Isabelle and Internet Protocol Theory: A User Guide

Michael Roberts
email: roberm67@mcmaster.ca

August 14, 2024

Contents

1 Introduction

Isabelle is a generic proof assistant originally developed at the University of Cambridge and the Technische Universität München. It is used and improved by many researchers in software engineering, computer science, and mathematics worldwide. This document outlines how Isabelle can be used to prove mathematical theorems in the context of Internet security, through tools for IP addresses, protocols, and ports.

This document contains 2 substantive chapters. Chapter 2 is a beginner's guide to Isabelle, showing a reader familiar with some discrete mathematics how to use Isabelle. The chapter will explore propositional calculus (i.e. boolean logic), predicate calculus (i.e. logic regarding existential and universal quantification), sets, and relations avend functions.

Chapter 3 is a guide to the suite I developed about internet security containing tools for IP addresses, protocols, and ports. Each tool will be thoroughly explained to show the reader the potential of the suite.

2 Beginner's Guide to Isabelle

2.1 Setup

The latest version of Isabelle is available for installation here.

2.1.1 Windows Setup

On Windows 10 or 11, download Isabelle2024.exe. Windows will flag your download as uvensafe, due to its infrequent usage, although override the warning and keep the download. Then, run the installer, and unpack the application files to your desired directory. Within the directory, Isabelle's files will be in the Isabelle2024 folder (hereinafter %ISABELLE_HOME%).

Then, run the application, Isabelle2024.exe. Its path is %ISABELLE_HOME%\Isabelle2024.exe. Alternatively, open Isabelle's Cygwin terminal (its path is %ISABELLE_HOME%\Cygwin-Terminal.bat), and execute

```
isabelle client
```

Afterward, you will have opened the interactive Isabelle client.

2.1.2 Setup for Mac, Linux, and Docker

Follow the guidelines in the installation link above. While very similar, there are some slight differences in setup with a Windows computer.

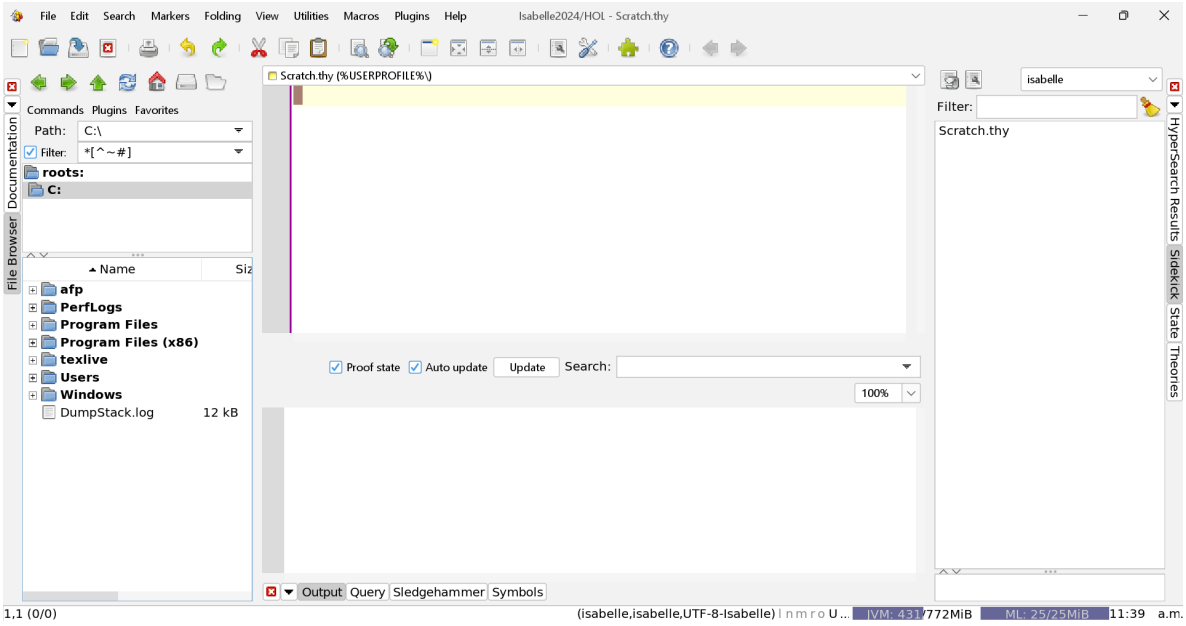

Figure 1: The Isabelle client, initially after startup

2.2 Environment

Below is a guide to the features of the Isabelle client used in this user guide. The main, central portion is a text editor, where Isabelle documents are written. The centre-left has two modes: file browser, to find files to open, and documentation, which contains example t

The bottom-centre contains four modes, Output, Query, Sledgehammer and Symbols. Output is the default, as this is where the fruit of your labour manifests. If the proof state and auto-update checkmarks are not checked, check them for an easier debugging/output experience.

Query is a keyword search utility for theorems and constants. Sledgehammer searches for proofs, although the **t**ry keyword usually suffices,and always within this user guide. If you want to use a non-ASCII key, check out the Symbols tab. Isabelle has codes, starting with a backslash, for every non-ASCII symbol it recognizes.

The right sidebar has several modes, although keep it in Sidekick mode. This enables you to access an outline of your file.

There are a vast amount of options available in the top menus. However, they will not be used in this user guide. Feel free to check them out yourself.

2.3 Your First Program

Within the Scratch.thy in your user profile (hereinafter%USERPROFILE%) enter:

```
theory Scratch imports Main begin

theorem True ..

end
```

Click within *theorem True*. You will see the goals you need to prove in the Output tab. Here, there is only 1 goal, *True*. Click after the two dots. You will see **theorem** True, indicating True is a valid theorem.

2.3.1 Understanding the Program

All Isabelle theory files begin **theory** *TheoryName* **imports** Main **begin** and end with the **end**. All theory files have the *.thy* extension, and *TheoryName* should be replaced by the file's name, excluding the *.thy* extension.

Another theory may be imported instead of Main, although ensure your theory imports Main, whether directly or indirectly. This is required to use Isabelle's Higher Order Logic (HOL) utility.

theorem *True* declares that *True* is a theorem. However, the declaration is insufficient, a theorem must be proven. Normally, a more elaborate proof would be required, however, True is such a trivial theorem. Therefore, .. can be used, signifying proof by triviality.

When a theorem is declared and during its proof process, you see its goals in the output tab. After the process is finished, you see the theorem validated by Isabelle as a theorem.

2.3.2 Comments

Comments may be written in a .thy file, even before the *theory* keyword and after the *end* keyword. They begin with *(** and end with ***)* You may escape the end of a comment with ***)* by replacing it with *** \)*

2.4 Prepositional Logic

Prepositional logic is the logic surrounding boolean expressions. Isabelle can be used to verify valid boolean expressions, where valid expressions always evaluate to *True* no matter how variables are assigned.

This guide assumes some proficiency in propositional logic. A good introductory text is Chapter 3 *A Logical Approach to Discrete Math*, by David Gries and F.B. Schneider.

2.4.1 A Structured Proof

A non-trivial theorem to start with is the symmetry of conjunction, or

$$p \wedge q = q \wedge p$$

And here is its proof in Isabelle:

```
theorem "(p & q) = (q & p)"
proof
  assume 0: "p & q"
  from 0 have "(p = q) = (p | q)" by simp
  then have "(q = p) = (q | p)" by auto
  then show "q & p" by auto
next
  assume "q & p"
  hence "(q = p) = (q | p)" by simp
  hence "(p = q) = (p | q)" by auto
  thus "p & q" by auto
qed
```

This was a lot. Using the above Isabelle code, the symmetry of conjunction was proven in the following steps:

1. Use the Golden Rule ($p \wedge q \equiv p \equiv q \equiv p \vee q$) to expand the conjunction
2. Use the symmetry of $\equiv$ and the symmetry of $\vee$ to set up the second usage of the Golden Rule
3. Use the Golden Rule (again) to transform the expansion into the symmetric conjunction.

QED.

Normally, this proof would only need to be done once. Because the Golden rule, the symmetry of $\equiv$ and the symmetry of $\vee$ are all equivalences, the proof would only need to be done one way. However, Isabelle is built on implication and *from A have B* does not imply *from B have A*. While it is possible to copy the right-hand side as the left-hand side is being transformed, such a process is very strenuous. Therefore, mutual implication is the way to go.

In Isabelle & represents $\wedge$ and | represents $\vee$. You could also type \& to get $\wedge$ or \| to get $\vee$ within the Isabelle editor. However, when writing Isabelle code outside the Isabelle client, you should use & and | instead for portability.

If an Isabelle expression contains a space, it must be surrounded by double-quotes. Otherwise, the client will interpret the text as two separate units, instead of one contiguous expression.

Structured proofs in Isabelle begin with the *proof* keyword and end with the *qed* keyword. Proof of different subgoals (such as $p \wedge q \Longrightarrow q \wedge p$ and $q \wedge p \Longrightarrow p \wedge q$) here), are separated by the keyword *next*.

For any expression in Isabelle, you can name it through *name: expression*. Then you can reference it in a future expression by using **from** *name* before the expression's normal preceding keyword or **using** *name* before **by** or ... The name can be any alphanumeric sequence. If the name is given to a **theorem** or another top-level construct, the convention is to give it a meaningful name. For example, this theorem's name could be *conj_sym*. Within a non-top-level construct, the name is not accessible outside the construct. Thus, giving the assumption a useful name is pointless. Instead, the convention is to enumerate these named expressions, starting from 0. If a lower number is available within a construct, reuse that number before using a higher number.

Therefore, start with the assumption of $p \wedge q$. Assumptions are proceeded with the *assume* keyword in Isabelle. The syntax is for an assumed *expression* is:

```
assume expression
```

Then, unpack the assumption using the Golden Rule. The resultant expression is not universally true. Rather, it follows from the assumption $p \wedge q$. Therefore, it is referenced using its abbreviation, 0:

```
from 0 have "(p = q) = (p | q)" by simp
```

Equivalently:

```
have "(p = q) = (p | q)" using 0 by simp
```

The latter notation will be used, as facts used are utilized by the proof methods (*simp* here), not by the expression.

Isabelle uses proof methods to prove that an expression is true or follows, directly or indirectly, from an assumption. All proof methods follow the expression and any facts referenced by the *using* keyword. The simplest proof method is triviality (..). For example,

```
have "p = q" using 2 ..
```

All other proof methods are separated from the expression and the facts used through the *by* keyword. There are many proof methods, see this cheat sheet for a list). Below are the ones used in this user guide:

- rule *X*. Transform A to B using rule X. While a good pedagogical tool, superior proof methods offered by Isabelle should be harnessed by users familiar with the relevant theory.
- *simp*. Use the default simplification rules (type **thm** simps) into the Isabelle client for a complete list. If you want to also use other facts, add them. For example, if you want to use facts 5 and 6:

  ```
  by (simp add:5 6)
  ```

  simp only solves one subgoal by default. To solve all subgoals using *simp*, replace it with *simp_all*.
- *auto*. Solves all subgoals, using the utilities of *simp* plus more. If a useful fact is not included, you can add them as a simplification. For example, add facts 5 and 6:

  ```
  by (auto simp:5 6)
  ```

.

- *blast*. Used in proofs involving quantifiers and sets. It will be discussed below.

Besides proof methods, proof tactics can be used. For example, use *induct_tac xs* for induction on xs, and *case_tac x* to split the proof into two sub-proofs, one when x and one when $\neg x$. However, unless the proof is a one-liner, tactics should not be used in this fashion.

Multiple facts in *from* and *using* are separated by spaces, similar to how they are separated in *auto simp* or *simp add*. Multiple proof methods or proof tactics are separated by commas.

If a non-assumed expression is an intermediate towards the goal, precede it with the keyword *have*. If it is the goal, preceede it with the keyword *show*. Using *then* before *have* or *show* gets automatic access to the previously defined fact, and results in an error if there is none. *hence* is an abbreviation for *then have* and *thus* is an abbreviation for *then show*.

Thus concludes the explanation of this proof. With the steps and the syntax, the proof becomes understandable.

2.4.2 Shortcuts

The structured proof wasn't really necessary. The theorem could be proven in one line. Here are two different ways:

```
theorem "(p & q) = (q & p)"
  by auto

theorem  "(p & q) = (q & p)"
  by (case_tac p, simp_all)
```

The first way uses *auto*, which is powerful enough to resolve virtually any boolean proof.

The second way uses case analysis to simplify it for the weaker *simp_all*. This method becomes more cumbersome if there are more than two variables, whereas the first method is still simple. For example,

```
theorem
  assumes "~w --> m"
  and "~a --> i"
  and "e --> (~i & ~m)"
  and "(a & w) --> p"
  and "~p"
shows "~e"
  using assms by auto


theorem
  assumes "~w --> m"
  and "~a --> i"
  and "e --> (~i & ~m)"
  and "(a & w) --> p"
  and "~p"
shows "~e"
  using assms by
    (case_tac a, case_tac e, case_tac i,
      case_tac m, case_tac p, case_tac w, simp_all)
```

Which one do you prefer? Six *case_tac* commands, or just concise *auto*?

Because this theorem is longer and is assumption-heavy, the *assumes* and *shows* commands are used to enhance readability. The first fact assumed is preceded by the keyword *assumes* and assumed facts are separated by the *and* keyword. The first theorem based on the assumptions is preceded by the

shows keyword and the resultant theorems are separated by the *and* keyword. Unless proven in one line, the *proof* keyword is necessary to begin the structured proof.

In Isabelle $\longrightarrow$, is represented using $--$ >, which will become $\longrightarrow$ if typed. $--$ > is used for portability. Similarly, ˜and ˜= are used to represent $\neg$ and $\neq$, respectively, in Isabelle for portability purposes.

The past example comes from Gries and Schneider, where an argument against the existence of an omnibenevolent and omnipotent being is made. The premises are:

- If the being is not willing to prevent evil (w), they are malevolent (m).
- If the being is not able to prevent evil (a), they are impotent (i).
- If the being exists (e), they are not impotent and they are not malevolent.
- If the being is able and willing to prevent evil, evil would be prevented (p).
- Evil is not prevented.

The conclusion that the being does not exist follows from the premises, as shown above using Isabelle.

2.5 A Brief Digression: Apply Proofs

In Isabelle's prog-prove guide, it first introduces command-line proofs, which repeatedly use the*apply* keyword, applying methods until the proof is done. Here is an example:

```
theorem "˜w --> m==> ˜a --> i==> e --> (˜i & ˜m)==> (a & w) --> p==> ˜p==> ˜e"
 apply(case_tac a)
 apply(case_tac i)
 apply(case_tac m)
 apply(case_tac w)
 apply(simp_all)
 done
```

Where ==> is a portable notation of $\Longrightarrow$.

First, one must use the ==> utility instead of the *assumes* and *shows* format, which is less visually appealing. Second, one must use the proof state and auto update to track progress from one step to another. Third, the toolbox is way more limiting; unless it can be solved in a few steps, repeatedly using *apply* is akin to waving a magic wand and hoping it works.

Therefore, structured proofs are far superior. The *assumes* and *shows* utility works, each step is clearly written, and one can freely attempt to get from one step to the next. In more advanced proofs, one can even stash a fact for later by using the *have* command in lieu of the *hence* command in the next line.

2.6 Predicate Logic

The next step is predicate logic, where existential and universal quantifiers are used for formal verification.

The universal quantifier is:

$$\forall x| : Px$$

And the existential qualifier is:

$$\exists x| : Px$$

This section assumes background knowledge in predicate theory. A good background text is Chapters 8 and 9 of Gries and Schneider.

In Isabelle, the universal quantifier is:

$$!\text{x. P x}$$

And the existential quantifier is:

$$?\text{x. P x}$$

! and ? are used for portability. In theory files, there are abbreviations for $\forall$ the $\exists$ symbols.

There are no ranges for quantification in Isabelle. Luckily, trading theorems exist for both universal and existential quantifications. They are:

$$(\forall x|R : P) = (\forall x| : R \longrightarrow P)$$

$$(\exists x|R : P) = (\exists x| : R \wedge P)$$

Here is an example existential proof:

```
theorem "? x. x + 5 = 6"
proof -
  have "1 + 5 = 6" by simp
  thus ?thesis ..
qed
```

To prove a value exists that meets the predicate, give an example. Then, state the predicate exists by triviality.

For universal quantifiers, fix an arbitrary value and show how the value meets the predicate. Here is an example, showing no natural number's successor is 0:

```
theorem "!x. (Suc x) ~= 0"
proof
  fix x show "(Suc x) ~= 0" by simp
qed
```

Here is a low-level example proof to tie both universal and existential proofs together:

```
theorem deMorganL:
  assumes "~(! x. P x)"
  shows "? x. ~P x"
proof (rule ccontr)
  assume 0: "~(? x. ~P x)"
  have "!x. P x"
  proof
    fix x
    show "P x"
    proof (rule ccontr)
      assume "~P x"
      hence "? x. ~P x" ..
      thus False using 0 by contradiction
    qed
  qed
  thus False using assms by contradiction
qed

theorem deMorganR:
  assumes "? x. ~P x"
  shows "~(! x. P x)"
proof
  assume 0: "! x. P x"
  from assms(1) obtain x where 1: "~P x" ..
  have 2: "P x" using 0 ..
  show False using 1 2 by contradiction
qed
```

```
theorem deMorgan:
  "(~(! x. P x)) = (? x. ~P x)"
proof
  assume "~(! x. P x)"
  thus "(? x. ~P x)" by (rule deMorganL)
next
  assume "(? x. ~P x)"
  thus "(~(! x. P x))" by (rule deMorganR)
qed
```

Where *ccontr* is the proof technique for classical contradiction ($\neg P \longrightarrow False \Longrightarrow P$) and contradiction is the technique which derives *False* from the facts of P and $\neg P$. The rule also shows any predicate Q when P and $\neg P$ are both facts.

Here is a proof of the Drinker's principle. The principle says that there is someone in a pub where if they are drinking, then everyone is drinking. Here it is:

```
theorem Drinker: "? x. (P x --> (! x. P x))"
proof (cases "! x. P x")
  case 0: True
  fix x
  have "P x --> (! x. P x)" using 0 ..
  thus ?thesis ..
next
  case False
  hence "? x. ~ P x" by (rule deMorganL)
  then obtain x where 0: "~P x" ..
  have "P x --> (! x. P x)"
  proof
    assume 1: "P x"
    show "(! x. P x)"
    proof -
      from 0 1 show ?thesis by contradiction
    qed
  qed
  thus ?thesis ..
qed
```

Where *cases "P x"* creates two subgoals from one $Q\ x$ here:

- $P\ x \Longrightarrow Q\ x$
- $\neg\ P\ x \Longrightarrow Q\ x$

And *?thesis* is the fact that is attempted to be proven.

One may correctly object that in an empty pub, the Drinker's principle is false because False is the neutral element of $\vee$. However, Isabelle assumes a non-empty universe for every datatype. When creating a new datatype, one must prove it has at least one member. This assumption can be formally stated as ontological commitment, where:

$$(\forall x | R : P) \longrightarrow (\exists x | R : P)$$

Or in Isabelle:

```
theorem
  assumes "! x. P x"
  shows "? x. P x"
proof -
```

```
  fix x
  have "P x" using assms ..
  thus ?thesis ..
qed
```

Where *proof* - tells the client to reject the default simplification of the subgoal.

Or more concisely in Isabelle:

```
theorem
  assumes "! x. P x"
  shows "? x. P x"
  using assms by blast
```

Also, de Morgan's and the drinker's principles can be proved concisely:

```
theorem deMorganAlt:
  "(~(! x. P x)) = (? x. ~ P x)"
  by simp

theorem DrinkerAlt:
  "? x. (P x --> (! x. P x))"
  by simp
```

2.7 Sets

The next background topic is sets. This sub-section assumes background in set theory. A good background text is Chapters 8 and 9 of Gries and Schneider.

2.7.1 Constants in Isabelle

To define a constant in Isabelle, use the following format:

abbreviation *name*[:: *type*]
  where "*name* == *value*"

For example, the constant M as 1000 would be:

```
abbreviation M::nat where
    "M == 1000"
```

define can also be used in lieu of *abbreviation*. If so, replace equivalence with equality in the constant assignment. The Isabelle client immediately replaces the abbreviation with its value, simplifying proofs. However, definitions must be explicitly invoked in proofs. For the definition of D, *D_def* must be invoked for proofs involving D to work. Therefore, abbreviations are preferable to definitions.

2.7.2 Set Syntax

Sets in Isabelle are formed with a dummy and a predicate using that dummy. The format used is:
$\{x.\ P\ x\ \}$, where x is the dummy and $P\ x$ is a predicate involving the dummy.

For example, the set of all natural numbers n where $n < 5$ is:

```
{x. x < 5}
```

Membership in a set is typically denoted as $x \in S$. In Isabelle, the typical denotation is used. However, I will replace $\in$ with *:* for portability. For example,

```
x : S
```

Non-membership is typically denoted as $\notin$. For portability, the ASCII representation $\sim$: will be used instead.

In Isabelle, $\emptyset$ is denoted as *bot*. The universe (typically U) is denoted as *UNIV*. For portability, $\cup$ is *Un* and $\cap$ is *Int*. In order to prove that sets S and T are equal, prove $S \subseteq T$ and $T \subseteq S$. This is similar to proving $p = q$ by proving $p \longrightarrow q$ and $q \longrightarrow p$.

There is no ASCII symbol for $\subseteq$. Therefore, type this code into your session to get *Sub* to be an abbreviation for a subset. All the symbols are available in the Isabelle client, on the bottom sidebar, in the Symbols utility (see Section 2.2).

abbreviation subset:: "'a set ⇒ 'a set ⇒ bool" (**infix** "Sub" 45) **where**
"subset A B ≡ A ⊆ B"

2.7.3 Set Compliment

Set compliment is not predefined in Isabelle. If you want it, first define an abbreviation:

```
abbreviation setcomp:: "'a set => 'a set" where
    "setcomp S == UNIV - S"
```

Then, you would overload the ~operator:

```
notation setcomp ("~")
```

To avoid the complexity of overloading, you could remove the previous definition of the ~operator:

```
no_notation Not  ("~ _" [40] 40)
```

Then, you would use the Isabelle symbol ¬ for negation.

While a trick like this will be used in section 3, I won't use it here to ensure easy portability while you are learning the syntax of Isabelle. Therefore, I will use the expression $UNIV - S$ for $\sim S$

2.7.4 Examples

Here are a few examples:

This example shows $S - T = (S \cap \sim T)$

```
theorem "(S - T) = (S Int (UNIV - T))"
proof
  show "(S - T) Sub (S Int (UNIV - T))"
  proof
    fix x
    assume "x : (S - T)"
    hence "x : S & x ~: T" by simp
    hence "x : S & x : (UNIV - T)" by simp
    thus "x : (S Int (UNIV - T))" by simp
  qed
next
  show "S Int (UNIV - T) Sub S - T"
  proof
    fix x
    assume "x : (S Int (UNIV - T))"
    hence "x : S & x : (UNIV - T)" by simp
    hence "x : S & x ~: T" by simp
    thus "x : (S - T)" by simp
  qed
qed
```

Here are two proofs, showing that complex set theorems can be easily proven by predefined rules. A good rule of thumb in Isabelle is not to reinvent the wheel - use automated provers when you can.

These predefined rules often are outputted in the proof state when the predefined rule can solve your goal. However, when many predefined rules are outputted, don't use them, just use the appropriate prover.

```
theorem "(S Int T = bot) = (!x. x : S --> x ~: T)"
  by (rule Set.disjoint_iff)

theorem "S - T Int U = (S - T) Un (S - U)"
  by (rule Set.Diff_Int)
```

Here is a final one, proving the transitivity of $\subseteq$:

```
theorem
  assumes "S Sub T" and " T Sub U"
  shows "S Sub U"
proof
  fix x
  assume "x : S"
  hence "x : T" using assms(1) by auto
  thus "x : U" using assms(2) by auto
qed
```

2.8 Functions and Notation

In the past sub-section, non-recursive functions and notation were used without explanation. In this sub-section, these concepts will be explained, along with recursive functions.

2.8.1 Non-recursive Functions

Non-recursive functions are defined using the *abbreviation* or *definition* keyword. *abbreviation* will be used here because it is generally superior to *definition*, for the same rationale describe in Section 2.7.1. The syntax is:

abbreviation *name*::"*param1Type* => *param2Type* => ... => *returnType*" **where**
"*name* == *definition*

A trivial example is doubling a natural number:

```
abbreviation double:: "nat => nat" where
  "double n == n + n"
```

This works well for any non-recursive function. Here is an example, which classifies natural numbers into the linguistic classifications of none (0), one (1), and many (≥ 2).

```
abbreviation none::nat where "none == 0"
abbreviation one::nat where "one == 1"
abbreviation many::nat where "many == 2"

abbreviation linguistic_classify:: "nat => nat" where
  "linguistic_classify n == (if n = 0 then none else
    (if n = 1 then one else many))"
```

The fact that *abbreviation* is used both for constants and functions may be confusing. To resolve the confusion, think of constants as functions without parameters.

2.8.2 Notation

Notation is used to define an alternate formation of a constant or a prefix operator. Numerals (e.g. 0, 2, -1, 1.1) are not constants; thus, notification cannot be defined for them. Use abbreviations instead.

The format for notation is:
notation *constant_or_prefix_op* ("*alternate_notation*").

After that line, you may use *alternate_notation* in lieu of *constant_or_prefix_op*.

A common example is that in Isabelle, ∅ is *bot*. Despite having an ∅ symbol, it is not predefined to mean its typical meaning. You may define it yourself using this line:

notation bot ("∅")

Symbols may not be used in names. However, a symbol can be assigned to a constant or a prefix operator through the *notation* keyword.

If you want to remove existing notation, use the following format:

no_notation *exact definition of the notation you want to remove.*
For example, if ∅ should relinquish its typical meaning, use this line:

no_notation bot ("∅")

While complex definitions for notation are unnecessary, they are sometimes used. For example, ˜was defined using this line:

notation Not ("˜_" [40] 40)

Thus, when removing it, use this line:

no_notation Not ("˜_" [40] 40)

2.8.3 Recursive Functions

While recursive functions are not used in guide, they deserve a brief mention. For example, the definition of addition using Peano arithmetic.

```
fun add::"nat => nat => nat" where
    "add 0 b = b" |
    "add (Suc a) b = Suc (add a b)"
```

The syntax is identical to abbreviations except:

- *fun* is used instead of *abbreviation*
- Separate cases are separated by a `|`.
- = is used in lieu of ≡.

A recursive function must have at least one base case, and all recursive cases must eventually terminate with a base case. Even though *fun* exists for recursive functions, a non-recursive function could be defined with *fun*. This however is bad form; so non-recursive functions will be defined with *abbreviation*.

2.8.4 Infix notation

If a function has two parameters, infix notation may be used. For example, if one wanted to add the + notation to the *add* function, one would write the first line of the add function as follows:

```
fun add::"nat => nat => nat" (infix "+" 45) where
    "add 0 b = b" |
    "add (Suc a) b = Suc (add a b)"
```

The syntax is:
(**infix** "*notation*" *max_size*)
The components are:

- **infix**: The keyword. If the direction of association is critical (e.g. exponents) use **infixl** or **infixr** instead, to indicate the notation associates to the left or right, respectively.
- *symbol*: The symbol for the infix notation. In the above example, the symbol being "+" causes $a + b$ to be equivalent to *add a b*. If the symbol was #, then $a \# b$ would be equivalent to *add a b* instead. The same principle applies for any binary function and any symbol.
- *max_size*: A natural number indicating the maximum length of the operands, in bytes. Just ensure the number is sufficiently large to avoid any errors from the imposed size restriction.

2.9 Relations

This section will explore relations, the final prerequisite to the IP theory in the next chapter. Background in the theory of relation is a pre-requisite. A good background text is Chapter 14 of Gries and Schneider.

2.9.1 Cross Product

A relation is a subset of a cross product. The cross product of sets A and B ($A \times B$) is defined as
$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$
Or, in Isabelle:

```
abbreviation cross:: "'a set => 'b set => ('a * 'b) set" where
  "cross A B == {(a, b). a : A & b : B}"
```

Where * is the Isabelle-supported ASCII representation of $\times$ used here for portability. A type beginning with an apostrophe indicates a generic type. The convention is to use 'a if available, otherwise 'b if it available, otherwise 'c if available, etc.

2.9.2 Composite of Two Relations

Taking the composite of two relations, the first over $A \times B$ and the second over $B \times C$ is:

$(A \times B) \cdot (B \times C) = \{(a, c) \mid (\exists b \mid: (a, b) \in (A \times B) \wedge (b, c) \in (B \times C)\}$
Or in Isabelle:

```
abbreviation comp:: "('a * 'b) set => ('b * 'c) set => ('a * 'c) set" where
"comp A B == {(a, c). ? b. (a, b) : A & (b, c) : B}"
```

In order to validate this definition, I have defined a relation F where $(a, b) \in F \equiv (a, b) \in N \wedge b = a + 1$ and a relation G where $(a, b) \in G \equiv (a, b) \in N \wedge a = b + 1$. Then, I defined a composite relation $H = F \cdot G$. Below are the definitions and the proof that $(a, b) \in H \equiv (a, b) \in N \wedge a = b$.

```
abbreviation F :: "(nat * nat) set" where
"F == {(a, b). b = Suc a}"

abbreviation G :: "(nat * nat) set" where
"G == {(a, b). a = Suc b}"

abbreviation H :: "(nat * nat) set" where
"H == comp F G"

theorem "H = {(a, b). a = b}"
  by auto
```

2.9.3 Properties of Binary Relations

Several properties of binary relations are pre-defined in Isabelle. Below are usages of the definitions for reflexivity, irreflexivity, symmetry, anti-symmetry, asymmetry, and transitivity. While all these usages are of the form that any relation with a property satisfies a certain universal quantification, the reverse would also hold.

```
theorem
  assumes "refl R"
  shows "! a. (a, a) : R"
  using assms by (simp add:reflD)

theorem
  assumes "sym R"
  shows "! a. ! b. (a, b) : R --> (b, a) : R"
  using assms by (simp add:symD)

theorem
  assumes "trans R"
  shows "! a. ! b. ! c. (a, b) : R --> (b, c) : R --> (a, c) : R"
  using assms
  by (meson transE)

theorem
  assumes "irrefl R"
  shows "! a. (a, a) ~: R"
  using assms
  by (simp add: irreflD)

theorem
  assumes "antisym R"
  shows "! a. ! b. (a, b) : R --> (b, a) : R --> (a = b)"
  using assms by (simp add:antisymD)

theorem
  assumes "asym R"
  shows "! a. ! b. (a, b) : R --> (b, a) ~: R"
  using assms by (simp add:asymD)
```

Two guidelines helped the generation of these examples. The first is that for plain quantifications, only one boolean variable may be quantified. Luckily, this is not a problem because:

$$(\forall a, b \mid R : P) \equiv (\forall a \mid R : (\forall b \mid R : P))$$

And;

$$(\exists a, b \mid R : P) \equiv (\exists a \mid R : (\exists b \mid R : P))$$

Second, you may struggle to prove facts in your proofs. To help, Isabelle has the *try* keyword. Insert it after the fact to direct Isabelle to use its sledgehammer to search for a proof. The alternative *try0* is faster but has less breadth. You may also use *try* and *try0* after the facts used through *using*, in case you are confident that the proof needs a fact to be used, but cannot finish a proof. In any of these examples, the try keyword could be placed after the fact each theorem *shows*, or after *assms*, the facts used to prove the shown fact.

Another property of binary relations is whet a binary relation is determinate (i.e. a function). It is defined as

$$(\forall a, b, c \mid : (a, b) \in R \longrightarrow (a, c) \in R \longrightarrow b = c)$$

Or, in Isabelle:

```
abbreviation "func":: "('a * 'b) set => bool" where
"func R == (! a. ! b. ! c. (a, b) : R --> (a, c) : R --> b = c)"
```

This definition shows that a relation equivalent to our *double* and *linguistic_classify* functions are determinate. Here is the proof:

```
theorem "func {(a, b). b = double a}"
  by simp

theorem "func {(a, b). b = linguistic_classify a}"
  by simp
```

This definition of determinism also shows the determinism of functions with multiple parameters, by recording the parameters as a single tuple. For example:

```
theorem "func {((a, b), c). c = add a b}"
  by simp
```

2.10 Note on Portability

The portable notation was used solely to aid your learning. When writing Isabelle code, use the standard mathematical notation. The required symbols available the bottom sidebar's Symbols tab and many have abbreviations, converting the abbreviation into the symbol. Some begin with a \, and some don't,

2.11 Note on Comprehensiveness

While this beginner's guide is sufficient to understand my IP theory utility, it is not comprehensive of every Isabelle capability. In addition to Aleksander Mendoza's incomplete book, check out the wealth of documentation in the Documentation tab on the left sidebar of the Isabelle client.

3 The IP Theory Utility

This section will explain the IP theory utility I developed. It is contained in three major theory files. Two of the major theory files are stand-alone and the third one has a minor theory file as an appendix. If you have access to this user guide but not to the four theory files this section elucidates, please contact me to gain access. I will just reference the files, without providing a screenshot.

3.1 IP Address Utility

The first utility is the IP address (hereinafter IP) theory file. It defines IPs as a list of four natural numbers. It defines the IP predicate, IP syntax, the IP universe, the IP compliment, IP classes, local and outside IPs, and provides several numerical demonstrations.

3.1.1 Lists

Unlike propositional logic, prepositional logic, sets, and relations, lists are only used as a proxy to represent IPs. Therefore, only a few rules about lists are required to fully exploit the IP address utility. These rules will be given below. For a more complete guide, see Chapter 2.9 of *Learn Mathematics and Computer Science with Isabelle* by Aleksander Mendoza, accessible here.

Declaring a list in Isabelle is similar to declaring an array in standard programming languages like C and Java. For example, an array the numbers, 1, 2, and 3 (in that order would be declared in standard programming languages as:

```
[1, 2, 3]
```

However in Isabelle, lists are ordered from right to left instead of the standard left to right. Thus, the declaration in Isabelle would be:

```
[3, 2, 1]
```

The reason for the departure form the standard convention is because in Isabelle, the first element is attached to the empty list, the second element is attached to first, and so on. Thus, an equivalent declaration of the list would be:

```
3 # 2 # 1 # []
```

Lastly, in standard programming languages, arrays are 0-indexed to speed up execution. However, in mathematics and Isabelle, the convention when counting up is to start at 1. Therefore, the first element in an Isabelle list has an index of 1, not 0.

3.1.2 IP Predicate

Not every list of natural numbers is a valid IP address. Therefore, the *isIP* predicate, which determines that. A list is a valid IP address if and only if the list has a length of four and all its elements are less than 256. Therefore, [1, 2, 3, 4] and [100, 200, 45, 255] are valid IPs, but [3, 4, 2], [], [900, 1, 1, 1] and [1, 2, 3, 4, 5] are not.

Technically, *isIP* is a function that takes one parameter of type *nat list* and returns a boolean.

3.1.3 IP Universe and IP Compliment

The IP universe (hereinafter U) is the set that contains every IP address and nothing else. The IP compliment of a set S is $U - S$, hereinafter $\sim S$.

3.1.4 IP Syntax

Isabelle is a typed construct, and the type of a positive number is ambigious, because it may be a natural number or an integer. To resolve this ambiguity, a special syntax is introduced for IP addresses:

$$\%octet\_1\ /octet\_2\ /octet\_3\ /host\%.$$

Where the standard syntax would be:

$$octet\_1.octet\_2.octet\_3.host$$

The standard syntax is incompatible with Isabelle's syntactical constraints. Thus, in Isabelle, 54.26.100.23 will be %54/26/100/23%.

3.1.5 The *elem* function

Isabelle has no native "element at" function for lists. Therefore I created one, which conforms to the constraints identified in the Lists sub-sub-section above. For example:

```
elem [a, b] 1 = 1
```

and;

```
elem [a, b] 2 = a
```

3.1.6 IP Classes

The following definitions are based on this Microsoft Learn resource.

There are three IP classes in use for end-users, Class A, Class B, and Class C.

In this section and the next one, the IP is in the form $o_1.o_2.o_3.h$. For every class, $1 \leq h \leq 254$. Below are the additional constraints for each IP class:

- Class A: $0 \leq o_1 < 127$

- Class B: $128 \leq o_1 \leq 191$
- Class C: $192 \leq o_1 \leq 223$

The theory file proves that each class is exclusive, and a list known satisfy a predicate for an IP class is an IP address.

Technically, these are functions that take a *nat list* as a parameter and return a boolean value. The functions are *classA*, *classB* and *classC*.

3.1.7 Local and Outside IPs

An IP is local to another if they only differ within the host octet. Otherwise, the former IP is outside to the latter.

The Isabelle theory proves that an IP is local or outside to another; never none and never both.

3.1.8 Demonstrations

Several theorems are proven within the IP theory file, including $I \cup \sim I = U$ and $\sim (\sim I) = I$, for any IP address I. Also, for any set S, $U \subseteq (S \cup \sim S)$.

Finally, several numerical demonstrations are included to show how the IP theory can be used.

3.2 Port Utility

This section will elucidate the internet port (hereinafter port) utility I developed. Ports are represented as a natural number n where $0 \leq n < 2^{16}$. This theory contains the port predicate, port classes, a port type classifier and numerical demonstrations.

All classification of ports is sourced from the Service Name and Transport Protocol Port Registry of the Internet Assigned Numbers Authority (IANA). The registry is availablehere.

3.2.1 Datatypes in Isabelle

There are many ways to declare new types in Isabelle. The datatype keyword in Isabelle instantiates a type similar to an enumeration in programming languages such as Java and Python. If you wanted to declare your own boolean type in Isabelle, here's how you would do it:

```
datatype boolean = True | False
```

The datatype for the three regions of time - the past, the present and the future would be:

```
datatype time_region = Past | Present | Future
```

Due to Isabelle's ontological commitment, there is no *null* value. If you want to classify a decimal string as positive, zero, or a negative number, you will need to handle the case where the "decimal string" is not a decimal string. Normally, you would return *null* or throw an error. In Isabelle, you would need a value explicitly in the *number_type* class. This example uses the value *NotNumber*

```
datatype number_type = Postive | Zero | Negative | NotNumber
```

3.2.2 Port Predicate

The *isPort* function takes in a natural number and returns a boolean representing whether the number represents a valid port.

3.2.3 Port Classes

There are three port classes:

- **System ports**: A port p is a system port if and only if $0 \leq p < 2^{10}$
- **User ports**: A port p is a user port if and only if $2^{10} \leq p < (2^{15} + 2^{14})$
- **Dynamic ports** A port p is a dynamic (or **private**) port if and only if $(2^{15} + 2^{14}) \leq p \leq 2^{16}$

These classes are represented in the *isSys*, *isUser* and *isDyn* predicates for system, user, and dynamic ports respectively. There are theorems that show that membership in one of these classes implies the member is a port, a port must be a member of one of the three classes and each class is exclusive to the others.

3.2.4 The *portType* function

First, a datatype *PortType* is defined with four instances: *Sys*, *User*, *Dyn* and *NotPort*. *NotPort* is necessary to classify invalid ports as non-ports. Otherwise, they would be classified as ports per Isabelle's ontological commitment.

The *portType* takes a natural number and returns the type of port it is, per the guidelines outlined above. If the number is $\geq 2^{16}$, the value *NotPort* is returned to indicate it is not a valid port. Otherwise, its port type is returned: *Sys* for system ports, *User* for user ports, and *Dyn* for dynamic ports.

Theorems are given to prove the functions conforms with the specification above, through showing that using the port predicates and using *portType* are equivalent.

3.2.5 Numerical Demonstrations

Numerical demonstrations in the theory are used to demonstrate both the port predicates and the *portType* function with concrete numbers.

3.3 Protocol Utility

The final sub-section in this section will discuss the internet protocol (hereinafter protocol) utility. It will elucidate the *NamedProtocols* theory, assigned and unassigned protocols, theorems and numerical demonstration.

Every classification is based on the IANA's Assigned Internet Protocol Numbers registry, available here.

Protocols are represented by a natural number n, where $0 \leq n < 256$. This is consistent with the IANA's protocol registry referenced above.

3.3.1 The NamedProtocols Theory

The *NamedProtocols* theory serves as an appendix to the protocol theory. It contains many protocol predicates, one for each specific protocol and one for each general protocol.

The predicate for each specific protocol is of the form *isProtcol*. Camel case is used so the predicate for BBN-RCC-MON is *isBbnRccMon*. The names of two protocols are amended slightly to fit Isabelle syntax. The predicate for the TP++ protocol is *isTpPP*. This is similar to how the Standard C++ website uses p to escape $+$. The predicate for the A/N (Active Networks) protocol is *isAN* because both / and - cannot be used within an Isabelle function name.

Some protocol numbers are not specific to one protocol. For example, protocol number 99 represents any private encryption scheme. The corresponding predicate is *isAnyGeneralPrivateScheme*. The other assigned numbers with general protocols are: 61, 63, 68 and 114.

The experimental protocol numbers, 253 and 254 are represented in the preddicate *isExperimental*.

3.3.2 The Protocol and Assignment Predicates

The rest of this on the Protocol theory file, which imports the NamedProtocols theory, which imports Main.

A natural number n is a protocol if $0 \leq n < 256$. This predicate is named *isProtocol*.

A protocol p is assigned if $0 \leq p \leq 145 \vee 253 \leq p < 256$ and unassigned if $146 \leq p \leq 252$. The representative predicates are *isAssigned* and *isUnassigned* respectively.

3.3.3 Theorems

The Protocol theory contains several theorems proving important facts about protocols. If a number is an assigned or unassigned protocol, it is a protocol. A protocol is either assigned or unassigned and not both. An experimental protocol or reserved protocol is an assigned protocol and a protocol.

While the rest of the named protocols do not have theorems proving they are assigned protocols and protocols, writing these theorems is trivial. Use *reserved_assigned* and *reserved_protocol* as bases, and replace the *isReserved n* assumption with the predicate representing the named protocol. The rest of the proof is identical.

3.3.4 Numerical Demonstrations

Numerical demonstrations demonstrate the theorems and defined predicates with concrete numbers. Demonstrations of all the named protocols would be to lengthy, so STP is chosen to serve as a blueprint for a numerical demonstration of any other named protocol. Nonetheless, the experimental and reserved protocols are also demonstrated. The *isProtocol*, *isAssigned* and *isUnassigned* predicates are also demonstrated. Each demonstration of a predicate shows two cases: a case where the predicate accepts a value, and a case where it rejects a different value.