

# An Introduction to Isabelle for Electrical Engineers

John Wickerson

November 12, 2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Getting started</b>	<b>4</b>
<b>3</b>	<b>Proofs by hand, and proofs by machine</b>	<b>5</b>
3.1	A first proof . . . . .	5
3.2	Discussion: The meta logic and the object logic . . . . .	9
3.3	A second proof . . . . .	10
3.4	Two styles of Isabelle proof . . . . .	12
<b>4</b>	<b>Inductive proofs</b>	<b>15</b>
4.1	A proof about triangle numbers . . . . .	16
4.2	A proof about tetrahedral numbers . . . . .	18
<b>5</b>	<b>A verified logic synthesiser</b>	<b>24</b>
5.1	Representing circuits . . . . .	24
5.2	Simulating circuits . . . . .	25
5.3	Structural induction on circuits . . . . .	26
5.4	A simple circuit optimiser . . . . .	28
5.5	Rule induction . . . . .	30
5.6	Proving termination . . . . .	31
5.7	Verifying our optimiser . . . . .	33

# Chapter 1

## Introduction

This document provides an introduction to automated theorem-proving using a tool called Isabelle. It is aimed at an audience of electrical engineers. Some prior programming experience is assumed. Competence with *functional* programming (e.g. in Haskell or OCaml) is a bonus. No particularly complicated mathematics is needed, but an ability to create structured, logical arguments will be helpful.

Isabelle is one of several tools available for automating mathematical proofs. Others include Coq, HOL, and PVS. Isabelle was invented in the late 1980s by Larry Paulson from the University of Cambridge, and has been under development ever since. It is freely available to download.

<https://isabelle.in.tum.de>

There is a growing interest in using tools like Isabelle. In short, computer systems are becoming increasingly complicated and increasingly relied upon. It is therefore becoming ever more *difficult* and ever more *important* to guarantee that they behave correctly. One way to establish the correctness of a computer system is to program it in a tool like Isabelle, and to prove its correctness as a mathematical theorem. Already, we have entire compilers (such as CompCert [3] and CakeML [2]), entire operating systems (such as seL4 [1]), and entire processors (such as Silver [4]) implemented and verified correct inside theorem-proving tools. And as these theorem-proving tools become ever more powerful, we are likely to see them applied to many other systems in the future.

The aim of this course, then, is simply to introduce electrical engineers to this important topic.

After getting you started (Chapter 2), these notes present a few worked examples of how some standard proofs can be recreated in Isabelle (Chapters 3 and 4). Then we show how Isabelle can be used to program a simple logic synthesiser, and ultimately to prove that the synthesiser is correct (Chapter 5).

**Acknowledgements** I'd like to thank Yann Herklotz, Matt Windsor, and Quentin Corradi for their invaluable help in developing this material.

## Chapter 2

### Getting started

Isabelle can be downloaded from <https://isabelle.in.tum.de>. The development environment is based on jEdit, and takes a little bit of getting used to. At some point there was an Eclipse plug-in for Isabelle but I'm not sure it's still maintained.

Some Linux systems may require the following environment variable to be set in order for jEdit to work properly:

```
export _JAVA_AWT_WM_NONREParenting=1
```

## Chapter 3

# Proofs by hand, and proofs by machine

In this chapter, we look at the structure of a few standard mathematical proofs, and see how they can be recreated in Isabelle.

### 3.1 A first proof

Figure 3.1 gives a standard mathematical theorem, and the kind of proof that might accompany it in a textbook. Figure 3.2 presents the same theorem and the same proof again, this time programmed into Isabelle. (I have taken a few typographical liberties to improve the presentation. The original proof script is available separately.) Let us step through the proof line-by-line.

- 1 When we state a **theorem**, Isabelle expects the statement to be immediately followed by a proof. If you're not ready to give a proof right now, you can type **sorry** instead. This will let you work on other theorems later on in your proof file, and come back to proving this theorem later. Instead of **theorem** we could have typed **lemma**. This means the same from Isabelle's point of view, but generally indicates a less important theorem.
- 2 A proof begins with **proof** and ends with **qed**. If the proof is a one-liner, you can just use **by** instead. The argument to the **proof** command tells Isabelle how to begin the proof. If we'd rather take full control, we can write **proof** – to tell Isabelle not to do anything yet. Here we are using `auto`, which instructs Isabelle to use a variety of automatic techniques to try to simplify our proof goals (and maybe even to finish the proof straight away if we're lucky). In this case, `auto` has changed our goal from

**Theorem 1.**  $\sqrt{2}$  is irrational.

*Proof.* To show that  $\sqrt{2}$  is *not* rational, we shall assume that it *is* rational, and deduce a contradiction. To this end, suppose  $\sqrt{2}$  is rational. Then we can obtain integers  $m$  and  $n$  such that  $\sqrt{2} = m/n$ , where  $n > 0$  and  $\gcd(m, n) = 1$ . Squaring both sides gives us  $2 = m^2/n^2$ , and hence  $2n^2 = m^2$ . This means that  $m^2$  is even, and hence so is  $m$ . Hence, we can obtain an integer  $m'$  such that  $m = 2m'$ . It follows that  $2n^2 = (2m')^2$ , and hence  $n^2 = 2m'$ . This means that  $n^2$  is even, and hence so is  $n$ . But if both  $m$  and  $n$  are even, their gcd is at least 2, which contradicts the fact that  $\gcd(m, n) = 1$ . Thus our original supposition must have been wrong, which means that  $\sqrt{2}$  is not rational.  $\square$

Figure 3.1: A proof by hand that  $\sqrt{2}$  is irrational.

```

1 theorem  $\sqrt{2} \notin \mathbb{Q}$ 
2 proof auto
3   assume  $\sqrt{2} \in \mathbb{Q}$ 
4   then obtain  $m\ n$  where
5      $n \neq 0$  and  $|\sqrt{2}| = \text{real } m / \text{real } n$  and  $\text{coprime } m\ n$ 
6     by (rule Rats_abs_nat_div_natE)
7   hence  $|\sqrt{2}|^2 = (\text{real } m / \text{real } n)^2$  by auto
8   hence  $2 = (\text{real } m / \text{real } n)^2$  by simp
9   hence  $2 = (\text{real } m)^2 / (\text{real } n)^2$  unfolding power_divide by auto
10  hence  $2 \times (\text{real } n)^2 = (\text{real } m)^2$ 
11    by (simp add: nonzero_eq_divide_eq ' $n \neq 0$ ')
12  hence  $\text{real}(2 \times n^2) = (\text{real } m)^2$  by auto
13  hence *:  $2 \times n^2 = m^2$ 
14    using of_nat_power_eq_of_nat_cancel_iff by blast
15  hence  $\text{even}(m^2)$  by presburger
16  hence  $\text{even } m$  by simp
17  then obtain  $m'$  where  $m = 2 \times m'$  by auto
18  with * have  $2 \times n^2 = (2 \times m')^2$  by auto
19  hence  $2 \times n^2 = 4 \times m'^2$  by simp
20  hence  $n^2 = 2 \times m'^2$  by simp
21  hence  $\text{even}(n^2)$  by presburger
22  hence  $\text{even } n$  by simp
23  with ' $\text{even } m$ ' and ' $\text{coprime } m\ n$ ' show False by auto
24 qed

```

Figure 3.2: A proof in Isabelle that  $\sqrt{2}$  is irrational.

Assume:	(nothing)
Show:	$\sqrt{2} \notin \mathbb{Q}$

to

Assume:	$\sqrt{2} \in \mathbb{Q}$
Show:	<i>False</i>

also known as *proof of negation*.

3 Our goal allows us to assume  $\sqrt{2} \in \mathbb{Q}$ , so we issue the **assume** command to make this fact available for future proof steps. We could make multiple assumptions in one line by separating them with **and**.

4–6 We want to exploit the fact that if a number is in  $\mathbb{Q}$  then it can be expressed as the ratio between two integers. A search of Isabelle’s library for theorems that include the pattern “ $_ \in \mathbb{Q}$ ” yields the following theorem:

Theorem name:	Rats_abs_nat_div_natE
Assumptions:	$x \in \mathbb{Q}$
Conclusions:	$n \neq 0$ $ x  = \text{real } m / \text{real } n$ $\text{coprime } m \ n$

By the way, we shall use the convention that variables appearing in assumptions are *universally quantified* (which means that the theorem is true for *any* values of those variables), while variables that only appear in conclusions are *existentially quantified* (which means that the theorem is true for *some* values of those variables). That means that the theorem above can be read as “For any  $x$ , if  $x$  is a rational number, then there exist integers  $n$  and  $m$  such that  $n \neq 0$ , the absolute value of  $x$  is  $m/n$ , and  $m$  and  $n$  are coprime.” We need the **real** function here to cast from integers to reals – it’s a bit like writing `(float)m` in C++.

We write out the conclusions of the theorem, with  $x$  instantiated to  $\sqrt{2}$ , using **obtain** for the existentially-quantified variables  $m$  and  $n$ , and **and** to separate the three conclusions of the theorem. At the end of line 5, our goal is

Assume:	$\sqrt{2} \in \mathbb{Q}$
Show:	$n \neq 0$ $ \sqrt{2}  = \text{real } m / \text{real } n$ $\text{coprime } m \ n$

This is a perfect match with `Rats_abs_nat_div_natE`, so the **rule** method will complete this goal easily.



- 7 We now wish to square both sides of the equation. We use the **hence** command, which allows us to state a new fact that we wish to prove, using the facts from the previous line as assumptions. Our goal becomes

Assume:	$n \neq 0$ $ \sqrt{2}  = \text{real } m / \text{real } n$ $\text{coprime } m \ n$
Show:	$ \sqrt{2} ^2 = (\text{real } m / \text{real } n)^2$

which is easily dealt with by `auto`. (In fact, only the second of those three assumptions is needed here, but the other two assumptions don't hurt.) In general it is quite a good idea to issue **'by auto'** after each fact you state, to see whether Isabelle can prove it for you without further ado. There are other methods available, such as `simp`, `clarify`, `clarsimp`, `blast`, and `presburger`. These methods tend to run faster than `auto`, so if you use them where possible then your proof script may run more quickly than one that is full of `autos`. If you're not sure which of these automated methods will work, you can simply type **try**. This invokes Isabelle's *sledgehammer*, which tries all of these automated methods simultaneously, and reports the first one that succeeds, for you to paste into your proof script.

- 8 We then wish to replace  $|\sqrt{2}|$  with `2`. This is a straightforward task for Isabelle's automatic simplifier (`simp`). You might find that the proof can be optimised by jumping straight to this fact and omitting line 7. Or you might prefer to include these intermediate steps so that the human reading the proof can see what's going on in more detail – it's up to you.
- 9 We now want to rewrite the right-hand side of our equation by pushing the squaring operation 'inside' the division operation. A search of the Isabelle library for theorems that include the pattern "`(_/_)^`" yields one called `power_divide`, which tells us  $(a/b)^n = a^n/b^n$ . Since this fact is expressed as an equality, we have an opportunity to demonstrate the **unfolding** command, which simply rewrites our goal by replacing any instances of the  $(a/b)^n$  pattern with a corresponding instance of the  $a^n/b^n$  pattern.
- 10–11 Next, we wish to move the  $n$  over to the other side of the equation. We find the `nonzero_eq_divide_eq` theorem in the Isabelle library, which tells that we can do this providing  $n$  is not zero. We can complete this step of our proof using Isabelle's simplification engine, once we provide it with two additional facts using the `add` parameter: the `nonzero_eq_divide_eq` theorem itself, and the fact that  $n$  is indeed not zero. Note the use of backticks to refer to a fact 'by contents'. We could alternatively have given the  $n \neq 0$

fact a label when we introduced it on line 5, and then referred to it by that label, at the small cost of complicating the proof with extra labels.

- 12 Next we bring the `real` function outside the multiplication. This step is straightforward for `auto`.
- 13–14 At this point we can drop the `real` function from both sides of our equation. The justification for this step uses a fairly arcane theorem, which was discovered using the `try` command. We give this fact the label `*` so that we can refer back to it later. You can also use numbers and letters to label facts.
- 15 We next deduce that  $m^2$  is even.
- 16 It follows that  $m$  is also even. Isabelle finds this step very straightforward – `simp` is quite sufficient. As a human, I find this step a little less obvious than some of the steps that Isabelle required quite a bit of guidance with.
- 17 We can introduce the new variable  $m'$  using the `obtain` keyword.
- 18 The `with * have` phrase means “using the facts from the previous line together with the fact with label ‘`*`’ we can prove the following fact”.
- 19–22 A straightforward sequence of steps leads us to deduce that  $n$  is even.
- 23 The combination of  $n$  being even,  $m$  being even, and  $m$  and  $n$  being coprime leads to a contradiction. We use the keyword `show` here rather than `have`, to show that we are meeting the goal we set ourselves immediately after the `proof auto` command.
- 24 The proof is now complete; *quod erat demonstrandum!*

## 3.2 Discussion: The meta logic and the object logic

Isabelle allows us not only to write down logical statements, but to write down judgements *about* those logical statements. This is a subtle distinction, best illustrated by example. Consider the following four statements.

1. For every  $x$ , if it is the case that  $\text{even}(x)$  holds and it is the case that  $\text{odd}(x)$  holds then it is the case that  $x = 0$  holds.
2. For every  $x$ , if it is the case that  $\text{even}(x) \wedge \text{odd}(x)$  holds then it is the case that  $x = 0$  holds.
3. For every  $x$ , it is the case that  $(\text{even}(x) \wedge \text{odd}(x)) \longrightarrow x = 0$  holds.

4. It is the case that  $\forall x. (\text{even}(x) \wedge \text{odd}(x)) \longrightarrow x = 0$  holds.

All four statements are expressing the same sentiment – a sentiment that, incidentally, is correct. In each case we are using the *meta logic* to talk about phrases in the *object logic*. The meta logic includes phrases like “it is the case that \_ holds” and “for every \_” and “if \_ then \_”. The object logic includes phrases like “\_  $\wedge$  \_” and “\_  $\longrightarrow$  \_” and “ $\forall$  \_”. In the first statement, we are using the meta logic to combine three small phrases in the object logic; in the fourth statement, we do all the combining within the object logic.

Isabelle is a *generic* proof assistant, which means that it has a single meta logic, but can be instantiated with several different object logics. We are using Isabelle/HOL in this course, which means Isabelle instantiated with ‘higher-order logic’. This is the default object logic, so we often just write ‘Isabelle’ for simplicity.

The actual syntax that Isabelle employs in its meta logic is not quite as verbose. It really looks like this

1.  $\wedge x. [\![\text{even}(x); \text{odd}(x)]\!] \Longrightarrow x = 0$
2.  $\wedge x. \text{even}(x) \wedge \text{odd}(x) \Longrightarrow x = 0$
3.  $\wedge x. (\text{even}(x) \wedge \text{odd}(x)) \longrightarrow x = 0$
4.  $\forall x. (\text{even}(x) \wedge \text{odd}(x)) \longrightarrow x = 0$

Quite a bit of the donkey work while building a proof in Isabelle involves moving phrases between the meta logic and the object logic. Typically, a theorem might be phrased as a single statement in the object logic (like the fourth statement in the list above), and in order to use it or prove it, the first thing we need to do is unpack it into the meta logic (so that it looks like the first statement in the list above). We will see an example of this in our next proof.

### 3.3 A second proof

Figure 3.3 gives another standard mathematical theorem and its proof ‘by hand’. Figure 3.4 gives that proof again, this time programmed into Isabelle. Again, I have taken a few typographical liberties to improve the presentation here. The original proof script is also available. Let us now step through the proof line-by-line.

**Theorem 2.** *There is no greatest even number.*

*Proof.* To show that the greatest even number does *not* exist, we shall assume that it *does*, and deduce a contradiction. To this end, suppose there is a greatest even number, and call it  $n$ . But if  $n$  is even, then so is  $n + 2$ , which is greater than  $n$ . This contradicts the assumption that  $n$  is the greatest even number. Therefore, the greatest even number does not exist.  $\square$

Figure 3.3: A proof ‘by hand’ that there is no greatest even number.

```

1 theorem  $\forall n :: \mathbb{Z}. \text{even } n \longrightarrow (\exists m. \text{even } m \wedge m > n)$ 
2 proof clarify
3   fix  $n :: \mathbb{Z}$ 
4   assume  $\text{even } n$ 
5   hence  $\text{even}(n + 2)$  by simp
6   moreover
7   have  $n < (n + 2)$  by simp
8   ultimately
9   show  $\exists m. \text{even } m \wedge n < m$  by blast
10 qed

```

Figure 3.4: A proof in Isabelle that there is no greatest even number.

- 1 The statement of the theorem is a little involved. It can be read as: “for any integer  $n$ , if  $n$  is even, then there exists an  $m$  that is also even and greater than  $n$ ”.

Our goal at this point is as follows:

Assume:	(nothing)
Show:	$\forall n :: \mathbb{Z}. \text{even } n \longrightarrow (\exists m. \text{even } m \wedge m > n)$

- 2 We begin the proof by applying the `clarify` method, which tries to move statements from the object logic into Isabelle’s meta logic, as discussed in the last section. The result of this is that our goal becomes:

Assume:	$\text{even } n$
Show:	$\exists m. \text{even } m \wedge m > n$

- 3 This line can be read as “Let  $n$  be an arbitrary integer.”
- 4 Our goal allows us to assume  $\text{even } n$ , so we use the `assume` command to do this.

- 5 We need to show the existence of an  $m$  that is greater than  $n$  and also even. We shall pick  $m$  to be  $n + 2$ . On this line of the proof we establish that  $n + 2$  is even.
- 6–8 The **moreover** keyword can be thought of as an instruction to set aside the current line of reasoning for a moment, so that another line of reasoning can be established. We can use the **moreover** keyword repeatedly to establish several lines of reasoning. Eventually we will use the **ultimately** keyword to draw together all of the lines of reasoning. The **moreover...ultimately** pattern is handy because it avoids the need to label as many facts, and too many labels can make proofs become unwieldy.
- 9–10 The **ultimately** command makes the facts  $\text{even}(n + 2)$  and  $n < (n + 2)$  available. From these, our desired conclusion follows by instantiating  $m$  to  $n + 2$ .

### 3.4 Two styles of Isabelle proof

The style of Isabelle proof we have used so far is called *structured proof*. This style uses keywords like **assume**, **hence**, and **moreover** in an attempt to mimic the language of proofs done ‘by hand’. There is another style of Isabelle proof, called *procedural proof*. Here, a proof consists solely of a list of instructions that manipulate the goal until there is nothing left to prove. This style can be more concise, but can lead to less readable proofs.

Here is our previous proof repeated using the procedural style.

```

1 theorem  $\forall n :: \mathbb{Z}. \text{even } n \longrightarrow (\exists m. \text{even } m \wedge m > n)$ 
2   apply clarify
3   apply (rule_tac  $x = n + 2$  in exI)
4   apply (rule conjI)
5   apply simp
6   apply (thin_tac  $\text{even } n$ )
7   apply simp
8   done

```

Let us now step through the proof line-by-line.

- 1 At the end of this line, our goal is

Assume:	(nothing)
Show:	$\forall n :: \mathbb{Z}. \text{even } n \longrightarrow (\exists m. \text{even } m \wedge m > n)$

- 2 After we apply the `clarify` method, our goal becomes

Assume:	even $n$
Show:	$\exists m. \text{even } m \wedge m > n$

So far, this is similar to the structured proof we already saw.

- 3 Our goal is to show a statement of the form “there exists  $m$  such that ...”. A handy rule for introducing existentially-quantified variables like this is called `exI` (which is short for ‘existential introduction’). That rule says:

Theorem name:	<code>exI</code>
Assumptions:	$P\ x$
Conclusions:	$\exists y. P\ y$

that is, in order to demonstrate the truth of  $\exists y. P\ y$  for some property  $P$ , it suffices to show that  $P\ x$  holds, for any choice of  $x$ . The conclusion of `exI` exactly matches with what we which to show, once  $P$  is instantiated to the function that checks whether its argument is even and greater than  $n$ . Because the conclusion matches, we can apply the `rule` method. In fact, we can be more precise here: we can specify that when we use the `exI` rule, we shall instantiate the  $x$  in that rule to  $n + 2$ . We do that by issuing the instruction `rule_tac x=n+2 in exI`. (‘Tac’ is short for tactic.) Our goal becomes:

Assume:	even $n$
Show:	$\text{even}(n + 2) \wedge (n + 2) > n$

- 4 Our goal is now to show a statement of the form “ $P \wedge Q$ ”. A handy rule for showing goals that look like this is called `conjI` (which is short for ‘conjunction introduction’). That rule says:

Theorem name:	<code>conjI</code>
Assumptions:	$P$ $Q$
Conclusions:	$P \wedge Q$

that is, to show the truth of  $P \wedge Q$ , it suffices to show the truth of  $P$  and the truth of  $Q$ . After applying the `rule conjI` method, we now have two goals, as follows:

Assume:	even $n$	Assume:	even $n$
Show:	$\text{even}(n + 2)$	Show:	$(n + 2) > n$

- 5 We attack the first goal first. However, we could choose to reorder our goals if we wanted: the **defer** command would send the current goal to the end of the list, and the **prefer**  $N^{\text{th}}$  command would bring the  $N^{\text{th}}$  goal to the start of the list. In this case, the first goal can be dispatched by `simp`.

- 6 Only a single goal remains. It is notable that this goal does not require its assumption that  $n$  is even. We can dispense with this assumption by using the `thin_tac` method. When conducting a large proof, it can be a good idea to ‘thin’ the list of assumptions by removing those that are no longer needed.
- 7 After applying the `simp` method, no goals remain. The proof is complete. (In fact, we could have used this method several steps earlier.)
- 8 A procedural proof is terminated using the **done** keyword.

# Chapter 4

## Inductive proofs

We shall now look at a couple of proofs that use the principle of *mathematical induction*.

Mathematical induction (or just ‘induction’ for short) is a useful tool for showing that some property of interest, say  $P$ , holds for all natural numbers. In symbols we would write:

$$\forall n \in \mathbb{N}. P(n) \tag{4.1}$$

which can be read literally as ‘for every  $n$  that is a natural number (i.e. a non-negative integer), property  $P$  holds of  $n$ ’. We will see some examples of these properties shortly.

Mathematical induction tells us that in order to prove (4.1), it suffices to prove the following two statements, which are (hopefully) easier to prove than just tackling (4.1) head-on. The first thing we must prove is that the property holds of zero, the first natural number:

$$P(0) \tag{4.2}$$

and the second thing we must prove is that whenever the property holds of  $k$  (for any  $k$ ), it also holds of  $k + 1$ :

$$\forall k \in \mathbb{N}. P(k) \Rightarrow P(k + 1) \tag{4.3}$$

Having established just two facts – (4.2), which is known as the *base case*, and (4.3), which is known as the *inductive step* – we can immediately deduce that  $P$  holds for *all* natural numbers. The base case tells us that  $P$  holds for 0. The inductive step with  $k$  instantiated to 0 tells us that since  $P$  holds for 0, it also holds for 1. Then the inductive step (again) with  $k$  instantiated to 1 tells us that since  $P$  holds for 1, it also holds for 2. You can see that this reasoning will extend to all natural numbers (eventually!).



**Remark 1.** We have seen a form of inductive reasoning already, in the Dafny part of the course. When establishing a *loop invariant*, we must show that it holds on entry to the loop (the base case), and that it is maintained by any iteration of the loop (the inductive step). From this we deduce that it must hold at the end of *any* number of iterations of the loop, and hence must hold if the loop finally exits.  $\square$

In summary, the principle of mathematical induction can be expressed concisely as follows:

Theorem name:	<code>nat_induct</code>
Assumptions:	$P\ 0$ $\forall k. P\ k \Rightarrow P(\text{Suc } k)$
Conclusions:	$\forall n. P\ n$

where `Suc  $k$`  is the ‘successor’ of  $k$ , also known as  $k + 1$ .

## 4.1 A proof about triangle numbers

Here are the first few *triangular numbers*:

<code>triangle(0)</code>		0
<code>triangle(1)</code>	•	1
<code>triangle(2)</code>	••	3
<code>triangle(3)</code>	•••	6
<code>triangle(4)</code>	••••	10

The following recursive function can be used to calculate triangular numbers:

$$\text{triangle}(n) = \begin{cases} 0 & \text{if } n = 0 \\ n + \text{triangle}(n - 1) & \text{if } 1 \leq n \end{cases}$$

In Isabelle, we would write that as:

```
fun triangle ::  $\mathbb{N} \Rightarrow \mathbb{N}$  where
  triangle  $n$  = (if  $n$  = 0 then 0 else  $n$  + triangle( $n$  - 1))
```

Triangular numbers can also be obtained via a direct, non-recursive formula, also called a *closed form*:

$$\text{triangle}(n) = \frac{(n + 1)n}{2}$$

**Theorem 3.** For all  $n \geq 0$ ,  $\text{triangle}(n) = (n + 1)n/2$ .

*Proof.* We proceed by mathematical induction.

**Base case.** To show the base case, we must prove that

$$\text{triangle}(0) = (0 + 1)0/2.$$

This is a straightforward consequence of the first clause of the definition of `triangle`.

**Inductive step.** Pick an arbitrary  $k$ , and assume

$$\text{triangle}(k) = (k + 1)k/2$$

as our induction hypothesis. We prove that

$$\text{triangle}(k + 1) = (k + 2)(k + 1)/2$$

by the following chain of equational reasoning:

$$\begin{aligned} \text{triangle}(k + 1) &= \{\text{by the definition of triangle, second clause}\} \\ k + 1 + \text{triangle}(k) &= \{\text{by the induction hypothesis}\} \\ k + 1 + (k + 1)k/2 &= \{\text{by algebraic manipulation}\} \\ (k + 2)(k + 1)/2 \end{aligned}$$

□

Figure 4.1: A proof by hand that the recursive definition and closed form of triangular numbers coincide.

```
1 theorem triangle_closed_form:  $\text{triangle } n = (n + 1) \times n \text{ div } 2$ 
2   apply (induct  $n$ )
3   apply simp+
4   done
```

Figure 4.2: A proof in Isabelle that the recursive definition and closed form of triangular numbers coincide.

In Figure 4.1, we state and prove a theorem that says that the recursive definition and the closed form of triangular numbers coincide.

In Figure 4.2, we state and prove the same theorem in Isabelle. Let us step through the proof line-by-line.

- 1 The statement of the theorem uses *integer division*, which is written using the infix operator ‘`div`’. (Ordinary division is defined on *real* numbers, which we don’t want here.) We have given this theorem a name, just in case we want to invoke it when proving a later theorem.
- 2 The proof is rather short, so we elect for the procedural style. We begin by asking Isabelle to apply the principle of mathematical induction on the variable  $n$ .
- 3 Our goals at the beginning of this line are the base case and the inductive step. In fact, both can be dispatched easily using Isabelle’s simplifier. To this end, we could write `apply simp` twice, or we can use the `+` symbol, as we have done here, to say ‘keep applying this tactic until it can do no more’.

## 4.2 A proof about tetrahedral numbers

Below are the first few *tetrahedral numbers*. Tetrahedral number  $n$  (written `tet( $n$ )`) is the number of spheres stacked in a pyramid whose base is a triangle with sides of length  $n$ .

<code>tet(0)</code>	0
<code>tet(1)</code>	1
<code>tet(2)</code>	4
<code>tet(3)</code>	10
<code>tet(4)</code>	20

The following recursive function can be used to calculate tetrahedral numbers:

$$\text{tet}(n) = \begin{cases} 0 & \text{if } n = 0 \\ \text{triangle}(n) + \text{tet}(n - 1) & \text{if } 1 \leq n \end{cases}$$

In Isabelle, we would write that as:

```
fun tet :: ℕ ⇒ ℕ where
  tet n = (if n = 0 then 0 else triangle n + tet(n - 1))
```

Tetrahedral numbers can also be obtained via a closed form:

$$\text{tet}(n) = \frac{(n+2)(n+1)n}{6}$$

In Figure 4.3, we state and prove a theorem that says that the recursive definition and the closed form of tetrahedral numbers coincide.

In Figure 4.4, we state and prove the same theorem in Isabelle. Let us step through the proof line-by-line.

- 2 This is going to be a fairly hefty proof, so we will use the structured proof style. We begin our structured proof by induction on  $n$ . (Note that when using the jEdit IDE, if you type **proof** (induct ...), a skeletal induction proof is provided for you to fill in.)
- 3 We begin the base case of our induction proof.
- 4 The base case can be dispatched by the simplifier. (Note that **?case** here is an abbreviation, provided by Isabelle, for the actual base case of the proof. It saves us having to type it out.)
- 5 We move on to the next case.
- 6 We begin the inductive step.
- 7 We assume our induction hypothesis. In fact, this line is not necessary – the induction hypothesis is assumed implicitly on line 6. But it doesn't hurt to make it explicit, and doing so gives us an opportunity to give it a memorable label, here  $\text{IH}$ .
- 9–10 Here we establish that  $(k+2) \times (k+1) \times 3$  is divisible by 6. This fact will be handy later on.
- 12–30 Here we establish that  $(k+2) \times (k+1) \times k$  is also divisible by 6, another fact that will be handy shortly. The reasoning is:  $(k+2) \times (k+1) \times k$  is divisible by 2 (line 12), moreover, it is divisible by 3 (lines 13–29), so ultimately we can deduce that it is divisible by 6 (line 30). The proof of divisibility by 3 is done by considering the three possible remainders when dividing  $k$  by 3. We use braces here to delimit the scope of the three local assumptions we make (lines 16, 20, and 24).
- 32–45 We now use a chain of equational reasoning to show that  $\text{tet}(\text{Suc } k)$  is equal to  $(\text{Suc } k + 2) \times (\text{Suc } k + 1) \times \text{Suc } k \text{ div } 6$ . Isabelle provides handy syntax for this. In general, if we want to show that  $A$  is equal to  $D$  by

showing that  $A$  is equal to  $B$ , which is equal to  $C$ , which is in turn equal to  $D$ , we have a few ways to write this. We could use a series of **have** commands:

```
have 1:  $A = B$  by <proof>
have 2:  $B = C$  by <proof>
have 3:  $C = D$  by <proof>
have  $A = D$  using 1 2 3 by simp
```

but this involves thinking up a label for each intermediate fact, which can clutter proofs. Instead, we could use the **moreover...ultimately** construction:

```
have  $A = B$  by <proof>
moreover have  $B = C$  by <proof>
moreover have  $C = D$  by <proof>
ultimately have  $A = D$  by simp
```

which avoids the need for labels, but still requires quite a lot of typing if  $B$  and  $C$  are large formulas – both need to be written out repeatedly. So, we can use the **also...finally** construction:

```
have  $A = B$  by <proof>
also have ... =  $C$  by <proof>
also have ... =  $D$  by <proof>
finally have  $A = D$  by simp
```

in which the previous line's right-hand side can be abbreviated as `....`. In fact, this construction even works for operators other than `=`. For instance, we could construct a chain like  $A = B \leq C = D < E \leq F$  and deduce that  $A < F$ . Of course, all the operators in the chain have to be compatible and pointing in the same direction – we can't deduce much from  $A < B \geq C = D$  for instance.

**41–42** There is a subtlety in the Isabelle proof that we didn't really think about in the proof by hand. It has to do with the fact that we are using integer division in the Isabelle proof, but we used ordinary division in the proof by hand. Isabelle demands that we are much more disciplined with types than ordinary maths does – we cannot just switch freely between real numbers and integers like we can on paper. In the proof by hand, we exploited the identity

$$\frac{a}{c} + \frac{b}{c} = \frac{a+b}{c}$$

which is true for ordinary division. But the analogous identity for integer division

$$a \operatorname{div} c + b \operatorname{div} c = (a + b) \operatorname{div} c$$

does not hold in general. (To see this, try  $a = 1$ ,  $b = 2$ , and  $c = 3$ .) It does work, however, if  $a$  and  $b$  are divisible by  $c$ . That is, the following theorem holds:

Theorem name:	<code>div_add</code>
Assumptions:	$c \operatorname{dvd} a$ $c \operatorname{dvd} b$
Conclusions:	$a \operatorname{div} c + b \operatorname{div} c = (a + b) \operatorname{div} c$

On line 39, we invoke this theorem. We use the `OF` syntax to instantiate the theorem's two assumptions with the two facts we proved earlier.

**43** Sometimes the sledgehammer comes up with some truly arcane proofs!

**Theorem 4.** For all  $n \geq 0$ ,  $\text{tet}(n) = (n + 2)(n + 1)n/6$ .

*Proof.* We proceed by mathematical induction.

**Base case.** To show the base case, we must prove that

$$\text{tet}(0) = (0 + 2)(0 + 1)0/6.$$

This is a straightforward consequence of the first clause of the definition of  $\text{tet}$ .

**Inductive step.** Pick an arbitrary  $k$ , and assume

$$\text{tet}(k) = (k + 2)(k + 1)k/6$$

as our induction hypothesis. We prove that

$$\text{tet}(k + 1) = (k + 3)(k + 2)(k + 1)/6$$

by the following chain of equational reasoning:

$$\begin{aligned} \text{tet}(k + 1) &= \{\text{by the definition of tet, second clause}\} \\ &\text{triangle}(k + 1) + \text{tet}(k) \\ &= \{\text{by Theorem 3}\} \\ &(k + 2)(k + 1)/2 + \text{tet}(k) \\ &= \{\text{by the induction hypothesis}\} \\ &(k + 2)(k + 1)/2 + (k + 2)(k + 1)k/6 \\ &= \{\text{by algebraic manipulation}\} \\ &(k + 3)(k + 2)(k + 1)/6 \end{aligned}$$

□

Figure 4.3: A proof by hand that the recursive definition and closed form of tetrahedral numbers coincide.

```

1 theorem tet  $n = ((n + 2) \times (n + 1) \times n) \text{ div } 6$ 
2 proof (induct  $n$ )
3   case 0
4   show ?case by simp
5 next
6   case (Suc  $k$ )
7   assume IH: tet  $k = (k + 2) \times (k + 1) \times k \text{ div } 6$ 
8
9   have 2 dvd  $(k + 2) \times (k + 1)$  by simp
10  hence *: 6 dvd  $(k + 2) \times (k + 1) \times 3$  by presburger
11
12  have 2 dvd  $(k + 2) \times (k + 1) \times k$  by simp
13  moreover have 3 dvd  $(k + 2) \times (k + 1) \times k$ 
14  proof -
15    {
16      assume  $k \bmod 3 = 0$ 
17      hence 3 dvd  $k$  by presburger
18      hence 3 dvd  $(k + 2) \times (k + 1) \times k$  by fastforce
19    } moreover {
20      assume  $k \bmod 3 = 1$ 
21      hence 3 dvd  $(k + 2)$  by presburger
22      hence 3 dvd  $(k + 2) \times (k + 1) \times k$  by fastforce
23    } moreover {
24      assume  $k \bmod 3 = 2$ 
25      hence 3 dvd  $(k + 1)$  by presburger
26      hence 3 dvd  $(k + 2) \times (k + 1) \times k$  by fastforce
27    } ultimately
28    show 3 dvd  $(k + 2) \times (k + 1) \times k$  by linarith
29  qed
30  ultimately have **: 6 dvd  $(k + 2) \times (k + 1) \times k$  by presburger
31
32  have tet(Suc  $k$ ) = triangle(Suc  $k$ ) + tet  $k$ 
33  by simp
34  also have ... =  $(k + 2) \times (k + 1) \text{ div } 2 + \text{tet } k$ 
35  using triangle_closed_form by simp
36  also have ... =  $(k + 2) \times (k + 1) \text{ div } 2 + (k + 2) \times (k + 1) \times k \text{ div } 6$ 
37  using IH by simp
38  also have ... =  $((k + 2) \times (k + 1) \times 3 + (k + 2) \times (k + 1) \times k) \text{ div } 6$ 
39  using div_add[OF * **] by simp
40  also have ... =  $(k + 2) \times (k + 1) \times (k + 3) \text{ div } 6$ 
41  by (simp add: distrib_left)
42  also have ... =  $(\text{Suc } k + 2) \times (\text{Suc } k + 1) \times \text{Suc } k \text{ div } 6$ 
43  by (metis One_nat_def Suc_1 add commute add_Suc_shift mult.assoc
44    mult commute numeral_3_eq_3 plus_1_eq_Suc)
45  finally show ?case by assumption
46 qed

```

Figure 4.4: A proof in Isabelle that the recursive definition and closed form of tetrahedral numbers coincide.



# Chapter 5

## A verified logic synthesiser

We shall now use Isabelle to implement and verify a logic synthesiser. A logic synthesiser is a program that takes a description of a circuit and outputs a new circuit that has the same behaviour but is ‘optimised’ in some way, perhaps by reducing the number of gates or by reducing the critical path.

### 5.1 Representing circuits

To implement our synthesiser, we first need a data structure that can represent circuits.

```
1 datatype circuit =  
2   NOT circuit  
3 | AND circuit circuit  
4 | OR circuit circuit  
5 | TRUE  
6 | FALSE  
7 | INPUT  $\mathbb{Z}$ 
```

This is an example of a *recursive* definition. It says that a circuit is either a constant TRUE, a constant FALSE, or an input terminal (identified by an integer), or else is built by combining smaller circuits with NOT, AND, and OR gates.

Here are some examples of circuits we can define using this data structure. Table 5.1 shows how each example can be depicted graphically.

```
1 definition circuit1 == AND (INPUT 1) (INPUT 2)  
2 definition circuit2 == OR (NOT circuit1) FALSE  
3 definition circuit3 == NOT (NOT circuit2)  
4 definition circuit4 == AND circuit3 (INPUT 3)
```

It is notable that our data structure only supports circuits that have no fan-out. That is, no gate has its output connected (directly) to more than one other gate.

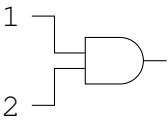
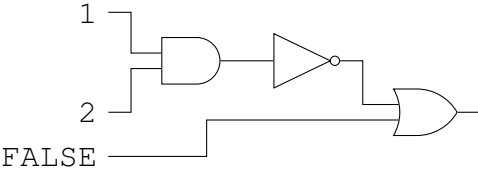
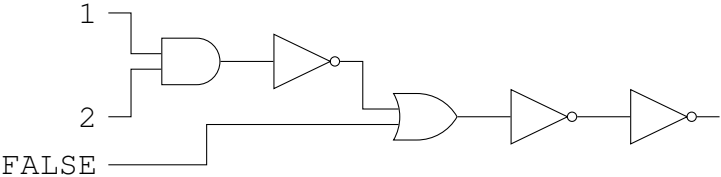
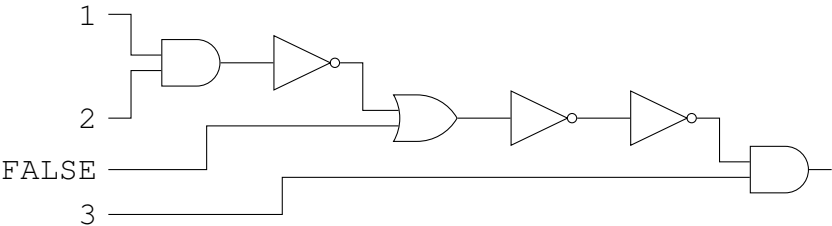
circuit1:	
circuit2:	
circuit3:	
circuit4:	

Table 5.1: Examples of circuits

Our circuits can have any number of inputs, but they always have exactly one output wire. This restriction will simplify the task of implementing a synthesiser. (An ‘extension’ task involves working out how to avoid this restriction.)

## 5.2 Simulating circuits

Next we define a function called *simulate* that can be used to simulate a circuit. The function takes two arguments: a circuit and a wire valuation. A wire valuation, written  $\rho$ , assigns a truth value (*True* or *False*) to every input. The output of the *simulate* function is the truth value of the output wire.

```

1 fun simulate where
2   simulate (AND c1 c2)  $\rho$  = ((simulate c1  $\rho$ )  $\wedge$  (simulate c2  $\rho$ ))
3 | simulate (OR c1 c2)  $\rho$  = ((simulate c1  $\rho$ )  $\vee$  (simulate c2  $\rho$ ))
4 | simulate (NOT c)  $\rho$  = ( $\neg$  (simulate c  $\rho$ ))
5 | simulate TRUE  $\rho$  = True

```

```

6 | simulate FALSE  $\rho$  = False
7 | simulate (INPUT i)  $\rho$  =  $\rho$  i

```

The *simulate* function is defined recursively. This means that the result of the function on a large circuit is built up by first simulating the smaller subcircuits that it contains. For instance, line 2 says that to simulate a circuit that is formed by ANDing together the outputs of two subcircuits (*c1* and *c2*), one should simulate both *c1* and *c2* and then use logical conjunction ( $\wedge$ ) to combine the two truth values thus obtained. Lines 3 and 4 use logical disjunction ( $\vee$ ) and logical negation ( $\neg$ ). To simulate a circuit that consists solely of an input port (line 7), we consult the wire valuation  $\rho$ .

As an example: if  $\rho$  is defined such that  $\rho(1) = \text{True}$ ,  $\rho(2) = \text{False}$ , and  $\rho(3) = \text{True}$  then *simulate* circuit4  $\rho$  will return *True*.

## 5.3 Structural induction on circuits

Let us define a function called *mirror* that takes a circuit and transforms it into one in which the two inputs to each AND or OR gate have been switched around. This function is not actually *useful* for anything; its purpose is to demonstrate a transformation that we might do to a circuit, and how we can prove that this transformation doesn't change the behaviour of the circuit.

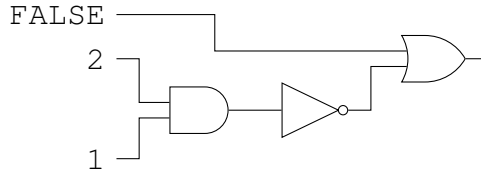
```

1 fun mirror where
2   mirror (NOT c) = NOT (mirror c)
3 | mirror (AND c1 c2) = AND (mirror c2) (mirror c1)
4 | mirror (OR c1 c2) = OR (mirror c2) (mirror c1)
5 | mirror TRUE = TRUE
6 | mirror FALSE = FALSE
7 | mirror (INPUT i) = INPUT i

```

The *mirror* function is, like *simulate*, defined recursively. The clauses on lines 3 and 4 are the interesting ones. Line 3 says that in order to mirror a circuit of the form AND *c1* *c2*, we should first mirror *c1* and *c2* individually, and then feed their outputs, swapped around, into an AND gate. Line 4 is similar, but for OR gates. Line 2 says that to mirror a circuit of the form NOT *c*, we should mirror *c* on its own, and then restore the NOT gate on its output. Once the transformation reaches the 'leaves' of the circuit 'tree' (that is, TRUE, FALSE, and INPUT *i*), there is nothing more for the transformation to do.

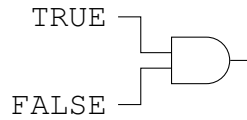
As an example, *mirror* circuit2 looks like the picture below.



We now wish to prove that this transformation is correct. In order to do so, we must first be precise about what we mean by ‘correct’. What we cannot say, for instance, is that the transformation does not change the circuit at all, which is what the following theorem tries to claim.

**theorem** *mirror*  $c = c$   
**oops**

Indeed, Isabelle is able to contradict that theorem for us even before we have started a (doomed) effort to prove it, by exhibiting the following simple counterexample.



What we *can* prove, however, is that our transformation will not change the result of simulating the circuit, whatever wire valuation we provide.

**theorem** *mirror\_is\_sound*: *simulate* (*mirror*  $c$ )  $\rho = \textit{simulate}  $c$   $\rho$$

How can we prove this theorem?

We use a principle called *structural induction*. This is a generalisation of mathematical induction. It can be used on any data structure that is defined recursively, such as our *circuit*. The principle of structural induction for our *circuit* data structure is as follows:

<b>Theorem name:</b>	<code>circuit.induct</code>
<b>Assumptions:</b>	$\forall k. P\ k \Rightarrow P(\text{NOT } k)$ $\forall k_1, k_2. (P\ k_1 \wedge P\ k_2) \Rightarrow P(\text{AND } k_1\ k_2)$ $\forall k_1, k_2. (P\ k_1 \wedge P\ k_2) \Rightarrow P(\text{OR } k_1\ k_2)$ $P(\text{TRUE})$ $P(\text{FALSE})$ $\forall i. P(\text{INPUT } i)$
<b>Conclusions:</b>	$\forall c. P\ c$

The conclusion of the `circuit.induct` theorem is that for every circuit  $c$ , some property  $P$  holds. In order to reach this conclusion, our theorem requires six

assumptions, one for each of the ways of building circuits. The first says that if the property holds of an arbitrary circuit  $k$ , then it still holds if a NOT gate is put on the output of  $k$  (which we write as  $\text{NOT } k$ ). The second says that if the property holds of two arbitrary circuits  $k_1$  and  $k_2$ , then it also holds of a circuit that is constructed by combining the outputs of  $k_1$  and  $k_2$  using an AND gate (which we write as  $\text{AND } k_1 \ k_2$ ). The third is similar, but for OR gates. The fourth says that the property must hold for the constant-true circuit. The fifth and sixth deal with the other two non-recursive constructions.

**Remark 2.** The best way to understand structural induction is to see how mathematical induction ‘drops out’ as a special case. To this end, consider the following data structure:

```
datatype myNat =
  ZERO
| SUC myNat
```

Every *myNat* is either ‘ZERO’ or is built from an existing *myNat* using the ‘SUC’ construction. We can derive a principle of structural induction for this data structure following the same pattern that we used for `circuit.induct`, like so:

Theorem name:	<code>mynat.induct</code>
Assumptions:	$P(\text{ZERO})$ $\forall k. P\ k \Rightarrow P(\text{SUC } k)$
Conclusions:	$\forall n. P\ n$

If you compare this with the `nat_induct` theorem from the previous section, you will see that it is the same (except for slightly different syntax). In fact, the recursive definition above is exactly how natural numbers are actually defined in Isabelle!  $\square$

As for the proof itself, we can proceed as shown in Figure 5.1. Isabelle can manage this proof with very little assistance, as shown in Figure 5.2.

## 5.4 A simple circuit optimiser

Next, we define a function called *opt<sub>NOT</sub>* that takes a circuit and transforms it into one in which all pairs of consecutive NOT gates have been removed. The motivation for this is that two successive inversions have no effect on the behaviour of a circuit, and by removing them we may reduce the circuit’s area and energy usage.

```
1 fun optNOT where
2   optNOT (NOT (NOT c)) = optNOT c
```

**Theorem 5.** For all circuits  $c$  and all wire valuations  $\rho$ , we have

$$\text{simulate } (\text{mirror } c) \rho = \text{simulate } c \rho.$$

*Proof.* First define the property  $P$  such that

$$P(c) = (\text{simulate } (\text{mirror } c) \rho = \text{simulate } c \rho)$$

Our aim is to show  $P(c)$  holds for all circuits  $c$ . We proceed by structural induction on  $c$ . We will present just two of the six cases.

**Case ‘TRUE’.** To show this case, we must prove  $P(\text{TRUE})$ , which is

$$\text{simulate } (\text{mirror TRUE}) \rho = \text{simulate TRUE } \rho.$$

This is a straightforward consequence of the definition of *mirror* (fourth clause) and the definition of *simulate* (fourth clause).

**Case ‘AND’.** To show this case, we pick arbitrary circuits  $k_1$  and  $k_2$ , assume  $P(k_1)$  and  $P(k_2)$ , and prove  $P(\text{AND } k_1 k_2)$ . That is, we assume

$$\text{simulate } (\text{mirror } k_1) \rho = \text{simulate } k_1 \rho \quad (5.1)$$

$$\text{simulate } (\text{mirror } k_2) \rho = \text{simulate } k_2 \rho \quad (5.2)$$

as our induction hypotheses, and must prove

$$\text{simulate } (\text{mirror } (\text{AND } k_1 k_2)) \rho = \text{simulate } (\text{AND } k_1 k_2) \rho.$$

This is a straightforward consequence of the definition of *mirror* (second clause), the definition of *simulate* (second clause), and the induction hypotheses.  $\square$

Figure 5.1: A proof by hand that mirroring a circuit doesn’t change its behaviour.

```

1 theorem simulate (mirror c)  $\rho$  = simulate c  $\rho$ 
2   apply (induct c)
3   apply auto
4   done

```

Figure 5.2: A proof in Isabelle that mirroring a circuit doesn’t change its behaviour.

```

3 | optNOT (NOT c) = NOT (optNOT c)
4 | optNOT (AND c1 c2) = AND (optNOT c1) (optNOT c2)
5 | optNOT (OR c1 c2) = OR (optNOT c1) (optNOT c2)
6 | optNOT TRUE = TRUE
7 | optNOT FALSE = FALSE
8 | optNOT (INPUT i) = INPUT i

```

The *opt*<sub>NOT</sub> function is, like *simulate*, defined recursively. The clause on line 2 is the most important one. It says that the result of optimising a circuit of the form NOT (NOT c) is obtained by discarding the two NOT gates and then optimising c. The clause on line 3 applies when the circuit takes the form NOT c but doesn't match the pattern of the first clause (the clauses are tried in order). It says that the result of optimising a circuit of this form is obtained by first optimising c (that is, the circuit with the final NOT gate removed) and then attaching a NOT gate to the output of the optimised circuit. AND and OR gates are optimised similarly (lines 4 and 5). Once the optimisation reaches the 'leaves' of the circuit 'tree' (that is, TRUE, FALSE, and INPUT i), there is nothing more for the optimiser to do.

As an example, let us consider the effect of *opt*<sub>NOT</sub> on our example *circuit4*. Figure 5.3 shows how *opt*<sub>NOT</sub> is repeatedly applied to smaller and smaller parts of the original circuit, eventually obtaining a fully optimised circuit.

## 5.5 Rule induction

Alongside mathematical induction and structural induction, there is a third type of induction that can be useful, called *rule induction*. Mathematical induction exploits the observation that any natural number is obtained by applying the 'successor' function finitely-many times to 'zero'. Structural induction exploits the observation that any instance of a recursively-defined data structure is obtained by applying the recursive constructors (like AND) of that data structure finitely-many times to the non-recursive constructors (like TRUE). Rule induction, on the other hand, exploits the observation that the result of a recursive function, say *f*, involves finitely-many recursive calls to *f*. The aim is to prove that some property of interest holds of the result of *f*, under the assumption that the property already holds for any recursive calls to *f* that the function makes.

For instance, consider the following recursive function

```

1 fun fib ::  $\mathbb{N} \Rightarrow \mathbb{N}$  where
2   fib (Suc (Suc k)) = fib k + fib (Suc k)
3 | fib (Suc 0) = 1
4 | fib 0 = 1

```

and suppose we wish to prove that  $\text{fib } n \geq n$  for all natural numbers *n*. Structural induction and mathematical induction are not good fits here, because the

definition of `fib` is not *syntax-directed*. That is, we do not have a 1-to-1 correspondence between the two constructors of the data structure (`Suc` and `0`) and the three clauses of `fib` (`Suc (Suc k)`, `Suc 0`, and `0`). In contrast, both *mirror* and *simulate* are syntax-directed, and this is what made the structural induction proofs so straightforward.

Every recursive function is associated with a principle of rule induction. If the function is called  $f$ , then its principle of rule induction is stored in the theorem  $f.induct$ . We shall show how this works using `fib` as an example.

First, it is convenient to rewrite the definition of `fib` using *inference rules*, where any statements above the line are the assumptions and the statement below the line is the conclusion, like so:

$$\frac{\text{fib } k = f_1 \quad \text{fib}(\text{Suc } k) = f_2}{\text{fib}(\text{Suc}(\text{Suc } k)) = f_1 + f_2} \quad \frac{}{\text{fib}(\text{Suc } 0) = 1} \quad \frac{}{\text{fib } 0 = 1}$$

The principle of rule induction for the function `fib` says that property  $P$  holds for all natural numbers ( $\forall n. P n$ ) if the following three things (which can be seen to resemble the trio of inference rules above) can all be proved:

$$\frac{P k \quad P(\text{Suc } k)}{P(\text{Suc}(\text{Suc } k))} \quad \frac{}{P(\text{Suc } 0)} \quad \frac{}{P 0 = 1}$$

In other words: if we can prove for arbitrary  $k$  that if  $P$  holds of both  $k$  and  $k + 1$  then it holds of  $k + 2$ , and if we can prove that  $P$  holds of both  $1$  and  $0$ , then we can deduce that  $P$  holds for all natural numbers.

Theorem name:	<code>fib.induct</code>
Assumptions:	$\forall k. (P k \wedge P(\text{Suc } k)) \Rightarrow P(\text{Suc}(\text{Suc } k))$ $P(\text{Suc } 0)$ $P 0$
Conclusions:	$\forall n. P n$

To prove our desired theorem about `fib` using this principle, it is natural to instantiate  $P$  so that  $P(n) = (\text{fib } n \geq n)$ .

## 5.6 Proving termination

As mentioned above, every recursive function in Isabelle is associated with its own principle of rule induction. (Technically, a non-recursive function could be associated with a rule induction principle too, but it wouldn't be a very interesting principle.) But Isabelle will only produce the rule induction principle once it is satisfied that the function always terminates. A function like



```

1 fun silly ::  $\mathbb{N} \Rightarrow \mathbb{N}$  where
2   silly (Suc k) = silly (Suc k) + silly k
3 | silly 0 = 1

```

has non-terminating recursion, so does not get a rule induction principle.

When you introduce a recursive function using the **fun** keyword, you are implicitly instructing Isabelle to try to prove automatically that your function terminates. Usually, Isabelle manages this fine. But sometimes, the reason why a function terminates is a little complicated, and Isabelle needs some guidance before it can see it.

In these situations, you can change **fun** into **function**. When you do that, you need to follow the function's definition with two proofs: a proof that the list of input patterns are exhaustive, and a proof that the function always terminates. For the `fib` function, the input patterns are `Suc (Suc k)`, `Suc 0`, and `0`. The first pattern applies when the input is greater than or equal to 2, the second applies when the input is equal to 1, and the third applies when the input is equal to 0, so altogether, these input patterns are exhaustive, and we can write **by** `pat_completeness auto` to dispatch this proof obligation. As for termination: we must come up with a *measure*; that is, a way to turn the actual parameters of a call to `fib` into a natural number in such a way that the measure of each recursive call is always less than the measure of the original call. For `fib`, this is easy: we can simply use the parameter itself. Thus, `fib` can be defined in a more long-winded way like so:

```

1 function fib ::  $\mathbb{N} \Rightarrow \mathbb{N}$  where
2   fib (Suc (Suc k)) = fib k + fib (Suc k)
3 | fib (Suc 0) = 1
4 | fib 0 = 1
5 by pat_completeness auto
6 termination by (relation "measure ( $\lambda n. n$ )", auto)

```

Since the  $\lambda n. n$  measure is a little too simple for the idea to come across properly, here is a slightly more interesting example.

```

1 function g ::  $\mathbb{N} \Rightarrow \mathbb{N}$  where
2   "g n = (if n mod 4 = 0 then n else g (n + 1))"
3 by pat_completeness auto

```

At first glance, it appears that `g` might not terminate, because the recursive call `g (n + 1)` uses a larger parameter than the original call `g n`. However, `g` *does* in fact terminate, because it only increases the parameter until it reaches a multiple of 4.

For instance, here is an execution trace of `g 5`:

$$g\ 5 \rightarrow g\ 6 \rightarrow g\ 7 \rightarrow g\ 8 \rightarrow 8$$

So we need our measure,  $\theta$ , to obey the following:

$$\theta(5) < \theta(6) < \theta(7) < \theta(8) \quad (5.3)$$

One possible definition for  $\theta$  is as follows:

```
1 fun  $\theta$  ::  $\mathbb{N} \Rightarrow \mathbb{N}$  where
2   " $\theta$  n = (case n mod 4 of
3     0  $\Rightarrow$  n
4     | 1  $\Rightarrow$  n + 6
5     | 2  $\Rightarrow$  n + 4
6     | 3  $\Rightarrow$  n + 2) "
```

It can be observed that Equation (5.3) holds with this definition of  $\theta$ . It is then possible to complete the proof of termination by issuing the following incantation:

```
1 termination
2 proof (relation "measure  $\theta$ ", simp,
3   simp only: measure_def inv_image_def, clarify)
```

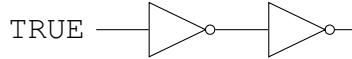
and then proving that  $\theta(n+1) < \theta(n)$  holds whenever  $0 < n \bmod 4$ .

## 5.7 Verifying our optimiser

We now wish to prove that this optimisation is correct. Like with the *mirror* transformation before, we must first be precise about what we mean by ‘correct’. What we cannot say, for instance, is that the optimisation does not change the circuit at all, which is what the following theorem tries to claim.

```
theorem  $opt_{NOT}$  c = c
oops
```

Indeed, Isabelle is able to contradict that theorem for us even before we have started a (doomed) effort to prove it, by exhibiting the following simple counterexample.



What we *can* prove, however, is that our optimisation will not change the result of simulating the circuit, whatever wire valuation we provide.

```
theorem opt_NOT_is_sound: simulate ( $opt_{NOT}$  c)  $\rho$  = simulate c  $\rho$ 
```

How can we prove this theorem?

Structural induction is tempting, but is actually not a good fit because  $opt_{NOT}$  is not syntax-directed – there are more clauses in the definition of  $opt_{NOT}$  than there are constructors for the *circuit* data structure. Therefore, we turn to *rule induction*. The basic observation is that the result of  $opt_{NOT} \ c$  is obtained by repeatedly unfolding the seven clauses that define  $opt_{NOT}$ , like we saw in Figure 5.3. If we can show that unfolding each *individual* clause is correct, then it follows that repeated unfolding of these clauses is also correct.

Another way to think of this is: we are showing that each clause in the definition of  $opt_{NOT}$  is correct, and to do so we can assume that any recursive calls it contains are *already* correct. This reasoning is legitimate providing the function does not contain infinite recursion (and Isabelle checks this when the function was defined). This assumption is known as the *induction hypothesis*.

So let us examine each clause in turn. It is easier to begin at the bottom, because the last three clauses do not contain any recursive calls. Let us prove our theorem in the case where the result of  $opt_{NOT} \ c$  is obtained using the clause on line 8. That means  $c$  must take the form  $INPUT \ i$  for some  $i$ . To prove that this clause is correct, we must prove that

$$simulate \ (opt_{NOT} \ (INPUT \ i)) \ \rho = simulate \ (INPUT \ i) \ \rho$$

which is a straightforward consequence of the definition of  $opt_{NOT}$ . The clauses on lines 6 and 7 are similarly easy.

Now let us instead suppose that the result of  $opt_{NOT} \ c$  is obtained using the clause on line 5. That means  $c$  must take the form  $OR \ c1 \ c2$  for some  $c1$  and  $c2$ . To prove that this clause is correct, we must prove that

$$simulate \ (opt_{NOT} \ (OR \ c1 \ c2)) \ \rho = simulate \ (OR \ c1 \ c2) \ \rho \quad (5.4)$$

under the assumption that the recursive calls on  $c1$  and  $c2$  are already correct. That is, we can assume the following induction hypotheses:

$$simulate \ (opt_{NOT} \ c1) \ \rho = simulate \ c1 \ \rho \quad (5.5)$$

and

$$simulate \ (opt_{NOT} \ c2) \ \rho = simulate \ c2 \ \rho \quad (5.6)$$

We can prove (5.4) like so:

$$\begin{aligned}
& \text{simulate } (\text{opt}_{\text{NOT}} (\text{OR } c1 \ c2)) \ \rho \\
&= \{\text{unfolding line 5 of definition of } \text{opt}_{\text{NOT}}\} \\
& \text{simulate } (\text{OR } (\text{opt}_{\text{NOT}} \ c1) \ (\text{opt}_{\text{NOT}} \ c2)) \ \rho \\
&= \{\text{unfolding line 3 of definition of } \text{simulate}\} \\
& (\text{simulate } (\text{opt}_{\text{NOT}} \ c1) \ \rho) \ \vee \ (\text{simulate } (\text{opt}_{\text{NOT}} \ c2) \ \rho) \\
&= \{\text{unfolding (5.5)}\} \\
& (\text{simulate } c1 \ \rho) \ \vee \ (\text{simulate } (\text{opt}_{\text{NOT}} \ c2) \ \rho) \\
&= \{\text{unfolding (5.6)}\} \\
& (\text{simulate } c1 \ \rho) \ \vee \ (\text{simulate } c2 \ \rho) \\
&= \{\text{folding line 3 of definition of } \text{simulate}\} \\
& \text{simulate } (\text{OR } c1 \ c2) \ \rho
\end{aligned}$$

The clauses on lines 3 and 4 are proved similarly. The final clause, on line 2, requires us to prove

$$\text{simulate } (\text{opt}_{\text{NOT}} (\text{NOT } (\text{NOT } c))) \ \rho = \text{simulate } c \ \rho$$

assuming

$$\text{simulate } (\text{opt}_{\text{NOT}} \ c) \ \rho = \text{simulate } c \ \rho$$

which is a straightforward matter of unfolding definitions. Putting it all together, we can conclude that the entire  $\text{opt}_{\text{NOT}}$  function is correct.

Isabelle can actually handle all of this reasoning with very little assistance. The entire proof takes just one line.

```
theorem opt_NOT_is_sound: simulate (opt_NOT c) ρ = simulate c ρ
by (induct rule: opt_NOT.induct, auto)
```

The proof instructs Isabelle first to perform rule induction, which leaves us with seven proof goals – one for each of the clauses of  $\text{opt}_{\text{NOT}}$ . Each of these proof goals is sufficiently straightforward that one application of `auto` is enough to dispense with all of them.

To examine the proof in a little more detail, it could be rewritten in the following more explicit form.

```
theorem opt_NOT_is_sound: simulate (opt_NOT c) ρ = simulate c ρ
apply (induct rule: opt_NOT.induct)
apply auto
done
```

```

optNOT circuit4
= {unfolding definition of circuit4}
optNOT (AND circuit3 (INPUT 3))
= {unfolding line 4 of definition of optNOT}
AND (optNOT circuit3) (optNOT (INPUT 3))
= {unfolding line 8 of definition of optNOT}
AND (optNOT circuit3) (INPUT 3)}
= {unfolding definition of circuit3}
AND (optNOT (NOT (NOT circuit2))) (INPUT 3)}
= {unfolding line 2 of definition of optNOT}
AND (optNOT circuit2) (INPUT 3)}
= {unfolding definition of circuit2}
AND (optNOT (OR (NOT circuit1) FALSE)) (INPUT 3)}
= {unfolding line 5 of definition of optNOT}
AND (OR (optNOT (NOT circuit1)) (optNOT FALSE)) (INPUT 3)}
= {unfolding line 7 of definition of optNOT}
AND (OR (optNOT (NOT circuit1)) FALSE) (INPUT 3)}
= {unfolding definition of circuit1}
AND (OR (optNOT (NOT (AND (INPUT 1) (INPUT 2)))) FALSE) (INPUT 3)}
= {unfolding line 3 of definition of optNOT}
AND (OR (NOT (optNOT (AND (INPUT 1) (INPUT 2)))) FALSE) (INPUT 3)}
= {unfolding line 4 of definition of optNOT}
AND (OR (NOT (AND (optNOT (INPUT 1)) (optNOT (INPUT 2)))) FALSE) (INPUT 3)}
= {unfolding line 8 of definition of optNOT}
AND (OR (NOT (AND (INPUT 1) (INPUT 2))) FALSE) (INPUT 3)}

```

Figure 5.3: How our optimisation works on our example `circuit4`. The expression on the first line is repeatedly rewritten to an equivalent form using the definitions of the circuit and of the *opt*<sub>NOT</sub> function.

# Bibliography

- [1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.
- [2] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *ACM Symp. on Principles of Programming Languages (POPL)*, 2014.
- [3] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [4] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox. Verified compilation on a verified processor. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, 2019.