

# Isabelle/jEdit as IDE for domain-specific formal languages and informal text documents

Makarius Wenzel  
<https://sketis.net>

Isabelle/jEdit is the main application of the Prover IDE (PIDE) framework and the default user-interface of Isabelle, but it is not limited to theorem proving. This paper explores possibilities to use it as a general IDE for formal languages that are defined in user-space, and embedded into informal text documents. It covers overall document structure with auxiliary files and document antiquotations, formal text delimiters and markers for interpretation (via control symbols). The ultimate question behind this: How far can we stretch a plain text editor like jEdit in order to support semantic text processing, with support by the underlying PIDE framework?

## 1 Introduction

Isabelle is a well-known system for interactive theorem proving<sup>1</sup> and jEdit a text editor written in Java<sup>2</sup>. Isabelle/jEdit is a separate application on its own [9], based on Isabelle and jEdit: it is a particular front-end for the Prover IDE (PIDE) framework, which has emerged in the past decade [5, 6, 8, 7]. Another PIDE front-end is Isabelle/VSCoDe<sup>3</sup>, but that is still somewhat experimental and needs to catch up several years of development.

Officially, the main purpose of Isabelle/jEdit is to support the user in reading and writing *proof documents*, with continuous checking by the prover in the background. This is illustrated by a small example in fig. 1, which is the first entry in the *Documentation* panel: it uses Isabelle/HOL as logical basis and as a library of tools for specifications and proofs. In bigger applications, e.g. from *The Archive of Formal Proofs* (AFP)<sup>4</sup>, the overall document may consist of hundreds of source files, with a typical size of of 50–500 KB each.

Another example document is *this paper* itself.<sup>5</sup> Isabelle documents can be rendered with PDF- $\text{\LaTeX}$  already since 1999, and the world has seen reports, articles, books, and theses produced by Isabelle without taking much notice. Thus Isabelle could be understood as a formal version of  $\text{\LaTeX}$ , but with more explicit structure, better syntax, and substantial IDE support.

Isabelle documents may contain both formal and informal text, with section headings, bullet lists, and paragraphs of plain words. New sub-languages can be defined by incorporating suitable Isabelle/ML modules into the theory context. The default setup of Isabelle/Pure and Isabelle/HOL already provides a wealth of languages for types, terms, propositions, definitions, theorem statements, proof text (in Isar), proof methods (in Eisbach) etc.

---

<sup>1</sup>See <https://isabelle.in.tum.de> for the official release Isabelle2017 (October 2017) and [https://isabelle.sketis.net/devel/release\\_snapshot](https://isabelle.sketis.net/devel/release_snapshot) for a nightly snapshot that approximates the forthcoming release Isabelle2018 (presumably August 2018).

<sup>2</sup><http://jedit.org>

<sup>3</sup><https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2017>

<sup>4</sup><https://www.isa-afp.org>

<sup>5</sup>See <https://bitbucket.org/makarius/fide2018/src> for the Mercurial repository and note that Isabelle/jEdit can open raw file URLs directly, e.g. <https://bitbucket.org/makarius/fide2018/raw/tip/Paper.thy> or <https://bitbucket.org/makarius/fide2018/raw/tip/document/root.bib> with semantic IDE support even for BibTeX.

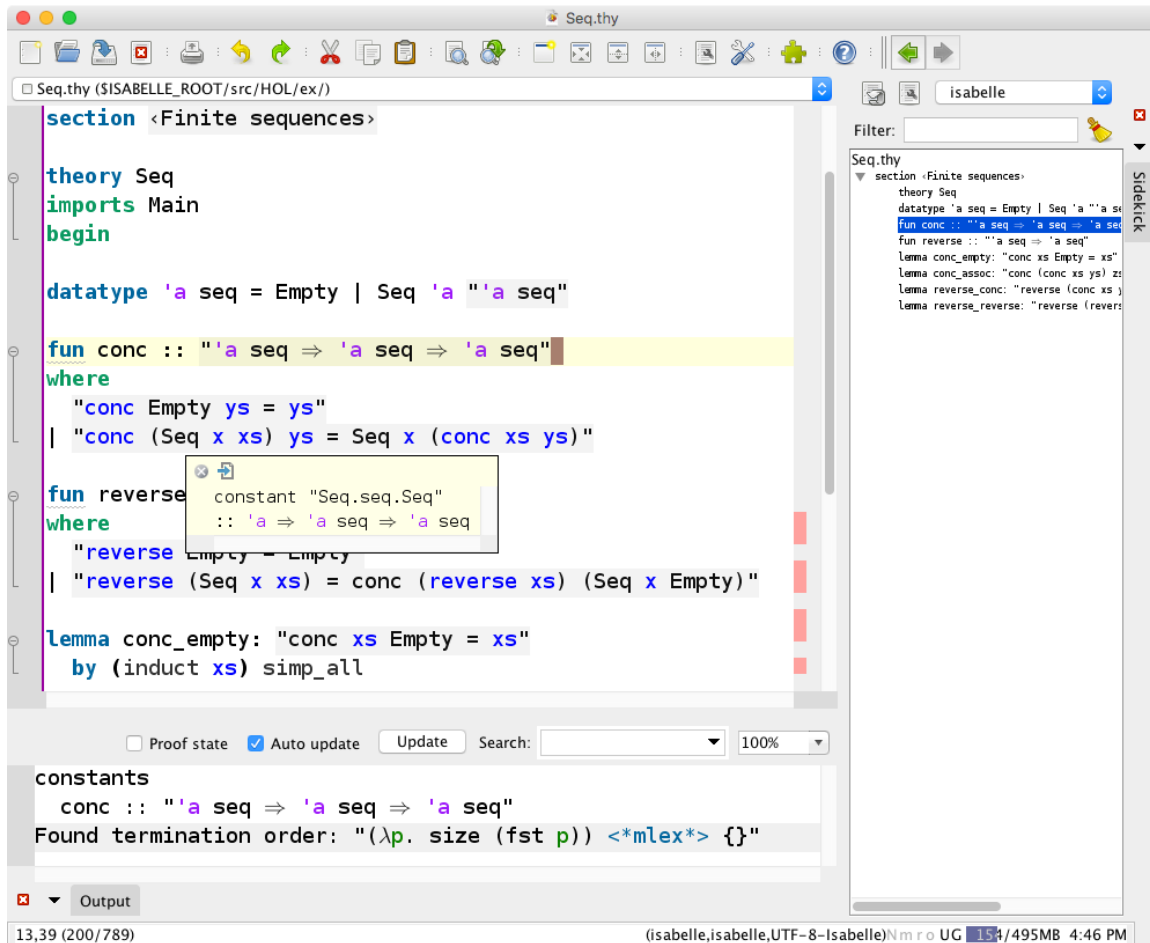


Figure 1: The Isabelle/jEdit Prover IDE with example proof document

A notable example beyond formal logic is the *rail* language for railroad syntax diagrams: e.g. see the grammar specification for the **find.theorems** command in the manual [9, §3.4.1] and its source in the IDE (fig. 2). The implementation consists of a small module in Isabelle/ML to parse rail expressions and report the recognized structure as PIDE markup<sup>6</sup>, together with a style file for L<sup>A</sup>T<sub>E</sub>X.<sup>7</sup>

Subsequently, we shall explore further uses of Isabelle/jEdit as a general-purpose IDE. §2 gives a systematic overview of PIDE document structure, as it is technically managed by Isabelle. This may provide some clues to fit other tools into the PIDE framework. §3 illustrates interactive PIDE document processing by the example of a SPARK/Ada verification environment from the Isabelle library. §4 explains document text structure, as a combination of formal and informal content. §5 presents an example of other file formats managed by the IDE: BibTeX databases and citations within PIDE documents.

<sup>6</sup>`ISABELLE_HOME/src/Pure/Tools/rail.ML`

<sup>7</sup>`ISABELLE_HOME/lib/texinputs/railsetup.sty`

```
subsection <Find theorems>

text <
  The *<Query> panel in *<Find Theorems> mode retrieves facts from the theory
  or proof context matching all of given criteria in the *<Find> text field. A
  single criterion has the following syntax:

  @ rail <
    ('-'?) ('name' ':' @syntax name | 'intro' | 'elim' | 'dest' |
           'solves' | ':' expression syntax term | @syntax term)
  > }

  See also the Isar command @command_ref find_theorems in @cite
  "isabelle-isar-ref".
>
```

Figure 2: PIDE document view with antiquotation for railroad syntax diagram

## 2 PIDE document structure

The structuring principles for PIDE documents have emerged over decades, driven by the demands of interactive proof development in Isabelle. The resulting status-quo may now be re-used for general documents with mixed languages, with some degree of scalability.

**Project directories** refer to locations in the file-system with session ROOT files (for session specifications) and ROOTS files (for lists of project sub-directories). Isabelle/jEdit and the Isabelle build tool provide option `-d` to specify additional project directories, the default is the Isabelle distribution itself. A common add-on is The Archive of Formal Proofs (AFP).

**Sessions** form an acyclic graph within a project directory. Sessions can be based on the special session `Pure`, which is the start of the bootstrap process, or on other sessions as a single *parent session* or multiple *import sessions*. The session graph restricted to the parent relation forms a tree: it corresponds to the bottom-up construction of *session images*, i.e. persistent store of the Isabelle/ML state as “dumped world”; these can be extended, but not merged. In contrast, session imports merge the name space of source files from other sessions, but the resulting session image needs to process such side-imports once again, to store them persistently.

Session names need to be globally unique wrt. the active project directories. Notable example sessions are: `Pure` (Isabelle bootstrap environment), `HOL` (main Isabelle/HOL), `HOL-Analysis` (a rather big Isabelle/HOL library with classic analysis), `Ordinary_Differential_Equations` (an AFP entry that formalizes differential equations on top of classic analysis).

Session content is specified in the session ROOT entry via **options**, **theories** (end points for the reachable import graph), and **document files** (headers, styles, graphics etc. for PDF-L<sup>A</sup>T<sub>E</sub>X output).

**Theories** form an acyclic graph within a session. The theory name is qualified according to the session name, e.g. `HOL.List`, but could be global in exceptional situations, e.g. `Pure` (instead of `Pure.Pure`),

Main (instead of HOL.Main).

A theory header is of the form **theory** *A* **imports**  $B_1 \dots B_n$  **begin** and refers to the theory base name and its *parent theories*. Imports either refer to other sessions (session-qualified theory name) or to .thy files within the session base directory (relative file name without the extension).

Semantically, a theory is a container for arbitrary *theory data* defined in Isabelle/ML, together with *extend* and *merge* operations to propagate it along the theory import graph. For example, theory *A* above begins with an extension of all data from theories  $B_1, \dots, B_n$  merged in a canonical order; its body may augment the data further. This is motivated by the contents of logical signatures, e.g. types and constants declared in various theories become available in the extension due to the *monotonicity* of the logical environment. Another example is the Isabelle/ML toplevel environment within the theory: ML types, value, signatures, structures, functors are propagated monotonically by merging the corresponding name spaces.

In other words, Isabelle theories organize “worksheets” with user-defined data, which is propagated monotonically along the foundational order of the acyclic relation of imports; there is no support for mutual theory dependencies.

**Commands** form the main body of a theory as sequence of semantic updates on theory data. Commands are defined within the theory body by updating a special data slot, but command names need to be declared beforehand as **keywords** in the theory header; this enables the Prover IDE to parse theory structure without semantic evaluation.

For example, the command **definition** is defined in theory Pure and defines a logical constant, based on an existing term from the current theory context:

```
definition foo :: (nat ⇒ nat) where (foo n ≡ 2 * n + 3)
```

As another example, the command **ML** is defined in the bootstrap process before theory Pure and allows to augment the ML environment (as data within the current theory) by ML toplevel declarations (types, values, signatures, structures etc.). This is the *meta language*, it allows to access the implementation of the logic from the outside (with references to the formal context):

```
ML (
  val foo_def = @{thm foo_def};
  val lhs = Thm.lhs_of foo_def;
  val rhs = Thm.rhs_of foo_def;
  assert (Thm.typ_of_cterm lhs = typ (nat));
)
```

Any command is free to define its own concrete syntax, within the token language of *outer syntax* of Isabelle theories, but excluding the keywords of other commands. This includes various quotations, e.g. single-quoted or double-quoted string literals and *cartouches*, which are text literals with balanced quotes that are outside of most other languages in existence: *(text)*.

Quotation allows a command to refer to a nested sub-language, and apply its own parsing and semantic processing on it, independently of the theory syntax. The semantic language context may be stored as theory data. Examples for that are again **definition** (for the embedded term language of Isabelle), and **ML** (for the ML environment that maps names to entities of the ML world).

**Auxiliary files** occur as arguments of special *load commands* within a theory. Syntactically, the argument is a file path specification, relatively to the location of the enclosing theory file. Semantically, the

argument is the text content of that file. The Prover IDE manages the physical file within the editor, and treats its current content as add-on to the load command within the theory body. The prover only sees the current content, ignorant of dependency management and edits.

For example, the command **ML\_file** (defined in theory Pure) is semantically equivalent to **ML**, but **ML\_file** uses the ML text from the given `.ML` file. This improves scalability and usability, e.g. the Prover IDE can manage the auxiliary file in its own editing mode, with different syntax rules than for the enclosing `.thy` file.

In summary, tool builders have two main possibilities to include other languages within a theory body:

1. as a regular command with literal text argument, e.g. **ML** `(val a = 1)`
2. as a load command with file path argument, e.g. **ML\_file** `(a.ML)`

All other document structure merely helps the user and the IDE to organize large projects, to manage file-system structure and name spaces of theories.

The commands **ML** and **ML\_file** have served here as canonical example for embedded languages within the Isabelle theory environment, but this language happens to be the main implementation and extension language of Isabelle itself. It is used to declare new theory data slots and to define commands operating over the content. This resembles the approach of Smalltalk: a highly dynamic and extensible environment (with IDE) that is used to bootstrap itself.

In fact, the Isabelle Prover IDE may be applied to Isabelle/Pure itself by opening the file `$ISABELLE_HOME/src/Pure/ROOT.ML` and exploring its uses of **ML\_file**. Note that `ROOT.ML` files are treated as special bootstrap theories in the Prover IDE: this allows to bootstrap pure ML projects within Isabelle/jEdit. In addition, there are some special tricks to load a fresh copy of Pure into an already running Pure session.

The special treatment of `ROOT.ML` files can be used to bootstrap other projects in Standard ML as well, independently of the Isabelle/ML environment. I have occasionally demonstrated that for the HOL4 theorem prover<sup>8</sup> and Metitarski<sup>9</sup>.

### 3 Example: Isabelle/HOL-SPARK

Isabelle/HOL-SPARK is a SPARK/Ada verification environment by Stefan Berghofer (secunet Security Networks AG). Technically it consists of a few sessions within the project directory of the official Isabelle distribution: `HOL-SPARK`, `HOL-SPARK-Examples`, `HOL-SPARK-Manual` (with full PDF [2]). As Isabelle/jEdit implicitly operates on the union of all sessions from the active project directories, we merely need to open any of the example theories and can start working after a few seconds of processing library imports: progress can be observed in the *Theories* panel. There is no need to build session images beforehand in the old-fashioned way, as described in the manual.

E.g. consider `$ISABELLE_HOME/src/HOL/SPARK/Examples/Sqrt/Sqrt.thy` (fig. 3): it uses the load command **spark.open** to read verification conditions from the output files of other SPARK tools by Altran Praxis Ltd (the results are already included in the Isabelle distribution). This augments the theory data slots defined by `HOL-SPARK`, such that other commands can refer to it in the subsequent document,

<sup>8</sup><https://sketis.net/2015/hol4-workshop-at-cade-25>

<sup>9</sup><https://sketis.net/2016/isabellepide-as-ide-for-ml-2>

```

spark_open "sqrt/isqrt"

spark_vc function_isqrt_4
proof -
  from <0 ≤ r> have "(r = 0 ∨ r = 1 ∨ r = 2) ∨ 2 < r" by auto
  then show "2 * r ≤ 2147483646"
  proof
    assume "2 < r"
    then have "0 < r" by simp
    with <2 < r> have "2 * r < r * r" by (rule mult_strict_right_mono)
    with <r * r ≤ n> and <n ≤ 2147483647> show ?thesis
      by simp
  qed auto
  then show "2 * r ≤ 2147483647" by simp
qed

spark_end

```

Figure 3: Isabelle/HOL-SPARK importing and proving a verification condition

notably **spark\_vc** to retrieve a named verification condition and commence a proof. The final **spark\_end** command ensures that all pending VCs have been solved.

It is easy to inspect the command implementations, by using control-hover-click on the keywords in the text, which leads to `$ISABELLE_HOME/src/HOL/SPARK/Tools/spark_commands.ML` and its auxiliary ML definitions to implement command parsers and theory data management. Thus we see how tool development and tool usage happens within the same universal IDE.

## 4 Document text structure

PIDE documents may contain embedded formal languages, together with *informal text*, which vaguely resembles  $\text{\LaTeX}$  source. The structure of informal text is explicitly specified, as document *markup commands* (e.g. “**section** *(text)*” or “**text** *(text)*”), and a version of *markdown notation* for item lists within the *text* body. Moreover, various sub-languages of Isabelle admit *marginal comments* alongside the formal content (e.g. “— *(text)*”). Any such informal text may again refer to formal languages via *document antiquotations*, (e.g. “@{*term t*}”).

Historically, some programming languages (e.g. Java) have introduced enhanced source comments by detecting special “doc comments” via add-on tools (e.g. javadoc), but Isabelle document text goes beyond this: it has been granted first-class status within the syntax. Note that Isabelle supports *source comments* as well (e.g. `(* source *)`), but these are omitted in document processing (like % in  $\text{\LaTeX}$ ).<sup>10</sup>

Below follows a systematic overview of Isabelle language elements for document text structure, with fine points of notation and rendering in Isabelle/jEdit.

<sup>10</sup>Users sometimes misunderstand Isabelle theory sources as “program code” and consequently use old-fashioned `(* source *)` comments to explain what they write, but readers of the final document won’t see that: it is stripped from the input and thus absent in the output.

**Markup commands** are special theory commands with a single *text* argument (double-quoted string or cartouche). Formally, this is the identity function on theory content, but the argument text is checked within the context: it may contain antiquotations. Informally, markup commands have a meaning to Isabelle document preparation, as section headings or paragraphs of plain text.

The following markup commands are predefined in the Isabelle/Pure bootstrap, and cannot be extended in user-space (which is atypical for Isabelle):

- Section headings (6 levels as in HTML): **chapter**, **section**, **subsection**, . . . , **subparagraph**. Section labels are not treated formally, but can be used via raw  $\LaTeX$ , e.g. “**section** *(Foo bar bla \label{sec:foo})*”.

Isabelle/jEdit turns section headings into a tree-view in the *SideKick* panel: this provides an informal document outline, over the structure of formal elements (e.g. definitions, theorem statements, while omitting proofs).

- Text blocks via **text** (paragraph with standard style), **txt** (paragraph with slightly smaller style), **text raw** (no change of style, e.g. for raw  $\LaTeX$ ).

Plain words within the text are marked up for spell-checking in the IDE: words that are missing from the (English) dictionary are rendered with blue underline, to say gently that something might be wrong. That markup is also used by the completion mechanism, to propose alternative words from the dictionary. For example, in fig. 2 the word “criterium” is actually wrong, but the technical term “Isar” is just unknown in the dictionary: it can be amended by suitable actions in the right-click menu of jEdit.

**Markdown lists** may occur within the body of text blocks, notably for commands **text** and **txt**. This quasi-visual format resembles official *Markdown*<sup>11</sup>, but the full complexity of that notation is avoided. Instead of ASCII art, we use the following Isabelle symbols as markers for list items:  $\langle \wedge \text{item} \rangle$  for itemize,  $\langle \wedge \text{enum} \rangle$  for enumerate,  $\langle \wedge \text{descr} \rangle$  for description. Adjacent list items with same indentation and same marker are grouped into a single list. Singleton blank lines separate paragraphs. Multiple blank lines escape from the current list hierarchy.

Isabelle/jEdit renders item markers nicely, using a special font. The depth of nested lists is emphasized by the standard color scheme for *text folds* in jEdit. Lists, items, and constituent text paragraphs are marked-up in the PIDE document model, such that control-hovering with the mouse reveals that structure to the user; see also fig. 4.

Isabelle users have occasionally reported that they like this enhanced source representation better than classic WYSIWYG text processors, because the structure of nested list items is clearly seen and easily edited.

**Formal comments** are an integral part of the document, but are logically void and removed from the resulting theory or term content. Document output supports various text styles, according to the subsequent kinds of comments.

- Marginal comment of the form “— *(text)*” (literally “ $\langle \text{comment} \rangle$  *(text)*”). The given argument is typeset as regular text, with formal antiquotations in the body. This allows to alternate formal sub-languages and informal text arbitrarily. Input works e.g. by completion of “ $\langle \text{co} \rangle$ ”.

---

<sup>11</sup><http://commonmark.org>

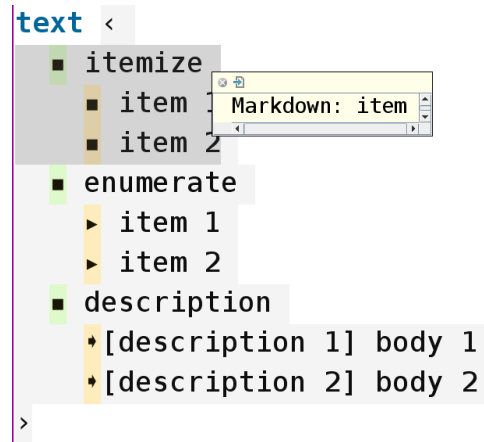


Figure 4: Markdown notation for lists in Isabelle/jEdit

- Canceled text of the form “*cancel* (*text*)” (literally “ $\backslash\langle^{\wedge}\text{cancel}\rangle\langle\text{text}\rangle$ ”). The argument is typeset as formal Isabelle source and overlaid with a “strike-through” pattern, e.g. *bold*. Input works e.g. by completion of “ $\backslash\text{ca}$ ”.
- $\text{\LaTeX}$  source of the form “*latex* (*text*)” (literally “ $\backslash\langle^{\wedge}\text{latex}\rangle\langle\text{text}\rangle$ ”). This allows to emit arbitrary (unchecked)  $\text{\LaTeX}$  source. Input works e.g. by completion of “ $\backslash\text{la}$ ”.

These formal comments work uniformly in the theory and term language, but also in Isabelle/ML and some other embedded languages. User-defined languages can easily include formal comments via standard scanner functions provided in module `Comment` of Isabelle/ML and Isabelle/Scala. A clash with existing language syntax is unlikely, due to the peculiar notation with non-ASCII Isabelle symbols.

**Document antiquotations** provide a uniform scheme to embed domain-specific formal languages within Isabelle document text (even some other sub-languages). The terminology is explained as follows: the outer formal syntax allows to *quote* informal text, which in turn allows to *antiquote* formal content. E.g. the term antiquotation “ $\text{\@}\{term\} \langle Groups.plus\ x\ y\rangle$ ” pretty-prints its argument into the generated PDF- $\text{\LaTeX}$  document using the concrete syntax from the theory context for the constant *Groups.plus*, it becomes “ $x + y$ ”. It is also possible to force printing of the original source text of the (formally checked) term, so  $\text{\@}\{term\} [source] \langle Groups.plus\ x\ y\rangle$  becomes “ $\langle Groups.plus\ x\ y\rangle$ ”.

Antiquotation syntax comes in the following variants:

**Long form:**  $\text{\@}\{name\} [options] arguments \dots$  with free-form arguments according to the parser provided for this antiquotation.

**Short forms:**

1.  $\backslash\langle^{\wedge}name\rangle\langle argument\rangle$  for  $\text{\@}\{name\} \langle argument\rangle$ , i.e. there are no options and exactly one argument that is a cartouche.
2.  $\langle argument\rangle$  for  $\text{\@}\{cartouche\} \langle argument\rangle$ , i.e. as above, but the name is the special constant “*cartouche*”. It means that one sub-language in the current context can get the privilege to be embedded without explicit marker.
3.  $\backslash\langle^{\wedge}name\rangle$  for  $\text{\@}\{name\}$  without options or arguments.



An entity of the form  $\langle \wedge name \rangle$  is called *control symbol* in Isabelle: here it acts like an operator on the subsequent text cartouche. Isabelle/jEdit provides two ways to render control symbols nicely:

1. As Unicode glyphs, according to a global symbol table of the Isabelle distribution. E.g.  $\langle \wedge file \rangle$ ,  $\langle \wedge dir \rangle$ ,  $\langle \wedge url \rangle$  are rendered as icons that are similar to common desktop file-managers.
2. As bold-italic keyword (default). For example,  $\langle \wedge term \rangle$  becomes *term*, and  $\langle \wedge prop \rangle$  becomes *prop*. This form serves as fall-back for names that lack a clear visual representation, and for arbitrary control symbols that users might invent for their own applications.

Rendering of antiquotation arguments follows the standard mechanisms of the Prover IDE, e.g. a term argument is decorated with colors for free and bound variables, identifier scopes, inferred types, hyper links etc. For unusual arguments, such as system resources, the semantic operation of the antiquotation emits markup to instruct the front-end to do the right thing, e.g. open a file in the text editor, or a directory in the file-system browser, or a URL in an external web browser (see fig. 5).

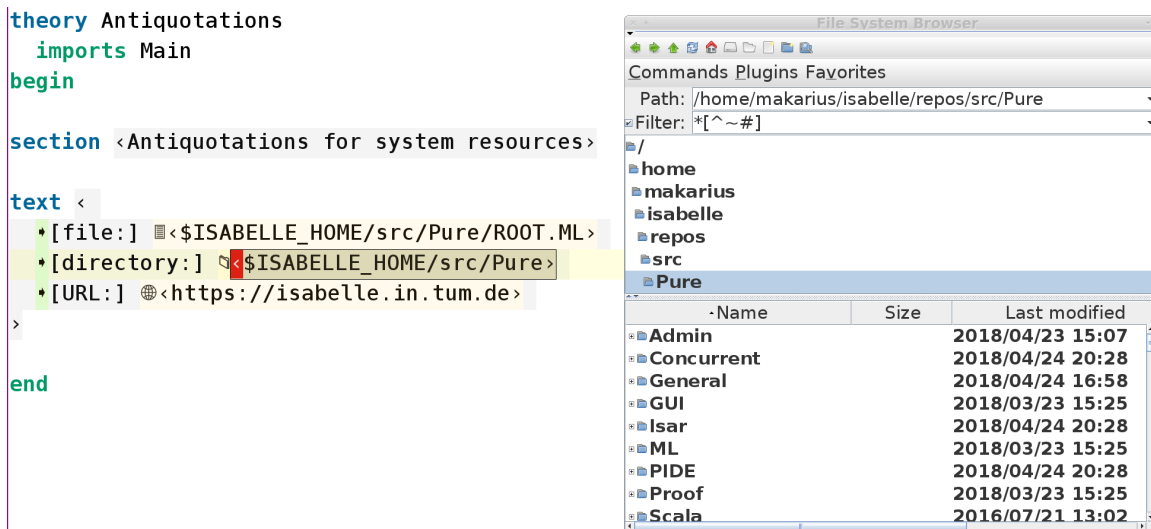


Figure 5: Document antiquotations for system resources (action on directory)

Isabelle tool developers may figure out how this is implemented, by putting an example antiquotation into the document, and performing control-hover-click onto its name to get to the definition in Isabelle/ML, and then use the ML IDE to explore that further (that also requires to open  $\$ISABELLE\_HOME/src/Pure/ROOT.ML$ ). This principle of self-application and self-sufficiency of Isabelle/jEdit makes it a truly integrated development environment.

Fig. 6 shows a somewhat artificial example for the nesting of sub-languages within a proof document.<sup>12</sup> It builds a local context with fixed variables and assumptions, followed by a **text** block with embedded  $\@ \{ lemma \}$  antiquotations: the stated results are proven in that context, but not recorded as facts. The final PDF merely prints the lemma statements as plain propositions.

Here the cursor has been placed at the binding position of the variable  $x$ : its subsequent uses are highlighted, thanks to markup provided by the sub-language parsers. It is also possible turn that scope

<sup>12</sup>See also <https://bitbucket.org/makarius/fide2018/raw/tip/Nesting.thy> and recall that Isabelle/jEdit can open URLs of text files directly.

```

section <Proof document with nesting of sub-languages>
notepad — <formal playground (block-structured context)>
begin
  fix x y z :: nat — <local entities of type typ<nat>>
  assume <x = y> and <y = z>
  text <
    Some trivial consequences of these assumptions:
    ■ symmetric results:
      ■ @lemma <y = x> by (rule sym) (tactic <resolve_tac (context thms<x = y> 1)>)}
      ■ @lemma <z = y> by (rule sym) (rule <y = z>)}
    ■ transitive result:
      ■ @lemma <x = z> by (rule trans) (rule <x = y>, rule <y = z>)}
  >
end

```

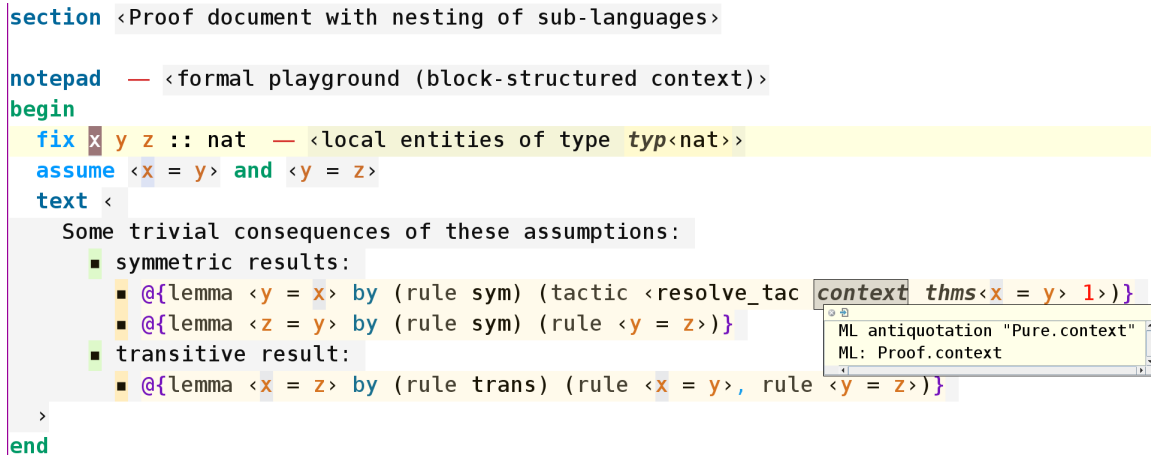


Figure 6: Proof document with nesting of sub-languages

group into a multi-selection of the editor, and thus rename the variable systematically (but without any sanity checks). The information about occurrences of  $x$  is produced by the individual parsers for nested antiquotations, which in turn refer to standard parsers of a term in a certain context: that emits PIDE markup for constants and variables encountered in the nested text.

The proof of  $@\{lemma\ (y = x)\}$  uses the proof method *tactic*, which takes an ML expression (of a suitable type) as argument. The highlighted box over *context* is the result of a control-hover GUI action; it explains that spot formally in the tooltip.

Thus users can explore the formal structure of example documents, and learn something about syntax and semantics of the many domain-specific formal languages of Isabelle, even without studying the documentation.

## 5 Example: BibTeX database and citation management

Management of bibliographies and citations is definitely outside of formal logic and functional programming, but it is practically relevant for Isabelle document authoring. Isabelle/jEdit provides IDE support as illustrated in fig. 7, with the following features:

- A jEdit syntax mode for `.bib` files: it recognizes the BibTeX syntax according to the original parser written in Pascal, re-implemented in Isabelle/Scala.<sup>13</sup> It recognizes the token language, and the names and block structure of entries.
- Support for text folds according to the block structure of BibTeX entries.
- A tree-view in the SideKick panel: each entry is associated with a name and content. The jEdit SideKick plugin allows to filter the tree-view by giving a substring for the name.
- A context-menu for BibTeX entry types (e.g. *Article*, *InProceedings*).
- Syntax highlighting for BibTeX entry fields, depending on the entry type, to distinguish required vs. optional fields.

<sup>13</sup>See also <http://ctan.org/tex-archive/biblio/bibtex/base/bibtex.web> module `@<Scan for and process a \.{.bib}> command or database entry@>`.

- HTML preview similar to  $\text{\LaTeX}$  output, using the `bib2xhtml` tool.<sup>14</sup> Users need to invoke the `jEdit` action `isabelle.preview` on an open `.bib` buffer.
- Semantic checking by the original `bibtex` tool (using a default style file). This attaches warnings and errors to the source, while the user is editing.
- Support for document antiquotations `@{cite NAME}`, with name completion according to all `.bib` files that happen to be open in the editor.
- Strict checking of cited BibTeX entries wrt. the `.bib` files in **document\_files** of the enclosing session; this only works for batch-builds, e.g. when producing the final PDF.

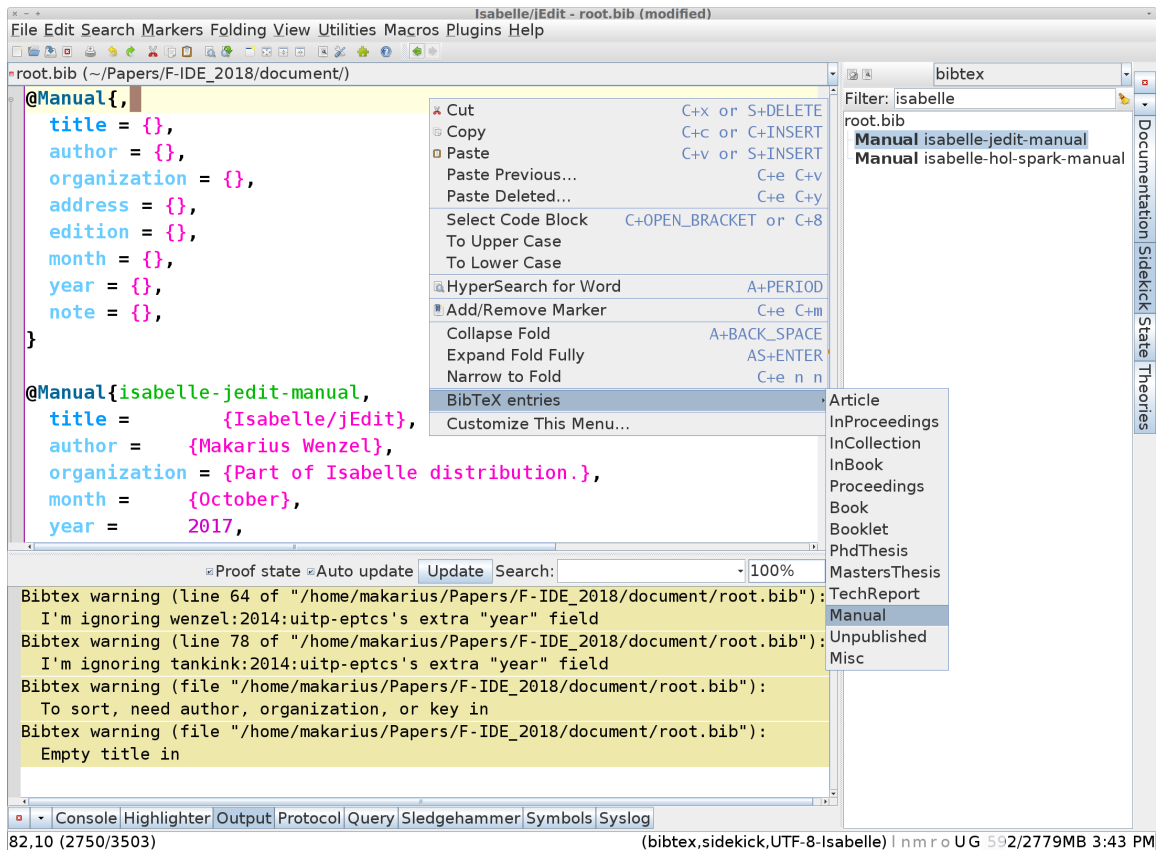


Figure 7: Semantic IDE support for BibTeX databases

The implementation of all this is quite concise. It mainly happens in Isabelle/Scala<sup>15</sup>, but semantic processing of `.bib` files and the `@{cite}` antiquotation need to happen in the stateless mathematical world of Isabelle/ML: special PIDE protocol messages are used to invoke a Scala function `String => String` and return its result in ML. Some further tricks are required to process `.bib` files faithfully and to produce messages with proper positions:

<sup>14</sup><https://github.com/dspinellis/bib2xhtml>

<sup>15</sup>See `$ISABELLE_HOME/src/Pure/Thy/bibtex.scala` and `$ISABELLE_HOME/src/Tools/jEdit/src/jedit_bibtex.scala`.

- Opening the file `foo.bib` in Isabelle/jEdit implicitly puts it into a theory context (derived from Pure) with the load command `bibtex_file` (`foo.bib`) in the body. This means that `.bib` files automatically become auxiliary files as explained in §2.
- The command `bibtex_file` is implemented in Isabelle/ML as a function `string -> unit` that emits PIDE messages according to its true meaning (here only warnings and errors). This works by asking Isabelle/Scala to invoke `Bibtex.check_database()` to do the main job.
- The Scala function `Bibtex.check_database()` tokenizes the given source text with the Isabelle BibTeX parser, which gives precise position information. Each token is put on a line of its own and the result given to the `bibtex` executable as temporary file, with options to use the `plain.bst` style file. Note that BibTeX styles are programs in a special language, to analyse and output database entries.
- The resulting BibTeX log file is scanned for warnings and errors, while observing its peculiar format (following the original implementation in Pascal). Line numbers are used as index positions for the tokens produced by the Isabelle BibTeX parser: the final results are precise positions for PIDE messages, which users will see as squiggly underlines in the source.

Thus we can pretend that Isabelle understands the meaning of BibTeX files, or that BibTeX understands the PIDE protocol. The Scala and ML sources for that are less complex than the above explanations might suggest. It demonstrates that unusual IDE applications work out with quite reasonable effort.

## 6 Conclusion

### 6.1 PIDE history and related work

Isabelle once shared the well-known Proof General Emacs interface [1] with other proof assistants, most notably Coq. In 2008, I started to think beyond the Proof General model of stepping forwards and backwards through a “script” of prover commands. This eventually led to the PIDE document model and the Isabelle/jEdit front-end, all implemented in Isabelle/Scala. The new prover front-end was sufficiently well-established in 2014 to remove the last remains of the command loop, so that Proof General is now exclusively used for Coq and has de-facto lost its generality.<sup>16</sup>

PIDE is document-oriented and timeless / stateless: it operates via functional updates on explicitly identified versions. Proof General and its many derivatives are script-oriented and stateful: one command after another on a global version. This means that various Proof General upgrades and clones (e.g. CoqIde or Coqoon<sup>17</sup>) are actually *unrelated* to PIDE. An exception is the Coq PIDE experiment from 2013/2014 [4], but it did not reach end-users so far.

Mainstream IDEs (e.g. Eclipse, Netbeans, IntelliJ IDEA) are more closely related to PIDE, but there is a conceptual difference: classic IDEs operate by static analysis of the program source and provide a separate debugger for dynamic exploration at runtime. In contrast, the PIDE model explores the syntactic structure and semantic content of the document uniformly: this works, because semantic operations are usually pure, without side-effects. Isabelle/jEdit also provides a classic debugger for Isabelle/ML, but that is strictly speaking not part of the PIDE document model: it interacts with the running ML program on a side-channel and a separate GUI panel.

<sup>16</sup><https://proofgeneral.github.io>

<sup>17</sup><https://coqoon.github.io>

Visual Studio Code (VSCode) is a remarkable open-source project by Microsoft<sup>18</sup>. Instead of a full-blown IDE, it is positioned as sophisticated a text editor, with support for “language smartness” provided by VSCode extensions (e.g. for JavaScript, TypeScript, Java). There is also a published *Language Server Protocol* to connect an external process that turns edits of sources into reports about its structure and meaning. This idea is very close to Isabelle/PIDE, and I have already connected the Isabelle/Scala process as external “language smartness provider” for Isabelle theory and ML files.

The resulting Isabelle/VSCode 1.0 for Isabelle2017 basically works,<sup>19</sup> but has a long way ahead, in order to catch up with Isabelle/jEdit 9.0 in Isabelle2017, or 10.0 in the forthcoming Isabelle2018 release. This is not just a matter to improve the Isabelle side: VSCode is quite different from jEdit in many respects. For example, VSCode imposes more explicit structure and policies on the internal text model, while jEdit is just a multi-layered paint program (for Java2D) with a plain text buffer behind it. Thus the Isabelle/jEdit “game engine” for real-time text rendering with rich markup was relatively easy to implement, by removing the default text painting layers and adding specific ones for PIDE. For VSCode, more work will be required to take it apart and access its HTML/CSS document-model directly (even worse, Microsoft explicitly restricts the access of external extensions to these important internals).

Nonetheless, the VSCode platform looks like an interesting alternative for future versions of the Isabelle Prover IDE. Some other proof assistants have already started to support it: VScoq<sup>20</sup> as a Proof General clone, and VSCode-Lean<sup>21</sup> based on a custom-made theory compilation server for Lean Prover [3].

## 6.2 Future work

Even after 10 years, the Isabelle/PIDE framework and the Isabelle/jEdit front-end are still not finished, but an open-ended enterprise. Here are possible improvements only for the topics covered in this paper:

- More explicit support for session definitions in the IDE. So far, Isabelle/jEdit uses the union of all sessions from the active project directories, and is thus able to map theory files to logical theory names. It would be nice to take specific **options** and **document files** from session ROOT files into account. The latter would also allow to relate BibTeX files to theories implicitly, and thus provide a context for  $\@{cite}$  antiquotations, without asking the user to open relevant .bib files.
- Building PDF- $\LaTeX$  documents on the spot within the IDE. So far this is still a batch-job, either via `isabelle build` on the shell, or via `Build.build()` in the Scala console of Isabelle/jEdit. Full integration into the IDE would mean that the user can select relevant theories for PDF output in a GUI panel, push a button, and see the generated (or updated) PDF quickly, lets say within 1–3s. In order to achieve this, the Isabelle document preparation system needs to be removed from the traditional Isabelle/ML session build process, and integrated into Isabelle/Scala. This reorganization would also grant access to PIDE markup information, such that the generated PDF could become more detailed (e.g. different text style for different identifiers: free variables vs. bound variables vs. constants).
- High-quality HTML document output, as a continuation of existing PDF- $\LaTeX$  output and the IDE rendering of the theory source. HTML + CSS have become sufficiently powerful for high-quality type-setting of books and journals. Imitating PDF- $\LaTeX$  documents in HTML would also spare

<sup>18</sup><https://code.visualstudio.com>

<sup>19</sup><https://marketplace.visualstudio.com/items?itemName=makarius.Isabelle2017>

<sup>20</sup><https://github.com/siegebell/vscoq>

<sup>21</sup><https://github.com/leanprover/vscode-lean>

time-consuming invocations of `pdflatex`, and thus make document output a matter of milliseconds instead of seconds. In recent years, we have seen text editors with Markdown source vs. preview in a split window: something like that could be achieved for Isabelle HTML documents as well, but with rich semantic PIDE markup. Instantaneous document preview might also help to overcome the occasional misunderstanding of users, to think of theory sources as “program code” instead of a document.

Apart from further refinement of the PIDE technology, we need to work more on the *sociology* of the project: Isabelle/jEdit is hardly known outside of the Isabelle community, and inside it users merely take it for granted. Hopefully this paper helps to apply PIDE in other applications, by Isabelle users or people building different formal tools.

## References

- [1] Aspinall, D., Lüth, C., Winterstein, D.: A framework for interactive proof. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) *Towards Mechanized Mathematical Assistants (CALCULEMUS and MKM 2007)*. LNAI, vol. 4573. Springer (2007)
- [2] Berghofer, S.: The HOL-SPARK Program Verification Environment. Part of Isabelle distribution. (October 2017), <http://isabelle.in.tum.de/website-Isabelle2017/dist/library/HOL/HOL-SPARK-Manual/document.pdf>
- [3] de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction*, Berlin, Germany, August 1-7, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9195. Springer (2015), [http://dx.doi.org/10.1007/978-3-319-21401-6\\_26](http://dx.doi.org/10.1007/978-3-319-21401-6_26)
- [4] Tankink, C.: PIDE for asynchronous interaction with Coq. In: Benzmüller, C., Paleo, B.W. (eds.) *User Interfaces for Theorem Provers (UITP 2014)*. EPTCS, vol. 167 (2014), <https://doi.org/10.4204/EPTCS.167.9>
- [5] Wenzel, M.: Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In: Coen, C.S., Aspinall, D. (eds.) *User Interfaces for Theorem Provers (UITP 2010)*. ENTCS (July 2010)
- [6] Wenzel, M.: READ-EVAL-PRINT in parallel and asynchronous proof-checking. In: Kaliszyk, C., Lüth, C. (eds.) *User Interfaces for Theorem Provers (UITP 2012)*. EPTCS, vol. 118 (2013)
- [7] Wenzel, M.: Asynchronous user interaction and tool integration in Isabelle/PIDE. In: Klein, G., Gamboa, R. (eds.) *Interactive Theorem Proving — 5th International Conference, ITP 2014*, Vienna, Austria. Lecture Notes in Computer Science, vol. 8558. Springer (2014)
- [8] Wenzel, M.: System description: Isabelle/jedit in 2014. In: Benzmüller, C., Paleo, B.W. (eds.) *User Interfaces for Theorem Provers (UITP 2014)*. EPTCS, vol. 167 (2014), <https://doi.org/10.4204/EPTCS.167.10>
- [9] Wenzel, M.: Isabelle/jEdit. Part of Isabelle distribution. (October 2017), <http://isabelle.in.tum.de/website-Isabelle2017/dist/Isabelle2017/doc/jedit.pdf>