrightarrow i_1 + i_2 \in \gamma\ (plus'\ a_1\ a_2)$ $(plus')$

Remarks:

- Every constructor of $aexp$ (except V) must have a counterpart on $'av$.
- num' and $plus'$ abstract N and $Plus$.
- The requirement $i \in \gamma\ (num'\ i)$ could be replaced by $\gamma\ (num'\ i) = \{i\}$. We have chosen the weaker formulation to emphasize that all operations must establish or preserve $i \in \gamma\ a$.
- Functions whose names end with a prime usually operate on $'av$.
- Abstract values are usually called a whereas arithmetic expressions are usually called e now.

From γ we define three lifted concretisation functions:

$\gamma_s :: 'av\ st \Rightarrow state\ set$
$\gamma_s = \gamma\_fun\ \gamma$

$\gamma_o :: 'av\ st\ option \Rightarrow state\ set$
$\gamma_o = \gamma\_option\ \gamma_s$

$\gamma_c :: 'a\ st\ option\ acom \Rightarrow state\ set\ acom$
$\gamma_c = map\_acom\ \gamma_o$

All of them are monotone (see the lemmas in Section 13.4.2).

Now we are ready for the actual abstract interpreter, first for arithmetic expressions, then for commands. Boolean expressions come in a later section.

13.5.3 Abstract Interpretation of Expressions

Abstract interpretation of $aexp$ is unsurprising:

fun $aval' :: aexp \Rightarrow 'av\ st \Rightarrow 'av$ **where**
$aval'\ (N\ i)\ S = num'\ i$
$aval'\ (V\ x)\ S = S\ x$
$aval'\ (Plus\ e_1\ e_2)\ S = plus'\ (aval'\ e_1\ S)\ (aval'\ e_2\ S)$

The correctness lemma expresses that $aval'$ overapproximates $aval$:

Lemma 13.29 (Correctness of $aval'$).
$s \in \gamma_s\ S \Longrightarrow aval\ e\ s \in \gamma\ (aval'\ e\ S)$

Proof. By induction on e with the help of (num') and $(plus')$. □

Example 13.30. We instantiate the interface to our abstract interpreter module with the *parity* domain described earlier:

$\gamma = \gamma\_parity$
$num' = num\_parity$
$plus' = plus\_parity$

where

$num\_parity\ i = (if\ i\ mod\ 2 = 0\ then\ Even\ else\ Odd)$

$plus\_parity\ Even\ Even = Even$
$plus\_parity\ Odd\ Odd = Even$
$plus\_parity\ Even\ Odd = Odd$
$plus\_parity\ Odd\ Even = Odd$
$plus\_parity\ x\ Either = Either$
$plus\_parity\ Either\ y = Either$

We had already discussed that *parity* is a semilattice with $\top$ and γ_parity is monotone. Both $\gamma\_parity\ \top = UNIV$ and (num') are trivial, as is $(plus')$ after an exhaustive case analysis on a_1 and a_2.

Let us call the result $aval'$ of this module instantiation *aval_parity*. Then the following term evaluates to *Even*:

$aval\_parity\ (Plus\ (V\ ''x'')\ (V\ ''y''))\ (\lambda\_.\ Odd)$

13.5.4 Abstract Interpretation of Commands

The abstract interpreter for commands is defined in two steps. First we instantiate *Step* to perform one interpretation step, later we iterate this $step'$ until a pre-fixpoint is found.

$step' :: 'av\ st\ option \Rightarrow 'av\ st\ option\ acom \Rightarrow 'av\ st\ option\ acom$
$step' = Step\ asem\ (\lambda b\ S.\ S)$

where

$asem\ x\ e\ S =$
$(case\ S\ of\ None \Rightarrow None \mid Some\ S \Rightarrow Some\ (S(x := aval'\ e\ S)))$

Remarks:

- From now on the identifier S will (almost) always be of type $'av\ st$ or $'av\ st\ option$.
- Function *asem* updates the abstract state with the abstract value of the expression. The rest is boiler plate to handle the *option* type.
- Boolean expressions are not analysed at all. That is, they have no effect on the state. Compare $\lambda b\ S.\ S$ with $\lambda b\ S.\ \{s \in S.\ bval\ b\ s\}$ in the collecting semantics. The former constitutes a gross overapproximation to be refined later.

Example 13.31. We continue the previous example where we instantiated the abstract interpretation module with the *parity* domain. Let us call the result *step'* of this instantiation *step_parity* and consider the program on the left:

```
x := 3 {None} ;
{None}
WHILE ...
DO {None}
    x := x+2 {None}
{None}
```

Odd			
	Odd		
		Odd	
			Odd
		Odd	

In the table on the right we iterate *step_parity*, i.e. we see how the annotations in $(step\_parity\ \top)^k\ C_0$ change with increasing k (where C_0 is the initial program with *None*s, and by definition $\top = (\lambda\_.\ Either)$). Each row of the table refers to the program annotation in the same row. For compactness, a parity value p represents the state $Some\ (\top(x := p))$. We only show an entry if it differs from the previous step. After 4 steps, there are no more changes, we have reached the least fixpoint.

Exercise: What happens if the 2 in the program is replaced by a 1?

Correctness of *step'* means that it overapproximates *step*:

Corollary 13.32 (Correctness of *step'*).
$step\ (\gamma_o\ S)\ (\gamma_c\ C) \leqslant \gamma_c\ (step'\ S\ C)$

Because *step* and *step'* are defined as instances of *Step*, this is a corollary of a general property of *Step*:

Lemma 13.33. *$Step\ f\ g\ (\gamma_o\ S)\ (\gamma_c\ C) \leqslant \gamma_c\ (Step\ f'\ g'\ S\ C)$ if $f\ x\ e\ (\gamma_o\ S) \subseteq \gamma_o\ (f'\ x\ e\ S)$ and $g\ b\ (\gamma_o\ S) \subseteq \gamma_o\ (g'\ b\ S)$ for all x,e,b.*

Proof. By induction on C. Assignments and boolean expressions are dealt with by the two assumptions. In the *IF* and *WHILE* cases one has to show properties of the form $\gamma_o\ S_1 \cup \gamma_o\ S_2 \subseteq \gamma_o\ (S_1 \sqcup S_2)$; they follow from monotonicity of γ_o by Lemma 13.27. □

The corollary follows directly from the lemma by unfolding the definitions of *step* and *step'*. The requirement $\{s(x := aval\ e\ s)\ |s.\ s \in \gamma_o\ S\} \subseteq \gamma_o\ (asem\ x\ e\ S)$ is a trivial consequence of the operapproximation of *aval* by *aval'*.

13.5.5 Abstract Interpreter

The abstract interpreter iterates *step'* with the help of a library function:

```
while_option :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a option
while_option b f x = (if b x then while_option b f (f x) else Some x)
```

The equation is an executable consequence of the (omitted) definition.

This is a generalization of the *while* combinator used in Section 10.4.2 for the same purpose as now: iterating a function. The difference is that *while_option* returns an optional value to distinguish termination (*Some*) from nontermination (*None*). Of course the execution of *while_option* will simply not terminate if the mathematical result is *None*.

The abstract interpreter *AI* is a search for a pre-fixpoint of *step'* ⊤:

```
AI :: com ⇒ 'av st option acom option
AI c = pfp (step' ⊤) (bot c)

pfp :: ('a ⇒ 'a) ⇒ 'a ⇒ 'a option
pfp f s = while_option (λx. ¬ f x ⩽ x) f s

bot :: com ⇒ 'a option acom
bot c = annotate (λp. None) c
```

Function *pfp* searches a pre-fixpoint of f starting from s. In general, this search need not terminate, but if it does, it has obviously found a pre-fixpoint. This follows from the following property of *while_option*:

```
while_option b c s = Some t ⟹ ¬ b t
```

Function *AI* starts the search from *bot c* where all annotations have been intialised with *None*. Because *None* is the least annotation, *bot c* is the least annotated program (among those structurally equal to c); hence the name.

Now we show partial correctness of *AI*: if abstract interpretation of c terminates and returns an annotated program C, then the concretisation of the annations of C overapproximate the collecting semantics of c.

Lemma 13.34 (Partial correctness of *AI* w.r.t. *CS*).
$AI\ c = Some\ C \Longrightarrow CS\ c \leqslant \gamma_c\ C$

Proof. By definition we may assume $pfp\ (step'\ \top)\ (bot\ c) = Some\ C$. By the above property of *while_option*, C is a pre-fixpoint: $step'\ \top\ C \leqslant C$. By monotonicity we have $\gamma_c\ (step'\ \top\ C) \leqslant \gamma_c\ C$ and because $step'$ overapproximates *step* (Corollary 13.32) we obtain $step\ (\gamma_o\ \top)\ (\gamma_c\ C) \leqslant \gamma_c\ (step'\ \top\ C)$ and thus $\gamma_c\ C$ is a pre-fixpoint of $step\ (\gamma_o\ \top)$. Because *CS* is defined as the least pre-fixpoint of $step\ UNIV$ and because $\gamma_o\ \top = UNIV$ (this is trivial) we obtain $CS\ c \leqslant \gamma_c\ C$ as required. □

13.5.6 Monotonicity

Although we know that if *pfp* terminates, by construction it yields a pre-fixpoint, we don't yet know under what conditions it terminates and which pre-fixpoint is found. This is where monotonicity of *Step* comes in:

Lemma 13.35 (Monotonicity of *Step*). *Let $'a$ be a semilattice and let $f :: vname \Rightarrow aexp \Rightarrow 'a \Rightarrow 'a$ and $g :: bexp \Rightarrow 'a \Rightarrow 'a$ be two monotone functions: $S_1 \leqslant S_2 \Longrightarrow f\ x\ e\ S_1 \leqslant f\ x\ e\ S_2$ and $S_1 \leqslant S_2 \Longrightarrow g\ b\ S_1 \leqslant g\ b\ S_2$. Then Step f g is also monotone, in both arguments: $[\![\ C_1 \leqslant C_2;\ S_1 \leqslant S_2\ ]\!] \Longrightarrow Step\ f\ g\ S_1\ C_1 \leqslant Step\ f\ g\ S_2\ C_2$.*

Proof. The proof is a straightforward computation induction on *Step*. Additionally it needs the easy lemma $C_1 \leqslant C_2 \Longrightarrow post\ C_1 \leqslant post\ C_2$. □

As a corollary we obtain the monotonicity of *step* (already used in the collecting semantics to guarantee that *step* has a least fixed point) because *step* is defined as $Step\ f\ g$ for some monotone f and g in Section 13.5.1. Similarly we have $step' = Step\ asem\ (\lambda b\ S.\ S)$. Although $\lambda b\ S.\ S$ is obviously monotone, *asem* is not necessarily monotone. The **monotone framework** is the extension of the interface to our abstract interpreter with a monotonicity assumption for abstract arithmetic:

Abstract arithmetic must be monotone:
$[\![a_1 \leqslant b_1;\ a_2 \leqslant b_2]\!] \Longrightarrow plus'\ a_1\ a_2 \leqslant plus'\ b_1\ b_2$

Monotonicity of $aval'$ follows by an easy induction on e:

$S_1 \leqslant S_2 \Longrightarrow aval'\ e\ S_1 \leqslant aval'\ e\ S_2$

In the monotone framework, $step'$ is therefore monotone too, in both arguments. Therefore $step'\ \top$ is also monotone.

Monotonicity is not surprising and is only a means to obtain more interesting results, in particular precision and termination of our abstract interpreter. For the rest of this section we assume that we are in the context of the monotone framework.

13.5.7 Precision

So far we have shown correctness (AI overaproximates CS) but nothing about precision. In the worst case AI could annotate every give program with $\top$ everywhere, which would be correct but useless.

We show that AI computes not just any pre-fixoint but the least one. Why is this relevant? Because of the general principle:

> Smaller is better because more precise.

The key observation is that iteration starting below a pre-fixpoint always stays below that pre-fixpoint:

Lemma 13.36. *Let $'a$ be a partial order and let f be a monotone function. If $x_0 \leqslant q$ for some pre-fixpoint q of f, then $f^i\ x_0 \leqslant q$ for all i.*

The proof is a straightforward induction on i and is left as an exercise.

As a consequence, if x_0 is a least element, then $f^i\ x_0 \leqslant q$ for all pre-fixpoints q. In this case $pfp\ f\ x_0 = Some\ p$ tells us that p is not just a pre-fixpoint of f, as observed further above, but by this lemma the least pre-fixpoint and hence, by Lemma 10.38, the least fixpoint.

In the end this merely means that the fixpoint computation is as precise as possible, but f itself may still overapproximate too much. In the abstract interpreter f is $step'\ \top$. Its precision depends on its constituents, which are the parameters of the module: Type $'av$ is required to be a semilattice, and hence $\sqcup$ is not just any upper bound (which would be enough for correctness) but the least one, which ensures optimal precision. But the assumptions (num') and ($plus'$) for the abstract arithmetic only require correctness and would, for example, allow $plus'$ to always return $\top$. We do not dwell on the issue of precision any further but note that the canonical approach to abstract interpretation (see Section 13.10.1) covers it.

13.5.8 Termination

We prove termination of AI under certain conditions. More precisely, we show that pfp can only iterate $step'$ finitely many times. The key insight is the following:

Lemma 13.37. *Let $'a$ be a partial order, let $f :: 'a \Rightarrow 'a$ be a monotone function, and let $x_0 :: 'a$ be such that $x_0 \leqslant f\ x_0$. Then the $f^i\ x_0$ form an ascending chain: $x_0 \leqslant f\ x_0 \leqslant f^2\ x_0 \leqslant \ldots$.*

The proof that $f^i\ x_0 \leqslant f^{i+1}\ x_0$ is by induction on i and is left as an exercise. Note that $x_0 \leqslant f\ x_0$ holds in particular if x_0 is the least element.

Assume the conditions of the lemma hold for f and x_0. Because *pfp* f x_0 computes the $f^i\ x_0$ until a pre-fixpoint is found, we know that the $f^i\ x_0$ form a strictly increasing chain. Therefore we can prove termination of this loop by exhibiting a measure function m into the natural numbers such that $x < y \Longrightarrow m\ x > m\ y$. The latter property is called **anti-monotonicity**.

The following lemma summarizes this reasoning. Note that the relativisation to a set L is necessary because in our application the measure function will not be be anti-monotone on the whole type.

Lemma 13.38. *Let* $'a$ *be a partial order, let* $L :: 'a\ set$*, let* $f \in L \to L$ *be a monotone function, let* $x_0 \in L$ *such that* $x_0 \leqslant f\ x_0$*, and let* $m :: 'a \Rightarrow nat$ *be anti-monotone on* L*. Then* $\exists p.\ pfp\ f\ x_0 = Some\ p$*, i.e.* *pfp* $f\ x_0$ *terminates.*

In our concrete situation f is *step'* $\top$, x_0 is *bot c*, and we now construct a measure function m_c such that $C_1 < C_2 \Longrightarrow m_c\ C_1 > m_c\ C_2$ (for certain C_i). The construction starts from a measure function on $'av$ that is lifted to $'av\ st\ option\ acom$ in several steps.

We extend the interface to the abstract interpretation module by two more parameters that guarantee termination of the analysis:

Measure function and height: $m :: 'av \Rightarrow nat$
$h :: nat$

Must be anti-monotone and bounded:
$a_1 < a_2 \Longrightarrow m\ a_1 > m\ a_2$
$m\ a \leqslant h$

Under these assumptions the ordering $\leqslant$ on $'av$ is of **height** at most h: every chain $a_0 < a_1 < \ldots < a_n$ has height at most h, i.e. $n \leqslant h$. That is, $\leqslant$ on $'av$ is of **finite height**.

Let us first sketch the intuition behind the termination proof. The annotations in an annotated command can be viewed as a big tuple of the abstract values of all variables at all annotation points. For example, if the program has two annotations A_1 and A_2 and three variables x, y, z, then Figure 13.8 depicts some assignment of abstract values a_i to the three variables at the two annotation points. The termination measure is the sum of all $m\ a_i$. Lemma 13.37

A_1			A_2		
x	y	z	x	y	z
a_1	a_2	a_3	a_4	a_5	a_6

Fig. 13.8. Annotations as tuples

showed that in each step of a pre-fixpoint iteration of a monotone function

the ordering strictly increases, i.e. $(a_1,a_2,\ldots) < (b_1,b_2,\ldots)$, which for tuples means $a_i \leqslant b_i$ for all i and $a_k < b_k$ for some k. Anti-monotonicity implies both $m\ a_i \geqslant m\ b_i$ and $m\ a_k > m\ b_k$. Therefore the termination measure strictly decreases. Now for the technical details.

We lift m in three stages to *'av st option acom*:

definition $m_s :: 'av\ st \Rightarrow vname\ set \Rightarrow nat$ **where**
$m_s\ S\ X = (\sum x{\in}X.\ m\ (S\ x))$

fun $m_o :: 'av\ st\ option \Rightarrow vname\ set \Rightarrow nat$ **where**
$m_o\ (Some\ S)\ X = m_s\ S\ X$
$m_o\ None\ X = h * card\ X + 1$

definition $m_c :: 'av\ st\ option\ acom \Rightarrow nat$ **where**
$m_c\ C = (\sum a{\leftarrow}annos\ C.\ m_o\ a\ (vars\ C))$

Measure $m_s\ S\ X$ is defined as the sum of all $m\ (S\ x)$ for $x \in X$. The set X is always finite in our context, namely the set of variables in some command.

By definition, $m_o\ (Some\ S)\ X < m_o\ None\ X$ because all our measures should be anti-monotone and *None* is the least value w.r.t. the ordering on *'av st option*.

In words, $m_c\ C$ sums up the measures of all the annotations in C. The definition of m_c uses the notation $\sum x{\leftarrow}xs.\ f\ x$ for the summation over the elements of a list, in analogy to $\sum x{\in}X.\ f\ x$ for sets. Function *vars* on annotated commands is defined by $vars\ C = vars\ (strip\ C)$ where $vars(c{::}com)$ is the set of variables in c (see Appendix A).

It is easy to show that all three measure functions are bounded:

$finite\ X \Longrightarrow m_s\ S\ X \leqslant h * card\ X$
$finite\ X \Longrightarrow m_o\ opt\ X \leqslant h * card\ X + 1$
$m_c\ C \leqslant length\ (annos\ C) * (h * card\ (vars\ C) + 1)$

The last lemma gives us an upper bound for the number of iterations (= calls of *step'*) that *pfp* and therefore *AI* require: at most $p * (h * n + 1)$ where p and n are the number of annotations and the number of variables in the given command. There are $p * h * n$ many such assignments (see Figure 13.8 for an illustration), and taking into account that an annotation can also be *None* yields $p * (h * n + 1)$ many assignments.

The proof that all three measures are anti-monotone is more complicated. For example, anti-monotonicity of m_s cannot simply be expressed as $finite\ X \Longrightarrow S_1 < S_2 \Longrightarrow m_s\ S_2\ X < m_s\ S_1\ X$ because this does not hold: $S_1 < S_2$ could be due solely to an increase in some variable not in X, with all other variables unchanged, in which case $m_s\ X\ S_1 = m_s\ X\ S_2$. We need to take

into account that X are the variables in the command and that therefore the variables not in X do not change. This "does not change" property (relating two states) can more simply be expressed as "is top" (a property of one state): variables not in the program can only ever have the abstract value $\top$ because we iterate $step'\ \top$. Therefore we define three "is top" predicates relative to some set of variables X:

definition $top\_on_s :: 'av\ st \Rightarrow\ vname\ set \Rightarrow bool$ **where**
$top\_on_s\ S\ X = (\forall x{\in}X.\ S\ x = \top)$

fun $top\_on_o :: 'av\ st\ option \Rightarrow\ vname\ set \Rightarrow bool$ **where**
$top\_on_o\ (Some\ S)\ X = top\_on_s\ S\ X$
$top\_on_o\ None\ X = True$

definition $top\_on_c :: 'av\ st\ option\ acom \Rightarrow bool$ **where**
$top\_on_c\ C\ X = (\forall a{\in}set\ (annos\ C).\ top\_on_o\ a\ X)$

With the help of these predicates we can now formulate that all three measure functions are anti-monotone:

$$[\![finite\ X;\ S_1 = S_2\ on\ -\ X;\ S_1 < S_2]\!] \Longrightarrow m_s\ S_2\ X < m_s\ S_1\ X$$

$$[\![finite\ X;\ top\_on_o\ o_1\ (-\ X);\ top\_on_o\ o_2\ (-\ X);\ o_1 < o_2]\!] \Longrightarrow m_o\ o_2\ X < m_o\ o_1\ X$$

$$[\![top\_on_c\ C_1\ (-\ vars\ C_1);\ top\_on_c\ C_2\ (-\ vars\ C_2);\ C_1 < C_2]\!] \Longrightarrow m_c\ C_2 < m_c\ C_1$$

The proofs involve simple reasoning about sums.

Now we can apply Lemma 13.38 to $AI\ c = pfp\ (step'\ \top)\ (bot\ c)$ to conclude that AI always delivers:

Theorem 13.39. $\exists C.\ AI\ c = Some\ C$

Proof. In Lemma 13.38, let L be $\{C.\ top\_on_c\ C\ (-\ vars\ C)\}$ and let m be m_c. Above we have stated that m_c is anti-monotone on L. Now we examine that the remaining conditions of the lemma are satisfied. We had shown already that $step'\ \top$ is monotone. Showing that it maps L to L requires a little lemma

$$top\_on_c\ C\ (-\ vars\ C) \Longrightarrow top\_on_c\ (step'\ \top\ C)\ (-\ vars\ C)$$

together with the even simpler lemma $strip\ (step'\ S\ C) = strip\ C$ $(*)$. Both follow directly from analogous lemmas about $Step$ which are proved by computation induction on $Step$. Condition $bot\ c \leqslant step'\ \top\ (bot\ c)$ follows from the obvious $strip\ C = c \Longrightarrow bot\ c \leqslant C$ with the help of $(*)$ above. And $bot\ c \in L$ holds because $top\_on_c\ (bot\ c)\ X$ is true for all X by definition. □

Example 13.40. Parity analysis can be shown to terminate because the *parity* semilattice has height 1. Defining the measure function

$$m\_parity\ a = (if\ a = Either\ then\ 0\ else\ 1)$$

Either is given measure 0 because it is the largest and therefore least informative abstract value. Both anti-monotonicity of *m_parity* and $m\_parity\ a \leqslant 1$ are proved automatically.

Note that finite height of $\leqslant$ is actually a bit stronger than necessary to guarantee termination. It is sufficient that there is no infinitely ascending chain $a_0 < a_1 < \ldots$. But then there can be ascending chains of any finite height and we cannot bound the number of iterations of the abstract interpreter, which is problematic in practice.

13.5.9 Executability

Above we have shown that for semilattices of finite height, fixpoint iteration of the abstract interpreter terminates. Yet there is a problem: *AI* is not executable! This is why: *pfp* compares programs with annotations of type *'av st option*; but $S_1 \leqslant S_2$ (where S_1, S_2 :: *'av st*) is defined as $\forall x.\ S_1\ x \leqslant S_2\ x$ where x comes from the infinite type *vname*. Therefore $\leqslant$ on *'av st* is not directly executable.

We learn two things: we need to refine type *st* such that $\leqslant$ becomes executable (this is the subject to the following section), and we need to be careful about claiming termination. We had merely proved that some term *while_option b f s* is equal to *Some*. This implies termination only if *b*, *f* and *s* are executable, which the given *b* is not. In general, there is no logical notion of executability and termination in HOL and such claims are informal. They could be formalised in principle but this would require a formalisation of the code generation process and the target language.

13.5.10 Exercises

Exercise 13.41. Redo Example 13.31 but replace the 2 in the program by 1.

Exercise 13.42. Take the Isabelle theories that define commands, annotated commands, the collecting semantics and the abstract interpreter and extend them with a nondeterministic choice construct as in Exercise 13.10.

The following exercises require class constraints like *'a* :: *order* as introduced in Section 13.4.1.

Exercise 13.43. Prove Lemma 13.36 and Lemma 13.37 in a detailed and readable style. Remember that $f^i\ x$ is input as $(f\ \hat{}\hat{}\ i)\ x$.

Exercise 13.44. Let $'a$ be a complete lattice and let $f :: 'a \Rightarrow 'a$ be a monotone function. Prove that if P is a set of pre-fixpoints of f then $\bigsqcap P$ is a pre-fixpoint of f, too.

13.6 Executable Abstract States thy

This section is all about a clever representation of abstract states. We define a new type $'a\ st$ that is a semilattice where $\sqsubseteq$, $\sqcup$ and $\top$ are executable (provided $'a$ is a semilattice with executable operations). Moreover it supports two operations that behave like function application and function update:

$$\begin{aligned} \mathit{fun} &:: 'a\ st \Rightarrow \mathit{vname} \Rightarrow 'a \\ \mathit{update} &:: 'a\ st \Rightarrow \mathit{vname} \Rightarrow 'a \Rightarrow 'a\ st \end{aligned}$$

This exercise in data refinement is independent of abstract interpretation and a bit technical in places. Hence it can be skipped on first reading. The key point is that our generic abstract interpreter from the previous section only requires two modifications: $S\ x$ is replaced by $\mathit{fun}\ S\ x$ and $S(x := a)$ is replaced by $\mathit{update}\ S\ x\ a$. The proofs carry over either verbatim or they require only local modifications. We merely need one additional lemma which reduces *fun* and *update* to function application and function update: $\mathit{fun}\ (\mathit{update}\ S\ x\ y) = (\mathit{fun}\ S)(x := y)$.

Before we present the new definition of $'a\ st$, we look at two applications, parity analysis and constant propagation.

13.6.1 Parity Analysis thy

We had already instantiated our abstract interpreter with a parity analysis in the previous section. In Example 13.31 we had even shown how the analysis behaves for a simple program. Now that the analyser is executable we apply it to a slightly more interesting example:

$$\begin{aligned} &''x'' ::= N\ 1;; \\ &\mathit{WHILE}\ \mathit{Less}\ (V\ ''x'')\ (N\ 100)\ \mathit{DO}\ ''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ 3) \end{aligned}$$

Same as in Section 13.3.1 we employ a hidden function *show_acom* which displays a function $\{x \mapsto a, y \mapsto b, \ldots\}$ as $\{(x,a), (y,b), \ldots\}$.

Four iterations of the step function result in this annotated command:

$$\begin{aligned} &''x'' ::= N\ 1\ \{\mathit{Some}\ \{(''x'', \mathit{Odd})\}\};; \\ &\{\mathit{Some}\ \{(''x'', \mathit{Odd})\}\} \\ &\mathit{WHILE}\ \mathit{Less}\ (V\ ''x'')\ (N\ 100) \\ &\mathit{DO}\ \{\mathit{Some}\ \{(''x'', \mathit{Odd})\}\} \\ &\quad ''x'' ::= \mathit{Plus}\ (V\ ''x'')\ (N\ 3)\ \{\mathit{Some}\ \{(''x'', \mathit{Even})\}\} \\ &\{\mathit{Some}\ \{(''x'', \mathit{Odd})\}\} \end{aligned}$$

Now the *Even* feeds back into the invariant and waters it down to *Either*:

```
''x'' ::= N 1 {Some {(''x'', Odd)}};;
{Some {(''x'', Either)}}
WHILE Less (V ''x'') (N 100)
DO {Some {(''x'', Odd)}}
     ''x'' ::= Plus (V ''x'') (N 3) {Some {(''x'', Even)}}
{Some {(''x'', Odd)}}
```

A few more steps to propagate *Either* and we reach the least fixpoint:

```
''x'' ::= N 1 {Some {(''x'', Odd)}};;
{Some {(''x'', Either)}}
WHILE Less (V ''x'') (N 100)
DO {Some {(''x'', Either)}}
     ''x'' ::= Plus (V ''x'') (N 3) {Some {(''x'', Either)}}
{Some {(''x'', Either)}}
```

Variable x can be *Either* everywhere, except after $''x'' ::= N\ 1$.

13.6.2 Constant Propagation thy

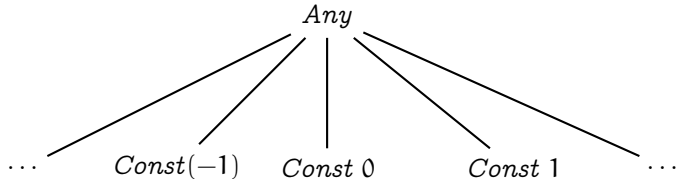
We have already seen a constant propagation analysis in Section 10.2. That one had been an *ad hoc* design. Now we obtain constant propagation (although not the folding optimisation) as an instance of our abstract interpreter. The advantage of instantiating a generic design and correctness proof is not code reuse but the many hours or even days one saves by obtaining the correctness proof for free.

The abstract domain for constant propagation permits us to analyse that the value of a variable at some point is either some interger i ($Const\ i$) or could be anything (Any):

```
datatype const = Const int | Any

fun γ_const :: const ⇒ val set where
γ_const (Const i) = {i}
γ_const Any       = UNIV
```

To reflect γ_const, the ordering must look like this:



In symbols:

$$(x \leqslant y) = (y = Any \vee x = y)$$

$$x \sqcup y = (\mathit{if}\ x = y\ \mathit{then}\ x\ \mathit{else}\ Any)$$

$$\top = Any$$

The best we can do w.r.t. addition on *const* is this:

fun *plus_const* :: *const* ⇒ *const* ⇒ *const* **where**
plus_const (*Const i*) (*Const j*) = *Const* (*i* + *j*)
plus_const _ _ = *Any*

Termination of the analysis is guaranteed because the ordering is again of height 1 as witnessed by this measure function:

m_const x = (*if x* = *Any then* 0 *else* 1)

Having instantiated the abstract interpreter with the above domain we run some examples. We have seen enough iterations by now and merely show the final results.

Some basic arithmetic:

''*x*'' ::= *N* 1 {*Some* {(''*x*'', *Const* 1), (''*y*'', *Any*)}};;
''*x*'' ::= *Plus* (*V* ''*x*'') (*V* ''*x*'') {*Some* {(''*x*'', *Const* 2), (''*y*'', *Any*)}};;
''*x*'' ::= *Plus* (*V* ''*x*'') (*V* ''*y*'') {*Some* {(''*x*'', *Any*), (''*y*'', *Any*)}}

A conditional where both branches set *x* to the same value:

IF Less (*N* 41) (*V* ''*x*'')
THEN {*Some* {(''*x*'', *Any*)}} ''*x*'' ::= *N* 5 {*Some* {(''*x*'', *Const* 5)}}
ELSE {*Some* {(''*x*'', *Any*)}} ''*x*'' ::= *N* 5 {*Some* {(''*x*'', *Const* 5)}}
{*Some* {(''*x*'', *Const* 5)}}

A conditional where both branches set *x* to different values:

IF Less (*N* 41) (*V* ''*x*'')
THEN {*Some* {(''*x*'', *Any*)}} ''*x*'' ::= *N* 5 {*Some* {(''*x*'', *Const* 5)}}
ELSE {*Some* {(''*x*'', *Any*)}} ''*x*'' ::= *N* 6 {*Some* {(''*x*'', *Const* 6)}}
{*Some* {(''*x*'', *Any*)}}

Conditions are ignored, which leads to imprecision:

''*x*'' ::= *N* 42 {*Some* {(''*x*'', *Const* 42)}};;
IF Less (*N* 41) (*V* ''*x*'')
THEN {*Some* {(''*x*'', *Const* 42)}} ''*x*'' ::= *N* 5 {*Some* {(''*x*'', *Const* 5)}}
ELSE {*Some* {(''*x*'', *Const* 42)}} ''*x*'' ::= *N* 6 {*Some* {(''*x*'', *Const* 6)}}
{*Some* {(''*x*'', *Any*)}}

This is where the analyser from Section 10.2 beats our abstract interpreter. Section 13.8 will rectify this deficiency.

As an exercise, compute the following result step by step:

```
''x'' ::= N 0 {Some {(''x'', Const 0), (''y'', Any), (''z'', Any)}};;
''y'' ::= N 0 {Some {(''x'', Const 0), (''y'', Const 0), (''z'', Any)}};;
''z'' ::= N 2
{Some {(''x'', Const 0), (''y'', Const 0), (''z'', Const 2)}};;
{Some {(''x'', Any), (''y'', Any), (''z'', Const 2)}}
WHILE Less (V ''x'') (N 1)
DO {Some {(''x'', Any), (''y'', Any), (''z'', Const 2)}}
     (''x'' ::= V ''y''
      {Some {(''x'', Any), (''y'', Any), (''z'', Const 2)}};;
      ''y'' ::= V ''z''
      {Some {(''x'', Any), (''y'', Const 2), (''z'', Const 2)}})
{Some {(''x'', Any), (''y'', Any), (''z'', Const 2)}}
```

13.6.3 Representation Type *'a st_rep* thy

After these two applications we come back to the challenge of making $\leqslant$ on *st* executable. We exploit that states always have a finite domain, namely the variables in the program. Outside that domain, states do not change. Thus an abstract state can be represented by an association list:

type_synonym *'a st_rep* = (*vname* × *'a*) *list*

Variables of type *'a st_rep* will be called *ps*.

Function *fun_rep* turns an association list into a function. Arguments not present in the association list are mapped to the default value $\top$:

fun *fun_rep* :: (*'a*::*top*) *st_rep* ⇒ *vname* ⇒ *'a* **where**
fun_rep [] = (λ*x*. ⊤)
fun_rep ((*x*, *a*) # *ps*) = (*fun_rep ps*)(*x* := *a*)

Class *top* is predefined. It is the class of all types with a top element $\top$. Here are two examples of the behaviour of *fun_rep*:

fun_rep [(''x'',*a*),(''y'',*b*)] = ((λ*x*. ⊤)(''y'' := *b*))(''x'' := *a*)
fun_rep [(''x'',*a*),(''x'',*b*)] = ((λ*x*. ⊤)(''x'' := *b*))(''x'' := *a*)
= (λ*x*. ⊤)(''x'' := *a*).

Comparing two association lists is easy now: you only need to compare them on their finite "domains", i.e. on *map fst*:

$$\begin{array}{l} less\_eq\_st\_rep\ ps_1\ ps_2 = \\ (\forall x{\in}set\ (map\ fst\ ps_1) \cup set\ (map\ fst\ ps_2).\\ \quad fun\_rep\ ps_1\ x \leqslant fun\_rep\ ps_2\ x) \end{array}$$

Unfortunately this is not a partial order (which is why we gave it the name *less_eq_st_rep* instead of $\leqslant$). For example, both $[(''x'', a), (''y'', b)]$ and $[(''y'', b), (''x'', a)]$ represent the same function and the *less_eq_st_rep* relationship between them holds in both directions, but they are distinct association lists.

13.6.4 Quotient Type *'a st*

Therefore we define the actual new abstract state type as a **quotient type**. In mathematics, the quotient of a set M by some equivalence relation $\sim$ (see Definition 7.7) is written $M/\sim$ and is the set of all equivalence classes $[m]_\sim \subseteq M$, $m \in M$, defined by $[m]_\sim = \{m' \,.\, m \sim m'\}$. In our case the equivalence relation is *eq_st* and it identifies two association lists iff they represent the same function:

$$eq\_st\ ps_1\ ps_2 = (fun\_rep\ ps_1 = fun\_rep\ ps_2)$$

This is precisely the point of the quotient construction: identify data elements that are distinct but abstractly the same.

In Isabelle the quotient type *'a st* is introduced like this:

quotient_type $'a\ st = ('a{::}top)\ st\_rep\ /\ eq\_st$

thereby overwriting the old definition of *'a st*. The class constraint is required because *eq_st* is defined in terms of *fun_rep* which requires *top*. Instead of the mathematical equivalence class notation $[ps]_{eq\_st}$ we write *St ps*. We do not go into further details of **quotient_type**: they are not relevant here and can be found elsewhere [93].

Now we can define the required operations on *'a st*:

$$update\ (St\ ps)\ x\ a = St\ ((x, a)\ \#\ ps)$$

$$fun\ (St\ ps) = fun\_rep\ ps$$

$$\gamma\_st\ \gamma\ S = \{f.\ \forall x.\ f\ x \in \gamma\ (fun\ S\ x)\}$$

$$\top = St\ []$$

$$(St\ ps_1 \leqslant St\ ps_2) = less\_eq\_st\_rep\ ps_1\ ps_2$$

$St\ ps_1 \sqcup St\ ps_2 = St(map2\_st\_rep\ (op \sqcup)\ ps_1\ ps_2)$

fun $map2\_st\_rep ::$
$('a::top \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ st\_rep \Rightarrow 'a\ st\_rep \Rightarrow 'a\ st\_rep$
where
$map2\_st\_rep\ f\ []\ ps_2 = map\ (\lambda(x, y).\ (x, f\ \top\ y))\ ps_2$
$map2\_st\_rep\ f\ ((x, y) \#\ ps_1)\ ps_2 =$
$(let\ y_2 = fun\_rep\ ps_2\ x\ in\ (x, f\ y\ y_2) \#\ map2\_st\_rep\ f\ ps_1\ ps_2)$

Remarks:

- γ_st replaces the earlier γ_fun.
- The auxiliary function $map2\_st\_rep$ is best understood via this lemma:

  $f\ \top\ \top = \top \Longrightarrow$
  $fun\_rep\ (map2\_st\_rep\ f\ ps_1\ ps_2) =$
  $(\lambda x.\ f\ (fun\_rep\ ps_1\ x)\ (fun\_rep\ ps_2\ x))$

  Here and in later uses of $map2\_st\_rep$ the premise $f\ \top\ \top = \top$ is always satisfied.

Most of the above functions are defined with the help of some representative ps of an equivalence class $St\ ps$. That is, they are of the form $f(St\ ps) = \ldots\ ps\ \ldots$. This is not a definition until one has shown that the right-hand side is independent of the choice of ps. Take $fun\ (St\ ps) = fun\_rep\ ps$. It must be shown that if $eq\_st\ ps_1\ ps_2$ then $fun\_rep\ ps_1 = fun\_rep\ ps_2$, which is trivial in this case. Isabelle insists on these proofs but they are routine and we do not go into them here.

Although the previous paragraph is mathematically accurate, the Isabelle details are a bit different because St is not a constructor. Hence equations like $fun\ (St\ ps) = fun\_rep\ ps$ cannot be definitions but are derived lemmas. The actual definition needs to be made on the st_rep level first and is then lifted automatically to the st level. For example, fun_rep is defined first and fun is derived from it almost automatically, except that we have to prove that fun_rep is invariant under eq_st as explained above.

13.6.5 Exercises

Exercise 13.45. Instantiate the abstract interpreter with an analysis of the sign of variables. The basic signs are "+", "−" and "0", but all combinations are also possible: e.g. {0,+} abstracts the set of non-negative integers. Think carefully about how to represent these abstract values. Formalise your analysis by modifing the parity analysis theory Abs_Int1_parity.

Exercise 13.46. Adjust the collecting semantics to keep track of whether a variable is initialised or not, just like in Section 10.1.2. Modify the abstract interpretation theory *Abs_Int*1 accordingly. Instantiate the abstract interpreter with a definite intialisation analysis by modifying *Abs_Int*1*_parity*.

13.7 Analysis of Boolean Expressions thy

So far we have ignored boolean expressions. A first, quite simple analysis could be the following one: Define abstract evaluation of boolean expressions w.r.t. an abstract state such that the result can be Yes, No, or Maybe. In the latter case, the analysis proceeds as before, but if the result is a definite Yes or No, then we can ignore one of the two branches after the test. For example, during constant propagation we may have found out that x has value 5, in which case the value of `x < 5` is definitely No.

We want to be more ambitious. Consider the test `x < 5` and assume that we perform an interval analysis where we have merely determined that x must be in the interval $[0...10]$. Hence `x < 5` evaluates to Maybe, which is no help. However, we would like to infer for `IF x < 5` that in the `THEN`-branch, x is in the interval $[1...4]$ and in the `ELSE`-branch, x is in the interval $[5...10]$, thus improving the analysis of the two branches.

For an arbitrary boolean expression such an analysis can be hard, especially if multiplication is involved. Nevertheless we will present a generic analysis of boolean expressions. It works well for simple examples although it is far from optimal in the general case.

The basic idea of any non-trivial analysis of boolean expressions is to simulate the collecting semantics *step*. Recall from Section 13.5.1 that *step* is a specialised version of *Step* where $bsem\ b\ S = \{s \in S.\ bval\ b\ s\}$ whereas for *step*′ we used $bsem\ b\ S = S$ (see Section 13.5.4). We would like $bsem\ b\ S$, for $S :: 'av\ st$, to be some $S' \leqslant S$ such that

$$\text{No state satisfying } b \text{ is lost: } \{s \in \gamma_s\ S.\ bval\ b\ s\} \subseteq \gamma_s\ S'$$

Of course $S' = S$ satisfies this specification, but we can do better.

Computing $S' \leqslant S$ from S requires a new operation because $\sqcup$ only increases but we need to decrease.

Definition 13.47. *A type* $'a$ *is a* ***lattice*** *if it is a semilattice and there is an* ***infimum*** *operation* $\sqcap :: 'a \Rightarrow 'a \Rightarrow 'a$ *that returns the greatest lower bound of its arguments:*

- *Lower bound:* $x \sqcap y \leqslant x$ *and* $x \sqcap y \leqslant y$
- *Greatest:* $[\![x \leqslant y;\ x \leqslant z]\!] \Longrightarrow x \leqslant y \sqcap z$

A partial order has a ***bottom element*** $\bot$ *if* $\bot \sqsubseteq x$ *for all* x.
A ***bounded lattice*** *is a lattice with both* $\top$ *and* $\bot$.

Isabelle provides the corresponding classes *lattice* and *bounded_lattice*. The infimum is sometimes called the **meet** operation.

We strengthen the abstract interpretation framework as follows:

- The abstract domain $'av$ must be a lattice
- Infimum must be precise: $\gamma\ (a_1 \sqcap a_2) = \gamma\ a_1 \cap \gamma\ a_2$
- $\gamma\ \bot = \{\}$

It is left as an easy exercise to show that $\gamma\ (a_1 \sqcap a_2) \subseteq \gamma\ a_1 \cap \gamma\ a_2$ holds in any lattice because γ is monotone (see Lemma 13.27). Therefore the actual requirement is the other inclusion: $\gamma\ a_1 \cap \gamma\ a_2 \subseteq \gamma\ (a_1 \sqcap a_2)$.

In this context, arithmetic and boolean expressions are analysed. It is a kind of backward analysis. Given an arithmetic expression e, its expected value $a :: 'av$, and some $S :: 'av\ st$, compute some $S' \sqsubseteq S$ such that

$$\{s \in \gamma_s\ S.\ aval\ e\ s \in \gamma\ a\} \subseteq \gamma_s\ S'$$

Roughly speaking, S' overapproximates the subset of S that makes e evaluate to a.

What if e cannot evaluate to a, i.e. $\{s \in \gamma_s\ S.\ aval\ e\ s \in \gamma\ a\} = \{\}$? We need to lift the whole process to S, $S' :: 'av\ st\ option$, then $S' = None$ covers this situation. However, there is a subtlety: *None* is not the only abstraction of the empty set of states. For every $S' :: 'av\ st$ where $fun\ S'\ x = \bot$ for some x, $\gamma_s\ S' = \{\}$ by definition of γ_s because $\gamma\ \bot = \{\}$. Why is this is an issue? Let $S' :: 'av\ st$ such that $\gamma_s\ S' = \{\}$. Both *None* and $Some\ S'$ abstract $\{\}$ but they behave differently in a supremum: $None \sqcup S'' = S''$ but we may have $Some\ S' \sqcup S'' > S''$. Thus $Some\ S'$ can lead to reduced precision, for example when joining the result of the analyses of the `THEN` and `ELSE` branches where one of the two branches is unreachable (leading to $\gamma_s\ S' = \{\}$). Hence, for the sake of precision, we adopt the following policy:

Avoid $Some\ S'$ where $fun\ S'\ x = \bot$ for some x, use *None* instead.

The backward analysis functions will typically be called *inv_f* because they invert some function f on the abstract domain in roughly the following sense. Given some expected result r of f and some arguments *args* of f, *inv_f r args* should return reduced arguments $args' \sqsubseteq args$ such that the reduction from *args* to $args'$ may eliminate anything that does not lead to the desired result r. For correctness we only need to show that that nothing that leads to r is lost by going from *args* to $args'$. The relation $args' \sqsubseteq args$ is just for intuition and precision but in the worst case $args' = \top$ would be correct too.

13.7.1 Backward Analysis of Arithmetic Expressions

The analysis is performed recursively over the expression. Therefore we need an inverse function for every constructor of *aexp* (except V). The abstract arithmetic in the abstract interpreter interface needs to provide two additional executable functions:

- $test\_num' :: val \Rightarrow 'av \Rightarrow bool$ such that

  $test\_num'\ i\ a = (i \in \gamma\ a)$

- $inv\_plus' :: 'av \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av$ such that

  $[\![inv\_plus'\ a\ a_1\ a_2 = (a_1', a_2'); i_1 \in \gamma\ a_1; i_2 \in \gamma\ a_2; i_1 + i_2 \in \gamma\ a]\!]$
  $\Longrightarrow i_1 \in \gamma\ a_1' \wedge i_2 \in \gamma\ a_2'$

A call $inv\_plus'\ a\ a_1\ a_2$ should reduce (a_1, a_2) to some (a_1', a_2') such that every pair of integers represented by (a_1, a_2) that adds up to some integer represented by a is still represented by (a_1', a_2'). For example, for intervals, if $a = [8...10]$, $a_1 = [4...11]$ and $a_2 = [1...9]$, then the smallest possible result is $(a_1', a_2') = ([4...9], [1...6])$; reducing either interval further loses some result in $[8...10]$

If there were further arithmetic functions, their backward analysis would follow the *inv_plus'* pattern. Function *test_num'* is an exception that has been optimised. In standard form it would be called *inv_num'* and would return *'av* instead of *bool*.

With the help of these functions we can formulate the backward analysis of arithmetic expressions:

```
fun inv_aval'' :: aexp ⇒ 'av ⇒ 'av st option ⇒ 'av st option where
inv_aval'' (N n) a S = (if test_num' n a then S else None)
inv_aval'' (V x) a S =
(case S of None ⇒ None
 | Some S ⇒
    let a' = fun S x ⊓ a
    in if a' = ⊥ then None else Some (update S x a'))
inv_aval'' (Plus e1 e2) a S =
(let (a1, a2) = inv_plus' a (aval'' e1 S) (aval'' e2 S)
 in inv_aval'' e1 a1 (inv_aval'' e2 a2 S))

fun aval'' :: aexp ⇒ 'av st option ⇒ 'av where
aval'' e None = ⊥
aval'' e (Some S) = aval' e S
```

The *let*-expression in the $inv\_aval''\ (V\ x)$ equation could be collapsed to $Some\ (update\ S\ x\ (fun\ S\ x \sqcap a))$ except that we would lose precision for the reason discussed above.

Function *aval''* is just a lifted version of *aval'* and its correctness follows easily by induction:

$$s \in \gamma_o\ S \Longrightarrow aval\ a\ s \in \gamma\ (aval''\ a\ S)$$

This permits us to show correctness of *inv_aval''*:

Lemma 13.48 (Correctness of *inv_aval''*).
If $s \in \gamma_o\ S$ *and* $aval\ e\ s \in \gamma\ a$ *then* $s \in \gamma_o\ (inv\_aval''\ e\ a\ S)$.

Proof. By induction on *e*. Case *N* is trivial. Consider case $V\ x$. From $s \in \gamma_o\ S$ we obtain an S' such that $S = Some\ S'$ and $s \in \gamma_s\ S'$, and hence $s\ x \in \gamma\ (fun\ S'\ x)$. We need to show $s \in \gamma_o\ (if\ a' = \bot\ then\ None\ else\ Some\ (update\ S'\ x\ a'))$ where $a' = fun\ S'\ x \sqcap a$. Note that $\gamma\ a' = \gamma\ (fun\ S'\ x \sqcap a) = \gamma\ (fun\ S'\ x) \cap \gamma\ a$ because $\sqcap$ must be precise. Assumption $aval\ (V\ x)\ s \in \gamma\ a$ implies $s\ x \in \gamma\ a$, and thus $s\ x \in \gamma\ a'$ (because $s\ x \in \gamma\ (fun\ S'\ x)$). Therefore the case $a' = \bot$ cannot arise in this proof because by assumptionn $\gamma\ \bot = \{\}$. Now we show the *else* branch is always correct: $s \in \gamma_o\ (Some\ (update\ S'\ x\ a')) = \gamma_s\ (update\ S'\ x\ a')$. This reduces to $s\ x \in \gamma\ a'$ (because $s \in \gamma_s\ S'$), which we have proved above.

The case $Plus\ e_1\ e_2$ is automatic by means of the assumption about *inv_plus'* together with the correctness of *aval''*. □

If the definition of $inv\_aval''\ (Plus\ e_1\ e_2)$ still mystifies you: instead of $inv\_aval''\ e_1\ a_1\ (inv\_aval''\ e_2\ a_2\ S)$ we could have written the more intuitive $inv\_aval''\ e_1\ a_1\ S \sqcap inv\_aval''\ e_2\ a_2\ S$, except that this would have required us to lift $\sqcap$ to *'av st option*, which we avoided by nesting the two *inv_aval''* calls.

13.7.2 Backward Analysis of Boolean Expressions

The analysis needs an inverse function for the arithmetic operator "<", i.e. abstract arithmetic in the abstract interpreter interface needs to provide another executable function:

$$inv\_less' :: bool \Rightarrow 'av \Rightarrow 'av \Rightarrow 'av \times 'av \quad \text{such that}$$

$$[\![inv\_less'\ (i_1 < i_2)\ a_1\ a_2 = (a_1', a_2');\ i_1 \in \gamma\ a_1;\ i_2 \in \gamma\ a_2]\!]$$
$$\Longrightarrow i_1 \in \gamma\ a_1' \wedge i_2 \in \gamma\ a_2'$$

The specification follows the *inv_plus'* pattern.

Now we can define the backward analysis of boolean expresions:

fun $inv\_bval'' :: bexp \Rightarrow bool \Rightarrow 'av\ st\ option \Rightarrow 'av\ st\ option$ **where**
$inv\_bval''\ (Bc\ v)\ res\ S = (if\ v = res\ then\ S\ else\ None)$
$inv\_bval''\ (Not\ b)\ res\ S = inv\_bval''\ b\ (\neg\ res)\ S$

$$
\begin{array}{l}
\mathit{inv\_bval''}\ (\mathit{And}\ b_1\ b_2)\ \mathit{res}\ S = \\
(\mathit{if}\ \mathit{res}\ \mathit{then}\ \mathit{inv\_bval''}\ b_1\ \mathit{True}\ (\mathit{inv\_bval''}\ b_2\ \mathit{True}\ S) \\
\ \mathit{else}\ \mathit{inv\_bval''}\ b_1\ \mathit{False}\ S \sqcup \mathit{inv\_bval''}\ b_2\ \mathit{False}\ S) \\
\mathit{inv\_bval''}\ (\mathit{Less}\ e_1\ e_2)\ \mathit{res}\ S = \\
(\mathit{let}\ (a_1, a_2) = \mathit{inv\_less'}\ \mathit{res}\ (\mathit{aval''}\ e_1\ S)\ (\mathit{aval''}\ e_2\ S) \\
\ \mathit{in}\ \mathit{inv\_aval''}\ e_1\ a_1\ (\mathit{inv\_aval''}\ e_2\ a_2\ S))
\end{array}
$$

The *Bc* and *Not* cases should be clear. The *And* case becomes symmetric when you realise that $\mathit{inv\_bval''}\ b_1\ \mathit{True}\ (\mathit{inv\_bval''}\ b_2\ \mathit{True}\ S)$ is another way of saying $\mathit{inv\_bval''}\ b_1\ \mathit{True}\ S \sqcap \mathit{inv\_bval''}\ b_2\ \mathit{True}\ S$ without needing $\sqcap$. Case *Less* is analogous to case *Plus* of $\mathit{inv\_aval''}$.

Lemma 13.49 (Correctness of $\mathit{inv\_bval''}$).
$s \in \gamma_o\ S \Longrightarrow s \in \gamma_o\ (\mathit{inv\_bval''}\ b\ (\mathit{bval}\ b\ s)\ S)$

Proof. By induction on b with the help of correctness of $\mathit{inv\_aval''}$ and $\mathit{aval''}$. □

13.7.3 Abstract Interpretation

As explained in the introduction to this section, we want to replace $\mathit{bsem}\ b\ S = S$ used in the previous sections with something better. Now we can: $\mathit{step'}$ is defined from *Step* with

$$\mathit{bsem}\ b\ S = \mathit{inv\_bval''}\ b\ \mathit{True}\ S$$

Correctness is straightforward:

Lemma 13.50. $\mathit{AI}\ c = \mathit{Some}\ C \Longrightarrow \mathit{CS}\ c \leqslant \gamma_c\ C$

Proof. Just as for Lemma 13.34, with the help of Corollary 13.32, which needs correctness of $\mathit{inv\_bval''}$. □

This section was free of examples because backward analysis of boolean expressions is most effectively demonstrated on intervals, which require a separate section, the following one.

13.7.4 Exercises

Exercise 13.51. Prove that if $'a$ is a lattice and γ is a monotone function, then $\gamma\ (a_1 \sqcap a_2) \subseteq \gamma\ a_1 \cap \gamma\ a_2$. Give an example of a lattice where $\gamma\ a_1 \cap \gamma\ a_2 \subseteq \gamma\ (a_1 \sqcap a_2)$ does not hold.

Exercise 13.52. Extend the abstract interpreter from the previous section with the simple forward analysis of boolean expressions sketched in the introduction to this section. You need to add a function $\mathit{less'} :: 'av \Rightarrow 'av$

$\Rightarrow$ *bool option* (where *None* indicates "maybe" or "unknown") to locale *Val_semilattice* in theory *Abs_Int0*, define a function *bval'* :: *bexp* $\Rightarrow$ *'av st* $\Rightarrow$ *bool option* in *Abs_Int1* and modify the definition of *step'* to make use of *bval'*. Finally adapt constant propagation analysis in theory *Abs_Int1_const*.

13.8 Interval Analysis thy

We had already seen examples of interval analysis in the informal introduction to abstract interpretation in Section 13.1, but only now are we in the situation to make formal sense of it.

13.8.1 Intervals

Let us start with the type of intervals itself. Our intervals need to include infinite endpoints, otherwise we could not represent any infinite sets. Therefore we base intervals on **extended integers**, i.e. integers extended with ∞ end $-\infty$. The corresponding polymorphic data type is

datatype *'a extended* = *Fin 'a* | ∞ | $-\infty$

and *eint* are the extended integers:

type_synonym *eint* = *int extended*

Type *eint* comes with the usual arithmetic operations.

Constructor *Fin* can be dropped in front of numerals: 5 is as good as *Fin* 5 (except for the type constraint implied by the latter). This applies to input and output of terms. We do not discuss the necessary Isabelle magic.

The naive model of an interval is a pair of extended integers that represent the set of integers between them:

type_synonym *eint2* = *eint* $\times$ *eint*

definition γ*_rep* :: *eint2* $\Rightarrow$ *int set* **where**
γ*_rep p* = (*let* (*l*, *h*) = *p in* {*i*. *l* $\leqslant$ *Fin i* $\wedge$ *Fin i* $\leqslant$ *h*})

For example, γ*_rep* (0, 3) = {0, 1, 2, 3} and γ*_rep*(0,∞) = {0, 1, 2, ...}.

Unfortunately this results in infinitely many empty intervals, namely all pairs (*l*, *h*) where *h* < *l*. Hence we need to perform a quotient construction, just like for *st* in Section 13.6.4. We consider two intervals equivalent if they represent the same set of integers:

definition $eq\_ivl :: eint2 \Rightarrow eint2 \Rightarrow bool$ **where**
$eq\_ivl\ p_1\ p_2 = (\gamma\_rep\ p_1 = \gamma\_rep\ p_2)$

quotient_type $ivl = eint2\ /\ eq\_ivl$

Most of the time we blur the distinction between *ivl* and *eint2* by introducing the abbreviation $[l,h] :: ivl$, where $l,h :: eint$, for the equivalence class of all pairs equivalent to $(l,\ h)$ w.r.t. *eq_ivl*. Unless $\gamma\_rep\ (l,\ h) = \{\}$, this equivalence class only contains $(l,\ h)$. Moreover let $\bot$ be the empty interval, i.e. $[l,\ h]$ for any $h < l$. Note that there are two corner cases where $[l,\ h] = \bot$ does not imply $h < l$: $[\infty,\ \infty]$ and $[-\infty,\ -\infty]$.

We will now "define" most functions on intervals by pattern matching on $[l,\ h]$. Here is a first example:

$\gamma\_ivl :: ivl \Rightarrow int\ set$
$\gamma\_ivl\ [l,\ h] = \{i.\ l \leqslant Fin\ i \wedge Fin\ i \leqslant h\}$

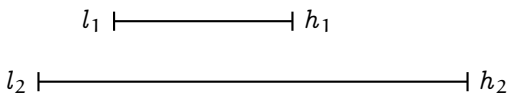
Recall from the closing paragraph of Section 13.6.4 that the actual definitions must be made on the *eint2* level and that the pattern matching equations are derived lemmas.

13.8.2 Intervals as a Bounded Lattice

The following definition turns *ivl* into a partial order:

$([l_1,\ h_1] \leqslant [l_2,\ h_2]) =$
$(if\ [l_1,\ h_1] = \bot\ then\ True$
$\ else\ if\ [l_2,\ h_2] = \bot\ then\ False\ else\ l_2 \leqslant l_1 \wedge h_1 \leqslant h_2)$

The ordering can be visualised like this:



Making *ivl* a bounded lattice is not hard either:

$[l_1,\ h_1] \sqcup [l_2,\ h_2] =$
$(if\ [l_1,\ h_1] = \bot\ then\ [l_2,\ h_2]$
$\ else\ if\ [l_2,\ h_2] = \bot\ then\ [l_1,\ h_1]\ else\ [min\ l_1\ l_2,\ max\ h_1\ h_2])$

$[l_1,\ h_1] \sqcap [l_2,\ h_2] = [max\ l_1\ l_2,\ min\ h_1\ h_2]$

$\top = [-\infty,\ \infty]$

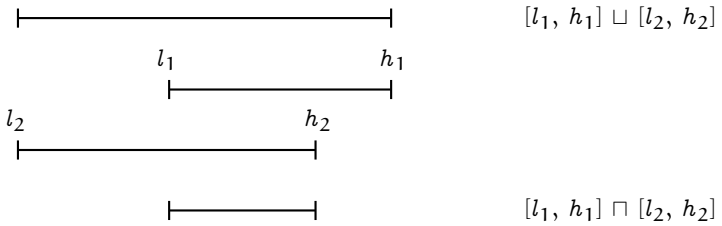


Fig. 13.9. Supremum and infimum of overlapping intervals

The supremum and infimum of two overlapping intervals is shown graphically in Figure 13.9. The loss of precision resulting from the supremum of two non-overlapping intervals is show in Figure 13.10; their empty infimum is not shown.

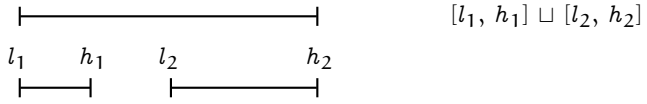


Fig. 13.10. Supremum of non-overlapping intervals

Intervals do in fact form a complete lattice, but we do not need this fact. It is easy to see that $\sqcap$ is precise:

$$\gamma\_ivl\ (iv_1 \sqcap iv_2) = \gamma\_ivl\ iv_1 \cap \gamma\_ivl\ iv_2$$

no matter if the two intervals overlap or not.

We will at best sketch proofs to do with intervals. These proofs are conceptually simple, you just need to make a case distinction over all extened integers in the proposition. More importantly, intervals are not germane to abstract interpretation. The interested reader should consult the Isabelle theories for the proof details.

13.8.3 Arithmetic on Intervals

Addition, negation and subtraction are straightforward:

$$[l_1, h_1] + [l_2, h_2] =$$
$$(\mathit{if}\ [l_1, h_1] = \bot \vee [l_2, h_2] = \bot\ \mathit{then}\ \bot\ \mathit{else}\ [l_1 + l_2, h_1 + h_2])$$

$$-\,[l, h] = [-\,h, -\,l]$$

$$iv_1 - iv_2 = iv_1 + - iv_2$$

So is the inverse function for addition:

$$inv\_plus\_ivl\ iv\ iv_1\ iv_2 = (iv_1 \sqcap (iv - iv_2),\ iv_2 \sqcap (iv - iv_1))$$

We sketch why this definition meets the requirement that if $i_1 \in \gamma\_ivl\ iv_1$, $i_2 \in \gamma\_ivl\ iv_2$ and $i_1 + i_2 \in \gamma\_ivl\ iv$ then $i_1 \in \gamma\_ivl\ (iv_1 \sqcap (iv - iv_2))$ $= \gamma\_ivl\ iv_1 \cap \gamma\_ivl\ (iv - iv_2)$ (and similarly for i_2). By assumption $i_1 \in \gamma\_ivl\ iv_1$. Moreover $i_1 \in \gamma\_ivl\ (iv - iv_2) = \{i_1.\ \exists\, i{\in}\gamma\_ivl\ iv.\ \exists\, i_2{\in}\gamma\_ivl\ iv_2.\ i_1 = i - i_2\}$ also holds, by assumptions $i_2 \in \gamma\_ivl\ iv_2$ and $i_1 + i_2 \in \gamma\_ivl\ iv$.

Backward analysis for "$<$" is more complicated because the empty intervals need to be treated specially and because there are two cases depending on the expected result $res :: bool$:

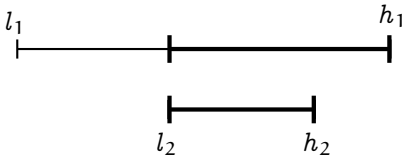
$$\begin{array}{l} inv\_less\_ivl\ res\ iv_1\ iv_2 = \\ (if\ res \\ \ then\ (iv_1 \sqcap (below\ iv_2 - [1,\ 1]),\ iv_2 \sqcap (above\ iv_1 + [1,\ 1])) \\ \ else\ (iv_1 \sqcap above\ iv_2,\ iv_2 \sqcap below\ iv_1)) \end{array}$$

$$above\ [l,\ h] = (if\ [l,\ h] = \bot\ then\ \bot\ else\ [l,\ \infty])$$

$$below\ [l,\ h] = (if\ [l,\ h] = \bot\ then\ \bot\ else\ [-\infty,\ h])$$

The correctness proof follows the pattern of the one for *inv_plus_ivl*.

For a visualisation of *inv_less_ivl* we focus on the "$\geqslant$" case, i.e. $\neg\ res$, and consider the following situation, where $iv_1 = [l_1,\ h_1]$, $iv_2 = [l_2,\ h_2]$ and the thick lines indicate the result:



All those points that can definitely not lead to the first interval being $\geqslant$ the second have been eliminated. Of course this is just one of finitely many situations how the two intervals can be positioned relative to each other.

The "$<$" case in the definition of *inv_less_ivl* is dual to the "$\geqslant$" case but adjusted by 1 because the ordering is strict.

13.8.4 Abstract Interpretation

Now we instantiate the abstract interpreter interface with the interval domain: $num' = \lambda i.\ [Fin\ i,\ Fin\ i]$, $plus' = op\ +$, $test\_num' = in\_ivl$ where *in_ivl*

$i\ [l, h] = (l \leqslant Fin\ i \wedge Fin\ i \leqslant h)$, $inv\_plus' = inv\_plus\_ivl$, $inv\_less' = inv\_less\_ivl$. In the preceding subsections we have already discussed that all requirements hold.

Let us now analyse some concrete programs. We mostly show the final results of the analyses only. For a start, we can do better than the naive constant propagation in Section 13.6.2:

$''x'' ::= N\ 42\ \{Some\ \{(''x'', [42, 42])\}\};;$
$IF\ Less\ (N\ 41)\ (V\ ''x'')$
$THEN\ \{Some\ \{(''x'', [42, 42])\}\}\ ''x'' ::= N\ 5\ \{Some\ \{(''x'', [5, 5])\}\}$
$ELSE\ \{None\}\ ''x'' ::= N\ 6\ \{None\}$
$\{Some\ \{(''x'', [5, 5])\}\}$

This time the analysis detects that the $ELSE$ branch is unreachable. Of course the constant propagation in Section 10.2 can deal with this example equally well. Now we look at examples beyond constant propagation. Here is one that demonstrates that the analysis of boolean expressions very well:

$''y'' ::= N\ 7\ \{Some\ \{(''x'', [-\infty, \infty]), (''y'', [7, 7])\}\};;$
$IF\ Less\ (V\ ''x'')\ (V\ ''y'')$
$THEN\ \{Some\ \{(''x'', [-\infty, 6]), (''y'', [7, 7])\}\}$
$\quad ''y'' ::= Plus\ (V\ ''y'')\ (V\ ''x'')$
$\quad \{Some\ \{(''x'', [-\infty, 6]), (''y'', [-\infty, 13])\}\}$
$ELSE\ \{Some\ \{(''x'', [7, \infty]), (''y'', [7, 7])\}\}$
$\quad ''x'' ::= Plus\ (V\ ''x'')\ (V\ ''y'')$
$\quad \{Some\ \{(''x'', [14, \infty]), (''y'', [7, 7])\}\}$
$\{Some\ \{(''x'', [-\infty, \infty]), (''y'', [-\infty, 13])\}\}$

Let us now analyse some $WHILE$ loops.

$\{Some\ \{(''x'', [-\infty, \infty])\}\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ \{Some\ \{(''x'', [-\infty, 99])\}\}$
$\quad ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ \{(''x'', [-\infty, 100])\}\}$
$\{Some\ \{(''x'', [100, \infty])\}\}$

This is very nice, all the intervals are precise and the analysis only takes three steps to stabilise. Unfortunately this is a coincidence. The analysis of almost the same program, but with x initialised, is less well-behaved. After five steps we have

$''x'' ::= N\ 0\ \{Some\ \{(''x'', [0, 0])\}\};;$
$\{Some\ \{(''x'', [0, 1])\}\}$
$WHILE\ Less\ (V\ ''x'')\ (N\ 100)$
$DO\ \{Some\ \{(''x'', [0, 0])\}\}$
$\quad ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ \{(''x'', [1, 1])\}\}$

$\{None\}$

The interval [0, 1] will increase slowly until it reaches the invariant [0, 100]:

$$\begin{array}{l} ''x'' ::= N\ 0\ \{Some\ \{(''x'', [0, 0])\}\};; \\ \{Some\ \{(''x'', [0, 100])\}\} \\ WHILE\ Less\ (V\ ''x'')\ (N\ 100) \\ DO\ \{Some\ \{(''x'', [0, 99])\}\} \\ \quad ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ \{(''x'', [1, 100])\}\} \\ \{Some\ \{(''x'', [100, 100])\}\} \end{array} \tag{13.1}$$

The result is as precise as possible, i.e. a least fixpoint, but it takes a while to get there.

In general, the analysis of this loop, for an upper bound of n instead of 100, takes $\Theta(n)$ steps (if $n \geqslant 0$). This is not nice. We might as well run the program directly (which this analysis more or less does). Therefore it is not surprising that the analysis of a nonterminating program may not terminate either. The following annotated command is the result of 50 steps:

$$\begin{array}{l} ''x'' ::= N\ 0\ \{Some\ \{(''x'', [0, 0])\}\};; \\ \{Some\ \{(''x'', [0, 16])\}\} \\ WHILE\ Less\ (N\ {-1})\ (V\ ''x'') \\ DO\ \{Some\ \{(''x'', [0, 15])\}\} \\ \quad ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ \{(''x'', [1, 16])\}\} \\ \{None\} \end{array} \tag{13.2}$$

The interval for x keeps increasing indefinitely. The problem is that *ivl* is not of finite height: $[0, 0] < [0, 1] < [0, 2] < \ldots$. The Alexandrian solution to this is quite brutal: if the analysis sees the beginning of a possibly infinite ascending chain, it overapproximates wildly and jumps to a much larger point, namely $[0, \infty]$ in this case. An even more brutal solution would be to jump directly to $\top$ to avoid nontermination.

13.8.5 Exercises

Exercise 13.53. Construct a terminating program where interval analysis does not terminate. Hint: a loop with two initialised variables; remember that our analyses cannot keep track of relationships between variables.

13.9 Widening and Narrowing thy

13.9.1 Widening

Widening abstracts the idea sketched at the end of the previous section: if an iteration appears to diverge, jump upwards in the ordering to avoid nonter-

mination or at least accelerate termination, possibly at the cost of precision. This is the purpose of overloaded **widening operators**:

$$op \bigtriangledown :: 'a \Rightarrow 'a \Rightarrow 'a \quad \text{such that}$$
$$x \leqslant x \bigtriangledown y \quad \text{and} \quad y \leqslant x \bigtriangledown y$$

which is equivalent to $x \sqcup y \leqslant x \bigtriangledown y$. This property is only needed for the termination proof later on. We assume that the abstract domain provides a widening operator.

Iteration with widening means replacing $x_{i+1} = f\, x_i$ by $x_{i+1} = x_i \bigtriangledown f\, x_i$. That is, we apply widening in each step. Pre-fixpoint iteration with widening is defined for any type $'a$ that provides $\leqslant$ and $\bigtriangledown$:

definition *iter_widen* $:: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$ **where**
$iter\_widen\ f = while\_option\ (\lambda x.\ \neg\ f\ x \leqslant x)\ (\lambda x.\ x \bigtriangledown f\ x)$

Comparing *iter_widen* with *pfp*, we don't iterate f but $\lambda x.\ x \bigtriangledown f\ x$. The condition $\neg\ f\ x \leqslant x$ still guarantees that if we terminate we have obtained a pre-fixpoint, but it no longer needs to be the least one because widening may have jumped over it. That is, we trade precision for termination. As a consequence, Lemma 10.38 no longer applies and for the first time we may actually obtain a pre-fixpoint that is not a fixpoint.

This behaviour of widening is visualised in Figure 13.11. The normal iteration of f takes a number of steps to get close to the least fixpoint of f (where f intersects with the diagonal) and it is not clear if it will reach it in a finite number of steps. Iteration with widening, however, boldly jumps over the least fixpoint to some pre-fixpoint. Note that any x where $f\ x$ is below the diagonal is a pre-fixpoint.

Example 13.54. A simple instance of widening for intervals compares two intervals and jumps to ∞ if the upper bound increases and to $-\infty$ if the lower bound decreases:

$$[l_1, h_1] \bigtriangledown [l_2, h_2] = [l, h]$$
$$\text{where} \quad l = (\mathit{if}\ l_1 > l_2\ \mathit{then}\ -\infty\ \mathit{else}\ l_1)$$
$$h = (\mathit{if}\ h_1 < h_2\ \mathit{then}\ \infty\ \mathit{else}\ h_1)$$

For example: $[0, 1] \bigtriangledown [0, 2] = [0, \infty]$
$[0, 2] \bigtriangledown [0, 1] = [0, 2]$
$[1, 2] \bigtriangledown [0, 5] = [-\infty, \infty]$

The first two examples show that although the symbol $\bigtriangledown$ looks symmetric, the operator need not be commutative: its two arguments are the previous and the next value in an iteration and it is important which one is which.

The termination argument for *iter_widen* on intervals goes roughly like this: if we have not reached a pre-fixpoint yet, i.e. $\neg\ [l_2, h_2] \leqslant [l_1, h_1]$, then

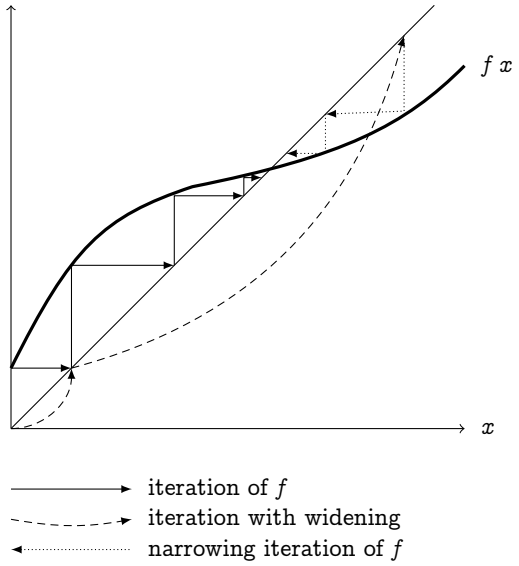


Fig. 13.11. Widening and narrowing

either $l_1 > l_2$ or $h_1 < h_2$ and hence at least one of the two bounds jumps to infinity, which can happen at most twice.

Widening operators can be lifted from $'a$ to other types:

- $'a\ st$:
  If st is the function space (Section 13.5), then $S_1 \triangledown S_2 = (\lambda x.\ S_1\ x \triangledown S_2\ x)$.
  For the executable version of st in Section 13.6 it is

  $St\ ps_1 \triangledown St\ ps_2 = St(map2\_st\_rep\ (op \triangledown)\ ps_1\ ps_2)$

- $'a\ option$:

  $$\begin{array}{rcl} None \triangledown x & = & x \\ x \triangledown None & = & x \\ Some\ x \triangledown Some\ y & = & Some\ (x \triangledown y) \end{array}$$

- $'a\ acom$:

  $C_1 \triangledown C_2 = map2\_acom\ (op \triangledown)\ C_1\ C_2$

  where $map2\_acom\ f$ combines two (structurally equal) commands into one by combining annotations at the same program point with f:

  $map2\_acom\ f\ C_1\ C_2 =$
  $annotate\ (\lambda p.\ f\ (anno\ C_1\ p)\ (anno\ C_2\ p))\ (strip\ C_1)$

Hence we can now perform widening and thus *iter_widen* on our annotated commands of type *'av st option acom*. We demonstrate the effect on two example programs. That is, we iterate $\lambda C.\ C \triangledown \mathit{step\_ivl}\ \top\ C$ where *step_ivl* is the specialisation of *step'* for intervals that came out of Section 13.8.4 (the name *step_ivl* was not mentioned there). The only difference to Section 13.8.4 is the widening after each step.

Our first example is the nonterminating loop (13.2) from the previous section. After four steps it now looks like this:

```
''x'' ::= N 0 {Some {(''x'', [0, 0])}};;
{Some {(''x'', [0, 0])}}
WHILE Less (N -1) (V ''x'')
DO {Some {(''x'', [0, 0])}}
    ''x'' ::= Plus (V ''x'') (N 1) {Some {(''x'', [1, 1])}}
{None}
```

Previously, in the next step the annotation at the head of the loop would change from $[0, 0]$ to $[0, 1]$, now this is followed by widening. Because $[0, 0] \triangledown [0, 1] = [0, \infty]$ we obtain

```
''x'' ::= N 0 {Some {(''x'', [0, 0])}};;
{Some {(''x'', [0, ∞])}}
WHILE Less (N -1) (V ''x'')
DO {Some {(''x'', [0, 0])}}
    ''x'' ::= Plus (V ''x'') (N 1) {Some {(''x'', [1, 1])}}
{None}
```

Two more steps and we have reached a pre-fixpoint:

```
''x'' ::= N 0 {Some {(''x'', [0, 0])}};;
{Some {(''x'', [0, ∞])}}
WHILE Less (N -1) (V ''x'')
DO {Some {(''x'', [0, ∞])}}
    ''x'' ::= Plus (V ''x'') (N 1) {Some {(''x'', [1, ∞])}}
{None}
```

This is very nice because we have actually reached the least fixpoint.

The details of what happened are shown in Table 13.1 where A_0 to A_4 are the five annotations from top to bottom. We start with the situation after four steps. Each iteration step is displayed as a block of two columns: first the result of applying the step function to the previous column, then the result of widening that result with previous column. The last column is the least pre-fixpoint. Empty entries mean that nothing has changed.

Now we look at the program were previously we needed $\Theta(n)$ steps to reach the least fixpoint. Now we reach a pre-fixpoint very quickly:

		f	∇	f	∇	f	∇
A_0	$[0, 0]$						$[0, 0]$
A_1	$[0, 0]$	$[0, 1]$	$[0, \infty]$	$[0, 1]$	$[0, \infty]$	$[0, \infty]$	$[0, \infty]$
A_2	$[0, 0]$			$[0, \infty]$	$[0, \infty]$	$[0, \infty]$	$[0, \infty]$
A_3	$[1, 1]$					$[1, \infty]$	$[1, \infty]$
A_4	*None*						*None*

Table 13.1. Windening example

$$
\begin{array}{l}
''x'' ::= N\ 0\ \{Some\ \{(''x'', [0, 0])\}\};; \\
\{Some\ \{(''x'', [0, \infty])\}\} \\
WHILE\ Less\ (V\ ''x'')\ (N\ 100) \\
DO\ \{Some\ \{(''x'', [0, \infty])\}\} \\
\quad ''x'' ::= Plus\ (V\ ''x'')\ (N\ 1)\ \{Some\ \{(''x'', [1, \infty])\}\} \\
\{Some\ \{(''x'', [100, \infty])\}\}
\end{array}
\tag{13.3}
$$

This takes only a constant number of steps, independent of the upper bound of the loop, but the result is worse than previously (13.1): precise upper bounds have been replaced by ∞.

13.9.2 Narrowing

Narrowing is the attempt to improve the potentially imprecise pre-fixpoint p obtained by widening. It assumes f is monotone. Then we can just iterate f:

$$p \geqslant f(p) \geqslant f^2(p) \geqslant \ldots$$

Each $f^i(p)$ is still a pre-fixpoint, and we can stop improving any time, especially if we reach a fixpoint. In Figure 13.11 you can see the effect of two such iterations of f back from an imprecise pre-fixpoint towards the least fixpoint. As an example, start from the imprecise widening analysis (13.3): four applications of *step_ivl* $\top$ take us to the least fixpoint (13.1) that we had computed without widening and narrowing in many more steps.

We may have to stop iterating f before we reach a fixpoint if we want to terminate (quickly) because there may be infinitely descending chains:

$$[0,\infty] > [1,\infty] > [2,\infty] > \ldots$$

This is where the overloaded **narrowing operators** come in:

$$op\ \triangle :: 'a \Rightarrow 'a \Rightarrow 'a \quad \text{such that}$$
$$y \leqslant x \Longrightarrow y \leqslant x \triangle y \leqslant x$$

We assume that the abstract domain provides a narrowing operator.

Iteration with narrowing means replacing $x_{i+1} = f\,x_i$ by $x_{i+1} = x_i \triangle f\,x_i$. Now assume that x_i is a pre-fixpoint of f. Then $x_i \triangle f\,x_i$ is again a pre-fixpoint of f: because $f\,x_i \leqslant x_i$ and $\triangle$ is a narrowing operator we have $f\,x_i \leqslant$

$x_i \triangle f\ x_i \leqslant x_i$ and hence $f\ (x_i \triangle f\ x_i) \leqslant f\ x_i \leqslant x_i \triangle f\ x_i$ by monotonicity. That is, iteration of a monotone function with a narrowing operator preserves the pre-fixpoint property.

definition $iter\_narrow :: ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$ **where**
$iter\_narrow\ f = while\_option\ (\lambda x.\ x \triangle f\ x < x)\ (\lambda x.\ x \triangle f\ x)$

In contrast to widening, we are not looking for a pre-fixpoint (if we start from one, it remains one), but we terminate as soon as we no longer make progress, i.e. no longer strictly dicrease. The narrowing operator can enforce termination by making $x \triangle f\ x$ return x.

Example 13.55. The narrowing operator for intervals only changes one of the interval bounds if it can be improved from infinity to some finite value:

$[l_1, h_1] \triangle [l_2, h_2] = [l, h]$
where $l = (if\ l_1 = -\infty\ then\ l_2\ else\ l_1)$
$h = (if\ h_1 = \infty\ then\ h_2\ else\ h_1)$

For example: $[0, \infty] \triangle [0, 100] = [0, 100]$
$[0, 100] \triangle [0, 0] = [0, 100]$
$[0, 0] \triangle [0, 100] = [0, 0]$
Narrowing operators need not be commutative either.

The termination argument for *iter_narrow* on intervals is very intuitive: finite interval bounds remain unchanged, therefore it takes at most two steps until both bounds have become finite and *iter_narrow* terminates.

Narrowing operators are lifted to $'a\ st$ and $'a\ acom$ verbatim like widening operators, with $\triangledown$ replaced by $\triangle$. On $'a\ option$ narrowing is dual to widening:

$$\begin{aligned} None \triangle x &= None \\ x \triangle None &= None \\ Some\ x \triangle Some\ y &= Some\ (x \triangle y) \end{aligned}$$

Hence we can now perform narrowing with *iter_narrow* on our annotated commands of type $'av\ st\ option\ acom$. Starting from the imprecise result (13.3) of widening, $iter\_narrow\ (step\_ivl\ \top)$ also reaches the least fixpoint (13.1) that iterating $step\_ivl\ \top$ had reached without the narrowing operator (see above). In general the narrowing operator is required to ensure termination but it may also lose precision by terminating early.

13.9.3 Abstract Interpretation

The abstract interpreter with widening and narrowing is defined like *AI*

$AI\_wn\ c = pfp\_wn\ (step'\ \top)\ (bot\ c)$

but instead of the simple iterator *pfp* the composition of widening and narrowing is used:

$$pfp\_wn\ f\ x =$$
$$(case\ iter\_widen\ f\ x\ of\ None \Rightarrow None \mid Some\ p \Rightarrow iter\_narrow\ f\ p)$$

In the monotone framework we can show partial correctness of AI_wn:

Lemma 13.56. $AI\_wn\ c = Some\ C \Longrightarrow CS\ c \leqslant \gamma_c\ C$

Proof. The proof is the same as for Lemma 13.34, but now we need that if $pfp\_wn\ (step'\ \top)\ C = Some\ C'$, then C' is a pre-fixpoint. For *iter_widen* this is guaranteed by the exit condition $f\ x \leqslant x$. Above we have shown that if f is monotone, then the pre-fixpoint property is an invariant of $iter\_narrow\ f$. Because $f = step'\ \top$ is monotone in the monotone framework, the narrowing phase returns a pre-fixpoint. Thus *pfp_wn* returns a pre-fixpoint if it terminates (returns *Some*). □

13.9.4 Termination of Widening

The termination proof for *iter_widen* is similar to the termination proof in Section 13.5.8. We again require a measure function $m :: 'av \Rightarrow nat$ and a height $h :: nat$ that bounds m: $m\ a \leqslant h$. The key requirement is that if we have not reached a pre-fixpoint yet, widening strictly decreases m:

$$\neg\ a_2 \leqslant a_1 \Longrightarrow m\ (a_1 \triangledown a_2) < m\ a_1 \tag{13.4}$$

Clearly there can only be h many widening steps.

In addition we require anti-monotonicity w.r.t. $\leqslant$, not $<$:

$$a_1 \leqslant a_2 \Longrightarrow m\ a_1 \geqslant m\ a_2$$

Note that this does not mean that the ordering is of finite height. Otherwise we could not apply it to intervals further down.

Let us first sketch the intuition behind the termination proof. In Section 13.5.8 we relied on monotonicity to argue that pre-fixpoint iteration gives rise to a strictly increasing chain. Unfortunately, $\lambda x.\ x \triangledown f\ x$ need not be monotone, even if f is. For example, on intervals let f be constantly $[0,1]$. Then $[0,0] \leqslant [0,1]$ but $[0,0] \triangledown f\ [0,0] = [0,\infty] \not\leqslant [0,1] = [0,1] \triangledown f\ [0,1]$. Nevertheless *iter_widen* gives rise to a strictly decreasing m-chain: if $\neg\ f\ x \leqslant x$ then $m\ (x \triangledown f\ x) < m\ x$ by (13.4). This is on $'av$. Therefore we need to lift (13.4) to annotated commands. Think in terms of tuples $(a_1,a_2,\ldots)$ of abstract values and let the termination measure m' be the sum of all $m\ a_i$, just as in Figure 13.8 and the surrounding explanation. Then $\neg\ (b_1,b_2,\ldots) \leqslant (a_1,a_2,\ldots)$ means that there is a k such that $\neg\ b_k \leqslant a_k$. We have $a_i \leqslant a_i \triangledown b_i$ for all i because

$\triangledown$ is a widening operator. Therefore $m(a_i \triangledown b_i) \leqslant m\ a_i$ for all i (by anti-monotonicity) and $m(a_k \triangledown b_k) < m\ a_k$ (by (13.4)). Thus $m'((a_1,a_2,\ldots) \triangledown (b_1,b_2,\ldots)) = m'(a_1 \triangledown b_1, a_2 \triangledown b_2,\ldots) < m'(a_1,a_2,\ldots)$. That is, (13.4) holds on tuples too. Now for the technical details.

Recall from Section 13.5.8 that we lifted m to m_s, m_o and m_c. Following the sketch above, (13.4) can be proved for these derived functions too, with a few side-conditions:

$$\llbracket \mathit{finite}\ X;\ \mathit{fun}\ S_1 = \mathit{fun}\ S_2\ \mathit{on} - X;\ \neg\ S_2 \leqslant S_1 \rrbracket$$
$$\Longrightarrow m_s\ (S_1 \triangledown S_2)\ X < m_s\ S_1\ X$$

$$\llbracket \mathit{finite}\ X;\ \mathit{top\_on}_o\ S_1\ (-X);\ \mathit{top\_on}_o\ S_2\ (-X);\ \neg\ S_2 \leqslant S_1 \rrbracket$$
$$\Longrightarrow m_o\ (S_1 \triangledown S_2)\ X < m_o\ S_1\ X$$

$$\llbracket \mathit{strip}\ C_1 = \mathit{strip}\ C_2;\ \mathit{top\_on}_c\ C_1\ (-\mathit{vars}\ C_1);$$
$$\mathit{top\_on}_c\ C_2\ (-\mathit{vars}\ C_2);\ \neg\ C_2 \leqslant C_1 \rrbracket$$
$$\Longrightarrow m_c\ (C_1 \triangledown C_2) < m_c\ C_1$$

The last of the above three lemmas implies termination of *iter_widen f C* for $C :: {'av}\ \mathit{st}\ \mathit{option}\ \mathit{acom}$ as follows. If you set $C_2 = f\ C_1$ and assume that f preserves the *strip* and $\mathit{top\_on}_c$ properties then each step of *iter_widen f C* strictly decreases m_c, which must terminate because m_c returns a *nat*.

To apply this result to widening on intervals we merely need to provide the right m and h:

$$m\_ivl\ [l,\ h] =$$
$$(\mathit{if}\ [l,\ h] = \bot\ \mathit{then}\ 3$$
$$\mathit{else}\ (\mathit{if}\ l = -\infty\ \mathit{then}\ 0\ \mathit{else}\ 1) + (\mathit{if}\ h = \infty\ \mathit{then}\ 0\ \mathit{else}\ 1))$$

Strictly speaking m does not need to consider the $\bot$ case because we have made sure it cannot arise in an abstract state, but it is simpler not to rely on this and cover $\bot$.

Function *m_ivl* satisfies the required properties: $m\_ivl\ iv \leqslant 3$ and $y \leqslant x \Longrightarrow m\_ivl\ x \leqslant m\_ivl\ y$.

Because *bot* suitably initialises and *step_ivl* preserves the *strip* and $\mathit{top\_on}_c$ properties, this implies termination of *iter_widen* (*step_ivl* $\top$):

$$\exists\, C.\ \mathit{iter\_widen}\ (\mathit{step\_ivl}\ \top)\ (\mathit{bot}\ c) = \mathit{Some}\ C$$

13.9.5 Termination of Narrowing

The proof is similar to that for widening. We require another measure function

$$n :: {'av} \Rightarrow \mathit{nat} \quad \text{such that}$$
$$\llbracket a_2 \leqslant a_1;\ a_1 \vartriangle a_2 < a_1 \rrbracket \Longrightarrow n\ (a_1 \vartriangle a_2) < n\ a_1 \tag{13.5}$$

This property guarantees that the measure goes down with every iteration of *iter_narrow* $f\ a_0$, provided f is monotone and $f\ a_0 \leqslant a_0$: let $a_2 = f\ a_1$; then $a_2 \leqslant a_1$ is the pre-fixpoint property that iteration with narrowing preserves (see Section 13.9.2) and $a_1 \vartriangle a_2 < a_1$ is the loop condition.

Now we need to lift this from $'av$ to other types. First we sketch how it works for tuples. Define the termination measure $n'(a_1,a_2,\ldots) = n\ a_1 + n\ a_2 + \ldots$. To show that (13.5) holds for n' too, assume $(b_1,b_2,\ldots) \leqslant (a_1,a_2,\ldots)$ and $(a_1,a_2,\ldots) \vartriangle (b_1,b_2,\ldots) < (a_1,a_2,\ldots)$, i.e. $b_i \leqslant a_i$ for all i, $a_i \vartriangle b_i \leqslant a_i$ for all i and $a_k \vartriangle b_k \leqslant a_k$ for some k. Thus for all i either $a_i \vartriangle b_i = a_i$ (and hence $n(a_i \vartriangle b_i) = n\ a_i$) or $a_i \vartriangle b_i < a_i$, which together with $b_i \leqslant a_i$ yields $n(a_i \vartriangle b_i) < n\ a_i$ by (13.5). Thus $n(a_i \vartriangle b_i) \leqslant n\ a_i$ for all i. But $n(a_k \vartriangle b_k) < n\ a_k$, also by (13.5), and hence $n'((a_1,a_2,\ldots) \vartriangle (b_1,b_2,\ldots)) = n'(a_1 \vartriangle b_1, a_2 \vartriangle b_2,\ldots) < n'(a_1,a_2,\ldots)$. Thus (13.5) holds for n' too. Now for the technical details.

We lift n in three stages to $'av\ st\ option\ acom$:

definition $n_s :: 'av\ st \Rightarrow vname\ set \Rightarrow nat$ **where**
$n_s\ S\ X = (\sum x{\in}X.\ n\ (fun\ S\ x))$

fun $n_o :: 'av\ st\ option \Rightarrow vname\ set \Rightarrow nat$ **where**
$n_o\ None\ X = 0$
$n_o\ (Some\ S)\ X = n_s\ S\ X + 1$

definition $n_c :: 'av\ st\ option\ acom \Rightarrow nat$ **where**
$n_c\ C = (\sum a{\leftarrow}annos\ C.\ n_o\ a\ (vars\ C))$

Property (13.5) carries over to $'av\ st\ option\ acom$ with suitable side conditions just like (13.4) carried over in the previous subsection:

$[\![strip\ C_1 = strip\ C_2;\ top\_on_c\ C_1\ (-\ vars\ C_1);$
$top\_on_c\ C_2\ (-\ vars\ C_2);\ C_2 \leqslant C_1;\ C_1 \vartriangle C_2 < C_1]\!]$
$\Longrightarrow n_c\ (C_1 \vartriangle C_2) < n_c\ C_1$

This implies termination of *iter_narrow* $f\ C$ for $C :: 'av\ st\ option\ acom$ provided f is monotone and C is a pre-fixpoint of f (which is preserved by narrowing because f is monotone and $\vartriangle$ a narrowing operator) because it guarantees that each step of *iter_narrow* $f\ C$ strictly decreases n_c.

To apply this result to narrowing on intervals we merely need to provide the right n:

$n\_ivl\ iv = 3 - m\_ivl\ iv$

It does what it is supposed to do, namely satisfy (a strengthened version of) (13.5): $x \vartriangle y < x \Longrightarrow n\_ivl\ (x \vartriangle y) < n\_ivl\ x$. Therefore narrowing terminates provided we start with a pre-fixpoint which is $\top$ outside its variables:

$$[\![top\_on_c\ C\ (-\ vars\ C);\ step\_ivl\ \top\ C \leqslant C]\!]$$
$$\Longrightarrow \exists\, C'.\ iter\_narrow\ (step\_ivl\ \top)\ C = Some\ C'$$

Because both preconditions are guaranteed by the output of widening we obtain the final termination theorem where *AI_wn_ivl* is *AI_wn* for intervals:

Theorem 13.57. $\exists\, C.\ AI\_wn\_ivl\ c = Some\ C$

13.9.6 Exercises

Exercise 13.58. Starting from

```
''x'' ::= N 0 {Some {(''x'', [0, 0])}};;
{Some {(''x'', [0, 0])}}
WHILE Less (V ''x'') (N 100)
DO {Some {(''x'', [0, 0])}}
    ''x'' ::= Plus (V ''x'') (N 1) {Some {(''x'', [1, 1])}}
{None}
```

tabulate the three steps that *step_ivl* with widening takes to reach (13.3). Follow the format of Table 13.1.

Exercise 13.59. Starting from (13.3), tabulate both the repeated application of *step_ivl* $\top$ alone and of *iter_narrow* (*step_ivl* $\top$), the latter following the format of Table 13.1 (with $\triangledown$ replaced by $\triangle$).

13.10 Summary and Further Reading

This concludes the chapter on abstract interpretation. The main points were:

- The reference point is a collecting semantics that describe for each program point which sets of states can arise at that point.
- The abstract interpreter mimics the collecting semantics but operates on abstract values representing (infinite) sets of concrete values. Abstract values should form a lattice: $\sqcup$ abstracts $\cup$ and models the effect of joining two computation paths, $\sqcap$ can reduce abstract values to model backward analysis of boolean expressions.
- The collecting semantics is defined as a least fixpoint of a step function. The abstract interpreter approximates this by iterating its step function. For monotone step functions this iteration terminates if the ordering is of finite height (or at least there is no infinitely ascending chain).
- Widening is an approach to accelerating the iteration process to guarantee termination even if there are infinitely ascending chains in the ordering, for example on intervals. It trades precision for termination. Narrowing

is an approach for improving the precision lost by widening by further iterations.

Of the analyses formalised in Chapter 10 we have only rephrased constant propagation as abstract interpretation. Definite initialisation can be covered if the collecting semantics is extended to distinguish initialised from uninitialised variables. Our simple framework cannot accommodate live variable analysis for two reasons. Firstly, it is a backward analysis, wheras our abstract interpreter is a forward analyser. Secondly, it requires a different semantics: liveness of a variable at some point is not a property of the possible values of that variable but of the sequences of operations executed after that point.

13.10.1 Further Reading

Abstract Interpretation was invented by Patrick and Radhia Cousot [23]. In its standard form, the concrete and abstract level are related by a concretisation function γ together with an adjoint abstraction function α. This allows the verification not just of correctness but also of optimality of an abstract interpreter. In fact, one can even "calculate" the abstract interpreter [22]. The book by Nielson, Nielson and Hankin [63] also has a chapter on abstract interpretation.

There are many variations, generalisations and applications of abstract interpretation. Particularly important are relational analyses which can deal with relations between program variables: they do not abstract $(vname \Rightarrow val)\ set$ to $vname \Rightarrow val\ set$ as we have done. The canonical example is polyhedral analysis that generalises interval analysis to linear inequalities between variables [24].

Our intentionally simple approach to abstract interpretation can be improved in a number of respects:

- We combine the program and the annotations into one data structure and performing all computations on it. Normally the program points are labelled and the abstract interpreter operates on a separate mapping from labels to annotations.
- We iterate globally over the whole program. Precision is improved by a compositional iteration strategy: the global fixpoint iteration is replaced by a fixpoint iteration for each loop.
- We perform widening and narrowing for all program points. Precision can be improved if we restrict to one program point per loop.

Cachera and Pichardie [18] have formalised an abstract interpreter in Coq that follows all three improvements. Iteration strategies and widening points have been investigated by Bourdoncle [15] for arbitrary control-flow graphs.

Our abstract interpreter operates on a single data structure, an annotated command. A more modular design transforms the pre-fixpoint requirement $step'\ \top\ C \leqslant C$ into a set of inequalities over the abstract domain and solves those inequalities by some arbitrary method. This separates the programming language aspect from the mathematics of solving particular types of constraints. For example, Gawlitza and Seidl [34] showed how to compute least solutions of interval constraints precisely, without any widening and narrowing.

The Kaster-Tarski fixpoint theorem is due to Tarski [86] and is more general than the one we proved. It says that the set of fixpoints of a monotone function on a complete lattice is itself a complete lattice. A simpler version for sets had already been obtained by Knaster and Tarski earlier [50].

A

Auxiliary Definitions

This appendix contains auxiliary definitions omitted from the main text.

Variables

fun *lvars* :: *com* $\Rightarrow$ *vname set* **where**
lvars SKIP $= \{\}$
lvars $(x ::= e)$ $= \{x\}$
lvars $(c_1;; c_2)$ $=$ *lvars* $c_1 \cup$ *lvars* c_2
lvars $(IF\ b\ THEN\ c_1\ ELSE\ c_2)$ $=$ *lvars* $c_1 \cup$ *lvars* c_2
lvars $(WHILE\ b\ DO\ c)$ $=$ *lvars* c

fun *rvars* :: *com* $\Rightarrow$ *vname set* **where**
rvars SKIP $= \{\}$
rvars $(x ::= e)$ $=$ *vars* e
rvars $(c_1;; c_2)$ $=$ *rvars* $c_1 \cup$ *rvars* c_2
rvars $(IF\ b\ THEN\ c_1\ ELSE\ c_2)$ $=$ *vars* $b \cup$ *rvars* $c_1 \cup$ *rvars* c_2
rvars $(WHILE\ b\ DO\ c)$ $=$ *vars* $b \cup$ *rvars* c

definition *vars* :: *com* $\Rightarrow$ *vname set* **where**
vars $c =$ *lvars* $c \cup$ *rvars* c

Abstract Interpretation

fun *strip* :: $'a$ *acom* $\Rightarrow$ *com* **where**
strip $(SKIP\ \{P\}) =$ *com.SKIP*
strip $(x ::= e\ \{P\}) = x ::= e$

```
strip (C1;;C2) = strip C1;; strip C2
strip (IF b THEN {P1} C1 ELSE {P2} C2 {P}) =
  IF b THEN strip C1 ELSE strip C2
strip ({I} WHILE b DO {P} C {Q}) = WHILE b DO strip C
```

```
fun annos :: 'a acom ⇒ 'a list where
annos (SKIP {P}) = [P]
annos (x ::= e {P}) = [P]
annos (C1;;C2) = annos C1 @ annos C2
annos (IF b THEN {P1} C1 ELSE {P2} C2 {Q}) =
  P1 # annos C1 @ P2 # annos C2 @ [Q]
annos ({I} WHILE b DO {P} C {Q}) = I # P # annos C @ [Q]
```

```
fun asize :: com ⇒ nat where
asize com.SKIP = 1
asize (x ::= e) = 1
asize (C1;;C2) = asize C1 + asize C2
asize (IF b THEN C1 ELSE C2) = asize C1 + asize C2 + 3
asize (WHILE b DO C) = asize C + 3
```

```
definition shift :: (nat ⇒ 'a) ⇒ nat ⇒ nat ⇒ 'a where
shift f n = (λp. f(p+n))
```

```
fun annotate :: (nat ⇒ 'a) ⇒ com ⇒ 'a acom where
annotate f com.SKIP = SKIP {f 0}
annotate f (x ::= e) = x ::= e {f 0}
annotate f (c1;;c2) = annotate f c1;; annotate (shift f (asize c1)) c2
annotate f (IF b THEN c1 ELSE c2) =
  IF b THEN {f 0} annotate (shift f 1) c1
  ELSE {f(asize c1 + 1)} annotate (shift f (asize c1 + 2)) c2
  {f(asize c1 + asize c2 + 2)}
annotate f (WHILE b DO c) =
  {f 0} WHILE b DO {f 1} annotate (shift f 2) c {f(asize c + 2)}
```

```
fun map_acom :: ('a ⇒ 'b) ⇒ 'a acom ⇒ 'b acom where
map_acom f (SKIP {P}) = SKIP {f P}
map_acom f (x ::= e {P}) = x ::= e {f P}
map_acom f (C1;;C2) = map_acom f C1;; map_acom f C2
map_acom f (IF b THEN {P1} C1 ELSE {P2} C2 {Q}) =
  IF b THEN {f P1} map_acom f C1 ELSE {f P2} map_acom f C2
  {f Q}
map_acom f ({I} WHILE b DO {P} C {Q}) =
  {f I} WHILE b DO {f P} map_acom f C {f Q}
```

B

Symbols

$[\![$	`[|`	`\<lbrakk>`
$]\!]$	`|]`	`\<rbrakk>`
$\Longrightarrow$	`==>`	`\<Longrightarrow>`
$\bigwedge$	`!!`	`\<And>`
$\equiv$	`==`	`\<equiv>`
λ	`%`	`\<lambda>`
$\Rightarrow$	`=>`	`\<Rightarrow>`
$\wedge$	`&`	`\<and>`
$\vee$	`|`	`\<or>`
$\longrightarrow$	`-->`	`\<longrightarrow>`
$\neg$	`~`	`\<not>`
$\neq$	`~=`	`\<noteq>`
$\forall$	`ALL`	`\<forall>`
$\exists$	`EX`	`\<exists>`
$\circ$	`o`	`\<circ>`
$\leqslant$	`<=`	`\<le>`
$\times$	`*`	`\<times>`
$\in$	`:`	`\<in>`
$\notin$	`~:`	`\<notin>`
$\subseteq$	`<=`	`\<subseteq>`
$\subset$	`<`	`\<subset>`
$\cup$	`Un`	`\<union>`
$\cap$	`Int`	`\<inter>`
$\bigcup$	`UN`, `Union`	`\<Union>`
$\bigcap$	`INT`, `Inter`	`\<Inter>`

Table B.1. Mathematical symbols, their ASCII-equivalents and internal names

References

1. Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford University Press, 1994.
2. Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
3. Eyad Alkassar, Mark Hillebrand, Dirk Leinenbach, Norbert Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load — leveraging a semantics stack for systems verification. *Journal of Automated Reasoning: Special Issue on Operating System Verification*, 42, Numbers 2–4:389–454, 2009.
4. Eyad Alkassar, Mark Hillebrand, Wolfgang Paul, and Elena Petrova. Automated verification of a small hypervisor. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Proceedings of Verified Software: Theories, Tools and Experiments 2010*, volume 6217 of *Lect. Notes in Comp. Sci.*, pages 40–54. Springer-Verlag, 2010.
5. Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84:129–162, 1990.
6. Krzysztof Apt. Ten Years of Hoare's Logic: A Survey — Part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.
7. Krzysztof Apt. Ten Years of Hoare's Logic: A Survey — Part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.
8. Krzysztof Apt, Frank de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 3rd edition, 2009.
9. Clemens Ballarin. *Tutorial on Locales and Locale Interpretation*. `http://isabelle.in.tum.de/doc/locales.pdf`.
10. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lect. Notes in Comp. Sci.*, pages 364–387. Springer-Verlag, 2005.
11. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

12. Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in Coq. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lect. Notes in Comp. Sci.*, pages 115–130. Springer-Verlag, 2009.
13. William R. Bevier, Warren A. Hunt Jr., J. Strother Moore, and William D. Young. An approach to systems verification. *J. Autom. Reasoning*, 5(4):411–428, 1989.
14. Richard Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lect. Notes in Comp. Sci.*, pages 102–126. Springer-Verlag, 2000.
15. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In D. Bjørner, Manfred M. Broy, and I. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lect. Notes in Comp. Sci.*, pages 128–141. Springer-Verlag, 1993.
16. David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
17. Rod Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.
18. David Cachera and David Pichardie. A certified denotational abstract interpreter. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving (ITP 2010)*, volume 6172 of *Lect. Notes in Comp. Sci.*, pages 9–24. Springer-Verlag, 2010.
19. 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.
20. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lect. Notes in Comp. Sci.*, pages 23–42, Munich, Germany, 2009. Springer-Verlag.
21. Stephen Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. on Computing*, 7:70–90, 1978.
22. Patrick Cousot. The calculational design of a generic abstract interpreter. In Broy and Steinbrüggen, editors, *Calculational System Design*. IOS Press, 1999.
23. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.
24. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symp. Principles of Programming Languages*, pages 84–97, 1978.
25. Marc Daumas, Laurence Rideau, and Laurent Théry. A generic library for floating-point numbers and its application to exact computing. In R. Boulton

and P. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lect. Notes in Comp. Sci.*, pages 169–184. Springer-Verlag, 2001.
26. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *Lect. Notes in Comp. Sci.*, pages 337–340, Budapest, Hungary, March 2008. Springer-Verlag.
27. Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
28. Edsger W. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.
29. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
30. Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jörg Pfähler, and Wolfgang Reif. A formal model of a virtual filesystem switch. In *Proc. 7th SSV*, pages 33–45, 2012.
31. Robert Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32. American Mathematical Society, 1967.
32. Anthony Fox. Formal specification and verification of ARM6. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th Int. Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *Lect. Notes in Comp. Sci.*, pages 25–40, Rome, Italy, September 2003. Springer-Verlag.
33. Anthony Fox and Magnus Myreen. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In Matt Kaufmann and Lawrence C. Paulson, editors, *1st Int. Conference on Interactive Theorem Proving (ITP)*, volume 6172 of *Lect. Notes in Comp. Sci.*, pages 243–258, Edinburgh, UK, July 2010. Springer-Verlag.
34. Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In Rocco De Nicola, editor, *Programming Languages and Systems, ESOP 2007*, volume 4421 of *Lect. Notes in Comp. Sci.*, pages 300–315. Springer-Verlag, 2007.
35. Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.
36. Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
37. M.C.J. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.
38. Mike Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.
39. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, 3rd edition*. Addison-Wesley, 2005.
40. Carl Gunter. *Semantics of programming languages: structures and techniques*. MIT Press, 1992.
41. Florian Haftmann. *Haskell-style type classes with Isabelle/Isar*. http://isabelle.in.tum.de/doc/classes.pdf.
42. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:567–580,583, 1969.

43. John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd edition, 2006.
44. Brian Huffman. A purely definitional universal domain. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lect. Notes in Comp. Sci.*, pages 260–275. Springer-Verlag, 2009.
45. Michael Huth and Mark Ryan. *Logic in Computer Science.* Cambridge University Press, 2004.
46. Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 132–146. ACM, 1999.
47. Gilles Kahn. Natural semantics. In *STACS 87: Symp. Theoretical Aspects of Computer Science*, volume 247 of *Lect. Notes in Comp. Sci.*, pages 22–39. Springer-Verlag, 1987.
48. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proc. 22nd ACM Symposium on Operating Systems Principles 2009*, pages 207–220. ACM, 2009.
49. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.
50. Bronisław Knaster. Un théorème sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6:133–134, 1928.
51. Alexander Krauss. *Defining Recursive Functions in Isabelle/HOL.* http://isabelle.in.tum.de/doc/functions.pdf.
52. Alexander Krauss. Recursive definitions of monadic functions. In A. Bove, E. Komendantskaya, and M. Niqui, editors, *Proc. Workshop on Partiality and Recursion in Interactive Theorem Provers*, volume 43 of *EPTCS*, pages 1–13, 2010.
53. Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.
54. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lect. Notes in Comp. Sci.*, pages 348–370. Springer-Verlag, 2010.
55. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proc. 33rd ACM Symposium on Principles of Programming Languages*, pages 42–54. ACM, 2006.
56. Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition.* Addison-Wesley, February 2013.
57. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.
58. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences (JCCS)*, 17(3):348–375, 1978.

59. Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404, New York, NY, USA, 2012. ACM.
60. Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
61. Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *J. Functional Programming*, 9:191–223, 1999.
62. 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.
63. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
64. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications. An Appatizer*. Springer-Verlag, 2007.
65. Tobias Nipkow. *What's in Main*. http://isabelle.in.tum.de/doc/main.pdf.
66. Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.
67. Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *Lect. Notes in Comp. Sci.*, pages 103–119. Springer-Verlag, 2002.
68. Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.
69. Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2002.
70. Tobias Nipkow and Leonor Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lect. Notes in Comp. Sci.*, pages 188–203. Springer-Verlag, 1999.
71. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
72. Gordon D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.
73. Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.
74. Wolfgang Reif. The KIV system: Systematic construction of verified software. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lect. Notes in Comp. Sci.*, pages 753–757. Springer-Verlag, June 1992.
75. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.

76. 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.
77. Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
78. 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.
79. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
80. David Schmidt. *Denotational semantics: A methodology for language development*. Allyn and Bacon, 1986.
81. Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 697–711. Springer-Verlag, 1997.
82. 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.
83. Dana Scott. Outline of a mathematical theory of computation. In *Information Sciences and Systems: Proc. 4th Annual Princeton Conference*, pages 169–176. Princeton University Press, 1970.
84. Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford University Computing Lab., 1971.
85. Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Seattle, Washington, USA, June 2013. ACM.
86. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.
87. Robert Tennent. Denotational semantics. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 169–322. Oxford University Press, 1994.
88. Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School of Computer Science and Engineering, University of NSW, Sydney, Australia, August 2008.
89. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, January 2007. ACM.
90. 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.
91. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th IEEE workshop on Computer Security Foundations*, CSFW '97, pages 156–169. IEEE Computer Society, 1997.

92. Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *Proc. 7th Int. Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '97)*, volume 1214 of *Lect. Notes in Comp. Sci.*, pages 607–621. Springer-Verlag, 1997.
93. Makarius Wenzel. *The Isabelle/Isar Reference Manual*. http://isabelle.in.tum.de/doc/isar-ref.pdf.
94. Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.