Gencot Functional Properties Proof Manual

Gunnar Teege

September 23, 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 LaTeXdocument 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 files names and other specification for the session, and it creates a simple frame LaTeXdocument.

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 file 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 **??**), also the session document is generated when a session is processed.

2.2 Isabelle Theories

2.2.1 Theory Structure

The content of a theory file has the structure

```
theory <name>
imports <name1> ... <namen>
begin
  ...
end
```

where `<name>` is the theory name and `<name1> ... <namen>` 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.

Additionally, text written in LaTeXsyntax can be embedded into the outer syntax using the form

```
text\isa{{\isachardot}{\isachardot}{\isachardot}}
```

and LaTeXsections can be created using

```
chapter\isa{{\isachardot}{\isachardot}{\isachardot}}
section\isa{{\isachardot}{\isachardot}{\isachardot}}
subsection\isa{{\isachardot}{\isachardot}{\isachardot}}
subsubsection\isa{{\isachardot}{\isachardot}{\isachardot}}
paragraph\isa{{\isachardot}{\isachardot}{\isachardot}}
subparagraph\isa{{\isachardot}{\isachardot}{\isachardot}}
```

Note that the delimiters used here are not the "lower" and "greater" symbols, but the "cartouche delimiters" available in the jedit Symols subwindow in tab "Punctuation".

It is also possible to embed inner syntax In the LaTeXsyntax.

2.2.2 Types and Constants

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. 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 denotes an object, which may also be a function of arbitrary order. Functions always have a single argument. The type of a function is written as *argtype* ⇒ *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 $\mathit{argtype}_1 \Rightarrow \mathit{argtype}_2 \Rightarrow ... \Rightarrow \mathit{argtype}_n \Rightarrow$ *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 $f\ a_1\ ...\ a_n$. Note that an application $f\ a_1\ ...\ a_m$ where $m < n$ (a "partial application") is a correct term and denotes a function taking $n-m$ arguments.

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

```
consts <name1> :: <type1> ... <namen> :: <typen>
```

declares `n` constant names with their types.

2.2.3 Definitions

- definitions
- abbreviations

2.2.4 Overloading

- overloading
- adhoc overloading

2.2.5 Statements

- theorem, lemma, ...
- statement names
- named statement collections (lemmas, buckets)

2.2.6 Locales

- locale as parameterized theory
- locale parameters
- locale content
- locale interpretation

2.3 Isabelle Proofs

2.3.1 Goals

- prop syntax
- assumptions
- conclusion
- view as rewrite rule
- subgoals (proof state)

2.3.2 Proof Scripts

- method syntax
- apply
- apply scripts
- done, by
- sorry

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