Gencot Functional Properties Proof Manual

Gunnar Teege

October 11, 2022

Contents

Chapter 1

Introduction

This manual describes how to prove functional properties for a Cogent program which resulted from translating a C program using Gencot. It is not a general manual about working with Isabelle. It only uses a very restricted subset of Isabelle features. It does not presume prior knowledge about Isabelle.

The Cogent compiler generates an Isabelle representation of the compiled program, the "shallow embedding". It mainly consists of definitions in the language of higher order logic for all functions defined in the Cogent program. The goal of the functional properties proofs described in this manual is to systematically develop more abstract definitions which together form an abstract specification of the Cogent program and can be used to prove program properties which are relevant for the use of the program, such as security properties.

The Cogent compiler also generates a C program and a proof that this program is a refinement of the shallow embedding, i.e., it behaves in the same way. Thus, if combined with the refinement proof, the abstract specification can be applied to the C program and the functional properties proofs are also valid for the C program.

Chapter 2

Isabelle Basics

Isabelle is a "proof assistant" for formal mathematical proofs. It supports a notation for propositions and their proofs, it can check whether a proof is correct, and it can even help to find a proof.

2.1 Invoking Isabelle

After installation, Isabelle can be invoked interactively as an editor for entering propositions and proofs, or it can be invoked noninteractively to check a proof and generate a PDF document which displays the propositions and proofs.

2.1.1 Installation and Configuration

** todo **

2.1.2 Theories and Sessions

The propositions and proofs in Isabelle notation are usually collected in "theory files" with names of the form `name.thy`. A theory file must import at least one other theory file to build upon its content. For theories based on higher order logic ("HOL"), as it is the case for the Cogent shallow embedding, the usual starting point to import is the theory *Main*.

Several theory files can be grouped in a "session". A session is usually stored in a directory in the file system. It consists of a file named `ROOT` which contains a specification of the session, and the theory files which belong to the session.

When Isabelle loads a session it loads and checks all its theory files. Then it can generate a "heap file" for the session which contains the processed

session content. The heap file can be reloaded by Isabelle to avoid the time and effort for processing and checking the theory files.

A session always has a single parent session, with the exception of the Isabelle builtin session `Pure`. Thus, every session depends on a linear sequence of ancestor sessions which begins at `Pure`. The ancestor sessions have separate heap files. A session is always loaded together with all ancestor sessions.

Every session has a name of the form `chap/sess` where `chap` is an arbitrary "chapter name", it defaults to `Unsorted`. The session name and the name of the parent session are specified in the `ROOT` file in the session directory. When a session is loaded by Isabelle, its directory and the directories of all ancestor sessions must be known by Isabelle.

Every session may be displayed in a "session document". This is a PDF document generated by translating the content of the session theory files to LaTeX. A frame LaTeX document must be provided which includes all content generated from the theory files. The path of the frame document, whether a session document shall be generated and which theories shall be included is specified in the `ROOT` file.

The command

```
isabelle mkroot [OPTIONS] [Directory]
```

can be used to initialize the given directory (default is the current directory) as session directory. It creates an initial `ROOT` file to be populated with theory file names and other specification for the session, and it creates a simple frame LaTeX document.

2.1.3 Invocation as Editor

Isabelle is invoked for editing using the command

```
isabelle jedit [OPTIONS] [Files ...]
```

It starts an interactive editor and opens the specified theory files. If no file is specified it opens the file `Scratch.thy` in the user's home directory. If that file does not exist, it is created as an empty file.

The editor also loads (but does not open) all transitively imported theory files. If these are Isabelle standard theories it finds them automatically. If they belong to the session in the current directory it also finds them. If they belong to other sessions, the option

```
-d <directory pathname>
```

must be used to make the session directory known to Isabelle. For every used session a separate option must be specified. Also the directories of all

ancestor sessions of the session the opened files belong to must be specified using this option if they are not yet known to Isabelle.

The option

```
-l <session name>
```

can be used to specify a session to load (together with all ancestor sessions) to use imported theories from it. If a heap file exists for that session it is used, otherwise a heap file is created by loading all the session's theories.

2.1.4 Invocation for Batch Processing

Isabelle is invoked for batch processing of all theory files in one or more sessions using the command

```
isabelle build [OPTIONS] [Sessions ...]
```

It loads all theory files of the specified sessions and checks the contained proofs. It also loads all required ancestor sessions. If not know to Isabelle, the corresponding session directories must be specified using option `-d` as described in Section 2.1.3. Sessions required for other sessions are loaded from heap files if existent, otherwise the corresponding theories are loaded and a heap file is created.

If option `-b` is specified, heap files are also created for all sessions specified in the command. Option `-c` clears the specified sessions (removes their heap files) before processing them. Option `-n` omits the actual session processing, together with option `-c` it can be used to simply clear the heap files.

The specified sessions are only processed if at least one of their theory files has changed since the last processing or if the session is cleared using option `-c`. If option `-v` is specified all loaded sessions and all processed theories are listed on standard output.

If specified for a session in its `ROOT` file (see Section 2.1.5), also the session document is generated when a session is processed.

2.1.5 Invocation for Document Creation

** todo **

2.2 Isabelle Theories

Remark: This document uses a pretty printed syntax for Isabelle theories. A major difference to the actual source syntax is that underscores in names are displayed as hyphens (minus signs). So for example the keyword displayed as **type-synonym** must be written `type_synonym` in the theory source text.

2.2.1 Theory Structure

The content of a theory file has the structure

theory *name*
imports $name_1 \cdots name_n$
begin
  ...
end

where *name* is the theory name and $name_1 \cdots name_n$ are the names of the imported theories. The theory name *name* must be the same which is used for the theory file, i.e., the file name must be `name.thy`.

The theory structure is a part of the Isabelle "outer syntax" which is mainly fixed and independent from the specific theories. Other kind of syntax is embedded into the outer syntax. The main embedded syntax ist the "inner syntax" which is mainly used to denote types and terms. Content in inner syntax must always be surrounded by double quotes. Note that in the pretty printed forms in this document these quotes are omitted.

Additionally, text written in LaTeX syntax can be embedded into the outer syntax using the form **text**‹ ··· › and LaTeX sections can be created using **chapter**‹ ··· ›, **section**‹ ··· ›, **subsection**‹ ··· ›, **subsubsection**‹ ··· ›, **paragraph**‹ ··· ›, **subparagraph**‹ ··· ›. Note that the delimiters used here are not the "lower" and "greater" symbols, but the "cartouche delimiters" available in the jedit Symbols subwindow in tab "Punctuation".

It is also possible to embed inner and outer syntax in the LaTeX syntax (see the Isabelle reference manuals).

2.2.2 Types, Constants, and Functions

As usual in formal logics, the basic building blocks for propositions are terms. Terms denote arbitrary objects like numbers, sets, functions, or boolean values. Isabelle is strongly typed, so every term must have a type. However, in most situations Isabelle can derive the type of a term automatically, so that it needs not be specified explicitly. Terms and types are always denoted using the inner syntax.

Types are usually specified by type names. There are predefined type names such as *nat* and *bool*. Types can be parameterized, then the type arguments are denoted *before* the type name, such as in *nat set* which is the type of sets of natural numbers.

New type names can be declared in the form

typedecl *name*

which introduces the *name* for a new type for which the values are different from the values of all existing types (and no other information about the

values is given). Alternatively a type name can be introduced as a synonym for an existing type in the form

type-synonym *name* = *type*

such as in **type-synonym** *natset* = *nat set*.

Terms are mainly built as syntactical structures based on constants and variables. Constants are usually denoted by names, using the same namespace as type names. Whether a name denotes a constant or a type depends on its position in a term.

A constant name can be introduced by declaring it together with its type. The declaration

consts $name_1$ :: $type_1$ $\cdots$ $name_n$ :: $type_n$

declares n constant names with their types. Constant names declared in this way are "atomic", no other information is given about them.

A constant name denotes an object, which, according to its type, may also be a function of arbitrary order. Functions always have a single argument. The type of a function is written as *argtype* $\Rightarrow$ *restype*. The result type of a function may again be a function type, then it may be applied to another argument. This is used to represent functions with more than one arguments. Function types are right associative, thus a type $argtype_1 \Rightarrow argtype_2 \Rightarrow \cdots \Rightarrow argtype_n \Rightarrow restype$ represents a function which can be applied to n arguments. Function application terms for a function f and an argument a are denoted by $f\ a$, no parentheses are required around the argument. Function application terms are left associative, thus a function application to n arguments is written $f\ a_1 \cdots a_n$. Note that an application $f\ a_1 \cdots a_m$ where $m < n$ (a "partial application") is a correct term and denotes a function taking $n-m$ arguments.

Functions can be denoted by lambda terms of the form $\lambda x.term$ where x is a "variable name" which can occur in the *term*. A function to be applied to n arguments can be denoted by the lambda term $\lambda x_1 \cdots x_n.term$ where $x_1 \cdots x_n$ are distinct variable names.

2.2.3 Definitions

Constant names can also be introduced as "synonyms" for terms. There are two forms for introducing constant names in this way, definitions and abbreviations.

A definition introduces the name as a new entity in the logic, in the same way as a declaration. A definition is denoted in the form

definition *name* :: *type*
where *name* $\equiv$ *term*

Note that the "definitional equation" *name* $\equiv$ *term* is specified in inner syntax and must be delimited by quotes in the source text.

If the type of the defined name is a function type, the *term* may be a lambda term. Alternatively, the definition for a function applicable to n arguments can be written in the form

definition *name* :: *type*
where *name* $x_1 \cdots x_n \equiv$ *term*

with variable names $x_1 \cdots x_n$ which may occur in the *term*. This form is mainly equivalent to

definition *name* :: *type*
where *name* $\equiv \lambda x_1 \cdots x_n$.*term*

An abbreviation introduces the name only as a syntactical item which is not known by the internal logics system, upon input it is automatically expanded, and upon output it is used whenever a term matches its specification. An abbreviation is denoted in a similar form as a definition:

abbreviation *name* :: *type*
where *name* $\equiv$ *term*

The alternative form for functions is also available.

2.2.4 Overloading

A declared constant name can be associated with a definition afterwards by overloading. Overloading depends on the type. Therefore, if a constant name has a parameterized type, different definitions can be associated for different type instantiations.

A declared constant name *name* is associated with n definitions by the following overloading specification:

overloading
  $name_1 \equiv name$
    $\cdots$
  $name_n \equiv name$
begin
  definition $name_1$ :: $type_1$ **where** $\cdots$
    $\cdots$
  definition $name_n$ :: $type_n$ **where** $\cdots$
end

where all $type_i$ must be instantiations of the type declared for *name*.

There is also a form of overloading which is only performed on the syntactic level, like abbreviations. To use it, the theory *HOL−Library.Adhoc-Overloading* must be imported by the surrounding theory:

imports *HOL−Library.Adhoc-Overloading*

Then constant *name* can be associated with n terms of different type instantiations by

adhoc-overloading *name* $term_1 \cdots term_n$

Several names can be overloaded in a common specification:

adhoc-overloading $name_1 \cdots$ **and** $\cdots$ **and** $name_k \cdots$

2.2.5 Statements

A statement specifies a proposition together with a proof, that the proposition is true. A simple form of a statement is

theorem *prop proof*

where *prop* is a proposition in inner syntax and *proof* is a proof as described in Section 2.3. The keyword **theorem** can be replaced by one of the keywords **lemma**, **corollary**, **proposition** to give a hint about the use of the statement to the reader.

Statements are often used in proofs of other statements. For this purpose they can be named so that they can be referenced by name. A named statement is specified in the form

theorem *name*: *prop proof*

It is also possible to introduce named collections of statements. A simple way to introduce such a named collection is

lemmas $name = name_1 \cdots name_n$

where $name_1 \cdots name_n$ are names of existing statements or statement collections.

Alternatively a "dynamic fact" can be declared by

named-theorems *name*

It can be used as a "bucket" where statements can be added afterwards by specifying the bucket name in the statement:

theorem [*name*]: *prop proof*

or with also specifying a statement name $name_s$ by

theorem $name_s$[*name*]: *prop proof*

2.2.6 Propositions

A proposition is specified by a statement or may occur in other contexts. In its simplest form it is a single term of type *bool*, written in inner syntax, such as

$6 * 7 = 42$

More complex propositions can express, e.g., "derivation rules" used to derive propositions from other propositions. Complex propositions are denoted using a "meta logic language". It is still written in inner syntax but uses a small set of metalogic operators common to all possible object logics in Isabelle.

Derivation rules consist of assumptions and a conclusion. They can be written using the metalogic operator $\Longrightarrow$ in the form

$$A_1 \Longrightarrow \cdots \Longrightarrow A_n \Longrightarrow C$$

where the $A_1 \cdots A_n$ are the assumptions and C is the conclusion, all of them are propositions. A derivation rule states that if the assumptions are known to be true, the conclusion can be derived to be true as well. So it can be viewed as a "meta implication" with a similar meaning as a boolean implication, but a different use.

An alternative, Isabelle specific syntax for derivation rules is

$$[\![A_1; \cdots; A_n]\!] \Longrightarrow C$$

which is often considered as more readable, because it better separates the assumptions from the conclusion. In the Jedit editor it may be necessary to switch to this form by setting `Print Mode` to `brackets` in `Plugin Options` for `Isabelle General`.

Note that in the literature a derivation rule $[\![P;\ Q]\!] \Longrightarrow P \wedge Q$ is often denoted in the form

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

A proposition may contain universally bound variables, using the metalogic quantifier $\bigwedge$ in the form

$$\bigwedge x_1 \cdots x_n.\ P$$

where the $x_1 \cdots x_n$ may occur free in the proposition P. If a proposition contains free variables they are implicitly bound in this way.

2.2.7 Locales

There are cases where theory content such as definitions and statements occur which has similar structure but differs in some types or terms. Then it is useful to define a "template" and instantiate it several times. This can be done in Isabelle using a "locale".

A locale can be seen as a parameterized theory fragment, where the parameters are terms. A locale with n parameters is defined by

locale *name* =
  fixes $name_1$ :: $type_1$ **and** $\cdots$ **and** $name_n$ :: $type_n$
begin
  $\cdots$
end

where for each parameter its name and its type is specified. The content between **begin** and **end** may consist of definitions and statements which may use the parameter names like constant names. Content may also be added to an existing locale in the form

context *name*
begin
  $\cdots$
end

Therefore the **begin** $\cdots$ **end** block can also be omitted in the locale definition and the locale can be filled later.

An instance of the parameterized theory fragment is created by "interpreting" the locale in the form

interpretation *name* $term_1 \cdots term_n$ **.**

where $term_1 \cdots term_n$ are the terms to be substituted for the locale parameters, their types must match the parameter types, i.e., must be instantiations of them. The final dot in the interpretation is a rudimentary proof. An actual proof is needed, if the locale definition specifies additional properties for the parameters.

Additional properties for locale parameters can be specified in the form

locale *name* =
  fixes $name_1$ :: $type_1$ **and** $\cdots$ **and** $name_n$ :: $type_n$
  assumes $namp_1$: $prop_1$ **and** $\cdots$ **and** $namp_m$: $prop_m$
begin
  $\cdots$
end

where $namp_1 \cdots namp_m$ are names declared for the properties $prop_1 \cdots prop_m$. They can be used to reference the properties in proofs in the locale

content. If the locale is interpreted, all the properties must be proved with the actual terms substituted for the parameters. Therefore the more general form of an interpretation is

interpretation *name* $term_1 \cdots term_n$ *proof*

A locale can extend one or more other locales using the form

locale *name* = $name_1 + \cdots + name_n$ +
  fixes $\cdots$
  assumes $\cdots$
begin
  $\cdots$
end

where $name_1 \cdots name_n$ are the names of the extended locales. Their parameters become parameters of the defined locale, inserted before the parameters declared by the **fixes** $\cdots$ clause.

2.3 Isabelle Proofs

Whenever a proposition occurs somewhere in an Isabelle theory it usually must be proved immediately by specifying a proof for it. A proof may consist of several steps, its structure is part of the outer syntax.

2.3.1 Goals

The proof steps incrementally prove parts of the proposition. The remaining parts which are still to be proved are called "subgoals" of the proof. Goals have the same form as propositions. When a proof for a proposition starts, this proposition is the only subgoal. If there are no more subgoals, the proof is finished.

The set of subgoals is called the "goal state" of the proof. In the Jedit editor the proof state is displayed in a separate window, according to the cursor position in the proof text.

2.3.2 Proof Scripts

One form of an Isabelle proof is a "proof script". It consists of a linear sequence of steps, each step applies a "proof method" to one or more goals in the goal state. Depending on the goal state, a proof method may be applicable or not. If it is not applicable the step is marked as an error and is omitted from the proof script. If it is applicable it may modify goals, solve goals (and remove them from the goal state), or create new goals.

A step has the form

apply *method*

where *method* is an expression denoting the proof method applied by the step. A proof method can be elementary or complex. An elementary proof method is denoted by a method name, optionally followed by arguments. A complex method has one of the following forms:

- m_1, ..., m_n : a sequence of methods which are applied in their order,
- m_1; ...; m_n : a sequence of methods where each is applied to the goals created by the previous method,
- $m_1|$ ...$|$ m_n : a sequence of methods where only the first applicable method is applied,
- $m[n]$: the method m is applied to the first n goals,
- $m?$: the method m is applied if it is applicable,
- $m+$: the method m is applied once and then repeated as long as it is applicable.

Parentheses are used to structure and nest complex methods.
The last step of a proof script must result in an empty goal state. After it the proof script is terminated by a step of the form

done

Alternatively a proof script may be terminated in an arbitrary goal state (even at the beginning) by a step of the form

sorry

This can be used to include statements which are not yet proved in a theory. However, such statements cannot be referenced and used in subsequent proofs.
The last step **apply** *method* together with the terminating **done** can be combined to

by *method*

where this form additionally solves remaining goals which unify with assumptions and applies a simple form of backtracking if the *method* can be applied in different ways to the goal state.

2.3.3 Backward Reasoning

- Reasoning step
- conclusion unification
- rule with one assumption
- rule with several assumptions
- rule method
- rule_tac method
- introduction rules
- intro method

2.3.4 Forward Reasoning

- assumption unification
- assumption selection
- rotate method
- drule method
- drule_tac method
- destruction rules
- erule method
- erule_tac method
- elimination rules

2.4 Isabelle HOL

2.4.1 Meta Level Operators

- rewrite rules
- $\equiv$
- $\Longrightarrow$

2.4.2 Logical Operators

- $\wedge$, $\vee$, $\neg$, $\longrightarrow$
- $=$, $\neq$, $\longleftrightarrow$
- $\forall$, $\exists$

2.4.3 Logic Rules

- conjI, conjE, disjI1, disjI2, disjE, impI, mp
- contrapos_*
- iffI, iffE, iffD1, iffD2
- allI, allE, exI, exE

2.4.4 Equational Reasoning

- Equations, conditional equations
- Substitution
- subst method
- symmetric attribut

2.4.5 The Simplifier

- simpset, simp attribute
- simp method
- add:, only:, del:
- simp_all method
- recursive simplification for conditions
- simp trace
- debugging with subst

Chapter 3

Abstractions for struct Types

Chapter 4

Abstractions for array Types