Isabelle as Document-oriented Proof Assistant

Makarius Wenzel

Université Paris-Sud 11, LRI
Orsay, France
http://www.lri.fr/~wenzel/

Abstract. Proof assistants in the LCF tradition, such as Coq, Isabelle, and the HOL family, are notorious for old-fashioned command-line interaction with input and output of plain text. Established prover interfaces like Proof General merely add a thin layer on top of the read-eval-print loop in the background. More sophisticated mathematical editors, Web-services, Wiki-servers for mathematical content do exist, but any project that aims at fully formal proof-checking struggles with recurrent problems posed by ancient prover engines.
Taking the perspective of Isabelle, we discuss common problems and solutions that have emerged in the past few years, to fit the prover smoothly into a document-oriented environment with rich semantic annotations for formal sources. For example, this enables a conventional editor framework to present formal content provided by the prover, without having to understand logic itself (or re-implement a prover). This can be achieved with minimal changes on the editor and prover side, but the combination is able to support the usual metaphors of squiggly underline, tooltips, popups etc. that are now commonplace in browsers or IDEs.
Many of these document-oriented traits of current Isabelle are sufficiently general to be transferred to other provers. If such principles are becoming routinely available in LCF-style provers, building combined mathematical assistants should become more feasible.

1 Introduction

Isabelle has been centered around some notion of *formal proof document* ever since the introduction of the Isar proof language around 1999 (see also the overview [12]). This philosophical notion of "document" emphasizes an idea of *primary presentation* of machine-checked proofs in human-readable form. To accompany the abstract principles with concrete implementations, Isabelle/Isar provides some means to produce pretty-printed versions based on superficial observations on the structure of source text that is known to have passed through formal checking. Thus the traditional Isabelle document preparation system can present theories with good typesetting quality in PDF-LaTeX, such as the present paper. Little of the formal content that the prover has accumulated in checking the text is available in the resulting PDF, though.

In the past few years, Isabelle has started to provide more substantial access to internal prover content. This was motivated by the requirements of formal

theory repositories and the user agents to access the content (via Web-interfaces, Wikis, Prover IDEs). The general question is how aspects of sophisticated formal processing of sources can be represented externally, without front-ends having to understand logic or re-implement provers, and without provers having to implement their own front-end technology. Some of this advanced document-oriented functionality of Isabelle is already accessible to users in the experimental Isabelle/jEdit Prover IDE [13]: results of proof checking are visualized as squiggly underlines, tooltips, or hyperlinks in the source. This is only the beginning of many more possibilities, based on the same document-oriented architecture.

These new concepts of Isabelle are sufficiently general to be applicable to other proof assistants, such as Coq or variants of HOL. Although not a prover, the former command-line compiler of Poly/ML (by David Matthews) has already been integrated: Isabelle/ML sources that are embedded into the formal context benefit from content revealed by Isabelle and the static analysis of SML. This makes Isabelle/jEdit also an IDE for Poly/ML.

Cultural Side-conditions and Programming Paradigms. We need to respect some cultural side-conditions, if old-school provers shall become native participants in a larger world that typically speaks XML instead of λ-calculus.

Major interactive theorem provers are implemented in some functional programming language. For systems from North America this usually means LISP, e.g. see ACL2 [16, §8] or PVS [16, §3]. For European systems it often means a descendant of ML [6]: Standard ML for Isabelle [16, §6] and some HOL variants, OCaml for Coq [16, §4] and other HOL variants [16, §1], or Haskell for some newer implementations such as Agda [16, §7].

This cultural background impacts our problem of document-oriented theorem provers. Implementors of proof assistants routinely use higher-order datatypes and Hindley-Milner polymorphism, but the outside world might only speak of XML and event-dispatch objects for SAX parsing. The Haskell community has managed to adopt significant parts of the XML world as libraries like HaXml, but less is available in ML. Our general attitude is to retain the established prover programming culture of ML, and extend our repertoire by a few elements that help to integrate smoothly with XML-based document markup. In Isabelle, the general prover integration problem is further materialized by the following bilingual approach:

Isabelle/ML represents the pure symbolic programming environment for sophisticated logical concepts implemented in the prover. Isabelle happens to use Standard ML (notably the Poly/ML implementation), which is also embedded into the logical context [15].

Isabelle/Scala provides an external API for prover integration on the Java VM, where many useful IDE or Web frameworks already exist. Scala/JVM [7] is sufficiently flexible to support the received ML programming style. Sometimes it goes beyond that, e.g. via typed views on untyped data.

Our document-oriented concepts are reflected both in Isabelle/ML and Isabelle/Scala. The main ideas are already present in ML, and can be re-used

without subscribing to the bilingual architecture. Conceptually, our Scala library provides a simplified programming model, without exposing the full details of document-oriented prover protocols to front-ends as was done in PGIP [4].

Scala is able to express the sublime programming style of Isabelle/ML, with parameterized types, higher-order combinators, pattern matching etc. It also provides direct access to existing Java frameworks at the JVM bytecode level. Although it is hard to imagine sophisticated provers implemented in Java, one might seriously consider doing it in Scala. For us the latter is a theoretical question, since we aim at re-use of provers in ML, not re-writing from scratch in Scala. There are also some inherent limitations of the JVM that make it hard to work with large symbolic structures as effectively as in existing ML implementations.

Overview. This paper is structured as follows. §2 introduces principles for robust support of text that is annotated by markup. This makes the prover speak XML natively with minimal impact on the existing code base. §3 presents some common uses of XML content in the prover for data representation and physical rendering. §4 describes a general scheme that can reconstruct content of the conceptual prover document model in external form.

2 Text with Markup

Proof assistants are traditionally based on text for input and output. We remain faithful to this principle, but add markup information in a conservative manner. Making this work smoothly for existing provers turns out to be surprisingly difficult: it might explain retrospectively why this is not yet established prover technology. We first need to answer questions like "What is text?" and "What is markup?" precisely, and keep in mind various fine points for our implementation.

2.1 Text Encoding and Rendering

Proof assistants have a natural demand for mathematical symbols, beyond traditional ASCII art. In recent years, Unicode has become sufficiently wide-spread to be considered, although it is not as universal and uniform as ASCII. Unicode standards are still evolving, and even within a given version there are variations.

Unicode defines tables of *codepoints* (represented as small integers) that are associated with names and visual appearance of glyphs. Physical representation in memory is defined separately by *encodings*: most common are now UTF-8 and UTF-16, both available in several variants. Thus the correspondence of bytes in memory to glyphs is not immediately obvious, and moving between different platforms such as ML (UTF-8) and JVM (UTF-16) requires some care.

For example, $\mathcal{A}$ (codepoint `U+1D49C`, `MATHEMATICAL SCRIPT CAPITAL A`) corresponds to 4 bytes in UTF-8 (due to its regular multi-byte encoding), and 4 different bytes in UTF-16 (due to "surrogate characters" that are used to work

around limitations of classic 16 bit Unicode). Some intermediate program module that is unaware of UTF-16 surrogates might recode the two wide characters behind $\mathcal{A}$ separately and arrive at yet another (wrong) byte sequence.

This illustrates some subtleties that need to be taken into account for robust representation of mathematical source text inside our provers. Even though the "supplementary codepoints" outside the 16 bit range provide interesting mathematical symbols and mathematical alphabets in the spirit of TEX math mode (variations of latin, greek, calligraphic, bold, italics), we need to be careful about depending on such corner cases of the Unicode standards.

Further uncertainty is caused by semantic alternatives in the collection of codepoints. For example, the visual appearance of ϕ (`\phi`) vs. φ (`\varphi`) in TEX is defined by Knuth's Metafonts, but Unicode symbol fonts often disagree. The official entry[1] for codepoint `U+03C6` says "the ordinary Greek letter, showing considerable glyph variation", and refers to the alternative codepoint `U+03D5` among others. Taking mathematical typesetting seriously, we should probably use codepoint `U+1D711 MATHEMATICAL ITALIC SMALL PHI` to imitate math italics in TEX, but the front-end would need a font that implements that glyph.[2]

Problems:

- Robust encoding of text that is suitable for long-term storage (avoiding recoding every few years to accommodate new Unicode standards).
- Robust rendering of symbols, potentially depending on the capabilities of the front-end encoding (e.g. 16 bit limit) and fonts (e.g. missing glyphs).

Solutions: We propose variants (a), (b), (c) as follows.

(a) Trusting that UTF-8 actually manages to represent Unicode text adequately in the next decades, prover sources are directly encoded using this particular standard. Semantic dilemmas in the official Unicode tables are resolved by our own internal standardization, judging glyph variants wrt. availability in important font families (such as DejaVu or STIX) and selecting codes for φ etc. once and for all.

(b) We are conservative about ASCII as main physical representation of text, but add explicitly named entities in the spirit of TEX. Already since 1998, Isabelle has supported the notation `\<`*name*`>` for that purpose, where *name* follows usual alphabetic identifier syntax. This provides an *infinite* collection of non-ASCII symbols that is *independent* of encoding.
Extra tables define the rendering of a finite subset of Isabelle symbols. For example, `\<forall>` can be mapped to `\forall` in TEX and `U+2200` in Unicode. Such tables can be fine-tuned over time to adjust to coming trends in

[1] http://www.unicode.org/charts/PDF/U0370.pdf

[2] After many years of delay, the STIX project http://www.stixfonts.org has managed to release version 1.0 of its free mathematical fonts in 2010. So fairly complete coverage of mathematical glyphs may be become more commonplace in the near future, although its rendering quality for screen resolution is not ideal.

Unicode and availability of mathematical fonts, without changing the main body of formal sources. Users can augment the tables by domain-specific notation for their own applications.

(c) Careful combination of (a) + (b). The formal language prefers named symbols of the form \<*name*> but the user is allowed to refer to UTF-8 Unicode, say in informal text. Since UTF-8 is conservative over 7-bit ASCII byte streams anyway, this hardly requires any change of the prover. Even though the prover never uses genuine Unicode in the standard libraries, users can write text in foreign scripts like Cyrillic or Chinese.

Although the re-mapping of (b) introduces some technical inconveniences in the implementation, such adjustments have been part of Proof General [3] for many years, re-using older packages for TEX. The JVM also allows to register character encodings defined in user space. The combination of (c) is already officially recognized in Isabelle2011 [14, §1.2.1]: it mainly refines the 1998 version of (b) by precise of addressing Unicode text positions as explained in §2.2.

In summary, we essentially use Unicode for *poor man's rendering* of mathematical text, using those parts of the technology that are known to work reliably.

2.2 Markup via Text Addressing

We define *markup* as any information that is adjoined with the original source text. In XML the markup elements happen to consist of a *name* and *property list* (called *attributes* in XML terminology). We take care to coincide with such public standards when delivering documents to external tools, although internally the exact structure of markup does not matter. In Isabelle/Scala many operations on markup are either parameterized over some arbitrary type `A` or use the universal union type `Any`.

There are two interchangeable ways to represent text together with nested markup:

1. Explicit trees that alternate markup nodes with body text (in the spirit of XML trees, see also §3).
2. Implicit annotations for the text as separate mapping from text intervals to markup nodes (maintained as side-result of formal checking, see also §4).

The second variant is not immediately obvious, but turns out very convenient for the prover in most situations: it can continue to digest text as before, but occasionally reports annotations about its internal state, in correspondence to precise source positions. So the markup problem is reduced to robust addressing of text positions, to anchor the attached semantic information in its proper place (e.g. inferred type information, or warnings and errors that refer to parts of text).

Problem: Robust physical addressing of text offsets that is suitable for persistent storage and stable under change of encodings or rendering (cf. §2.1).

Solution: Given a byte stream, individual *symbols* are identified as follows:

– a single ASCII character (byte 0...127)
– a codepoint according to UTF-8 (byte 128...255 followed by the longest possible sequence of bytes 128...191)
– a named symbol `\<`*name*`>` where *name* consists of ASCII letters and digits
– a malformed symbol `\<` that cannot be extended to a named symbol

This defines a *total* scan function on packed symbol sequences, which is available as `Symbol.explode: string -> string list` in Isabelle/ML. Totality is achieved by tolerating malformed symbols (including illegal UTF-8, which happens to be self-synchronizing). Further syntax layers can reject those, e.g. in lexical analysis. Positions gained from counting symbols are passed through the syntax hierarchy as intervals of logical offsets. Raw byte addressing is not used.

Isabelle/Scala is subject to UTF-16 on the JVM, but the same notion of symbols can be accommodated. We provide some *iterator* in Scala that enumerates the symbols as consecutive intervals over JVM strings. The handling of surrogate UTF-16 characters is also included here, to ensure that logical codepoints are counted in the same way as the corresponding UTF-8 sequences in Isabelle/ML.

2.3 YXML Transfer Syntax

Proof assistants prefer to output plain text, but we would like to augment that by markup information. It is unrealistic to expect that existing ML code for thousands of messages in the prover and add-on tools are reworked to use XML trees instead of strings. Here is a typical example from the Isabelle sources, which composes literal text and formatted output of pretty-printed formal entities:

```
error ("Unbound schematic variable: " ^ Syntax.string_of_term ctxt t)
```

Error messages are particularly delicate, since they only occur in exceptional situations. It would take a long time to discover the mistakes that are inevitably introduced by forcing XML onto the prover in a crude manner.

Problem: Robust presentation of markup for plain text, such that markup is orthogonal to the text (with its encoding of symbols, cf. §2.1) and markup can be nested safely (applied to text that might have been marked before).

Solution: First we observe that text produced by the prover never contains low-ASCII control characters, except for white-space. Second we observe exactly the same for regular text characters in the XML standard.[3] Third we check old ASCII control tables, and choose $\mathbf{X} = \mathit{chr}\ 5$ and $\mathbf{Y} = \mathit{chr}\ 6$, which are unlikely to have any special meaning in current computer systems. We can now define our own *YXML transfer syntax* for XML documents as follows:

	XML	YXML
open tag	`<`*name attribute*`=`*value* ...`>`	$\mathbf{XY}\mathit{name}\mathbf{Y}\mathit{attribute}\mathbf{=}\mathit{value}...\mathbf{X}$
close tag	`</`*name*`>`	$\mathbf{XYX}$

[3] http://www.w3.org/TR/xml/#charsets

Note that there is no special case for elements with empty body text, i.e. `<foo/>` is treated like `<foo></foo>`. Body text is represented literally, without escapes.

Unlike official XML syntax, our format has some nice properties:

- YXML can be inlined directly into the string representation of text messages. It is orthogonal to UTF-8 encoding (since **X** and **Y** are part of 7-bit ASCII), orthogonal to Isabelle symbol notation (since it avoids `<>`), and independent of previous markup (marking text does not alter it, no escaping of `<>\&"'`).
- YXML parsing is straight-forward: first split the text into chunks separated by **X**, then split each chunk into sub-chunks separated by **Y**. Markup tags start with an empty sub-chunk, and a second empty sub-chunk indicates a close tag. Any other non-empty chunk consists of literal text.
  Thus efficient YXML parsers can be implemented in the programming language of choice in one afternoon, e.g. by taking the 1-page implementations in Isabelle/ML or Isabelle/Scala as blueprints.

Isabelle/ML already provides some library functions to inline YXML markup into text, using `Markup.markup: Markup.T -> string -> string` like this:

ML ⟪
```
warning ("Potential problem:\n" ^ Markup.markup Markup.malformed
  ("bad variable variable " ^ Markup.markup Markup.hilite "x"))
```
⟫

JVM-based front-ends that receive Isabelle messages can use Isabelle/Scala library operations to parse the YXML representation, and work directly with XML trees. The above example results in the following structured message (in XML syntax), which also includes a header provided by the `warning` function:

```
<warning serial="409542" offset="1" end_offset="3" id="31">Potential problem:
<malformed>bad variable variable <hilite>x</hilite></malformed></warning>
```

Adding occasional markup to prover messages is adequate, but we do not even need this in most practical situations. Provers already provide standard library functions to produce string representations of formal entities (types, terms, theorems), such as our `Syntax.string_of_term` seen before. Since YXML is inlined into the ML string type, we can easily augment the standard operations to add markup transparently, without affecting user code that is well-behaved (refrains from analyzing formatted prover output).

Likewise, the ML function `Position.str_of: Position.T -> string` that prints text positions can be modified to include its own markup. Thus we get machine-readable positions in messages, and front-ends can easily display prover output with clickable spots or attach messages directly to the source view.

The following example illustrates such implicitly enhanced messages produced by existing text-oriented application code.

ML ⟪
```
warning ("Term: " ^ Syntax.string_of_term ctxt t ^ Position.str_of pos)
```
⟫

Here the printed term `t` is $x + y$: it contains some undeclared variables that Isabelle highlights to warn the user. There is also some reference to the logical constant behind the "+" notation. The front-end receives the following output:

```
<warning serial="409564" offset="1" end_offset="3" id="37"Term:
<term><block indent="0"><block indent="0"><hilite><block
indent="0"><free><block indent="0">x</block></free></block></hilite>
<const name="Groups.plus_class.plus"><block
indent="0">+</block></const><break width="1"> </break><hilite><block
indent="0"><free><block
indent="0">y</block></free></block></hilite></block></block></term><position
offset="79" end_offset="82" id="-35"/></warning>
```

This means the prover now speaks XML natively, without changing the application code. Further markup content can be provided gradually, using XML as data language to represent certain aspects of the prover state, see also §3.

In retrospect, the re-use of text characters `<>\&"'` in official XML syntax (inherited from SGML) turns out as big obstacle for adoption in old-school applications. Earlier attempts of Isabelle/PGIP implementation [4] suffered from the difference of escaped vs. unescaped text in prover messages: the "polarity" of marked strings was often wrong, resulting in display errors or protocol crashes.

YXML can easily and efficiently repair the problems of concrete XML syntax. In a sense, our format returns to the original idea of "markup" before SGML and XML: information is attached to some text without altering it. One can also think of a physical text marker, which does not "escape" any text it touches.

3 XML Content

Stripped of its concrete syntax, what is the essence of XML and how can we use it to represent content of the prover adequately? At the bottom of the enormous XML standard documents, there is the bare tree structure consisting of text and markup elements (with name and attributes). This is specified in 3 lines of ML:

```
datatype tree =
  Elem of (string * (string * string) list) * tree list
| Text of string
```

The same is specified in 3 lines of Scala using *case classes* [7]:

```
sealed abstract class Tree
case class Elem(markup: (String, List[(String, String)], body: List[Tree])
  extends Tree
case class Text(content: String) extends Tree
```

This is naked, untyped XML — no more no less. Add-on standards for typed XML documents do exist (e.g. DTD, XML Schema, Relax NG), but there is no

universal agreement, and availability in ML or Scala is limited. Nonetheless, the prover and its front-ends need to assign some meaning to XML trees.

Problem: Select suitable mechanisms to interpret raw XML documents and define concrete meaning for various prover document markup elements.

Solutions: From the many possibilities, we subsequently present schemes that are already used in Isabelle.

3.1 Typed Views on Untyped Data

We can think of XML trees as the raw material to represent content in memory, transport it between provers and front-ends, or store it in persistent databases and repositories. This is similar, to frugal s-expressions in LISP, although XML provides even less explicit data formats, basing everything on plain strings.

In ML and Scala we prefer to work with typed trees, following the tradition of algebraic datatypes. In practice there is a slight asymmetry: ML functions of the prover usually *produce* XML trees (by direct inlining of YXML, cf. §2.3), while the Scala/JVM front-ends *consume* them (parsing some tree content). Luckily each programming language provides sufficient means to represent these notions in simple manners, without requiring a full framework of "meta-programming" for externally specified XML datatypes.

ML *constructor functions* encode typed data as untyped markup, which can be inlined into YXML text.

Scala *extractor functions* analyse the content of untyped XML trees, after YXML parsing. This works via customized `unapply` methods of some Scala objects that model the notions of document content. (Symmetrically, it is also possible to define constructors via `apply` methods.)

For example, the ML function `Markup.proof_state: int -> Markup.T` encodes the number of pending sub-goals (of integer type) as textual markup node: `Markup.proof_state 42` is received by the JVM front-end as `<proof_state subgoals="42"/>` where the `tree` can be matched in Scala via `tree match { case Proof_State(i) => i }` to recover the integer 42.

To make such pattern matching work, Scala requires object `Proof_State` with some `unapply` method defined beforehand (e.g. in the prover library):

```
object Proof_State {
  def unapply(tree: XML.Tree): Option[Int] =
    tree match {
      case XML.Elem(Markup("proof_state",
        List(("subgoals", Markup.Int(i)))), Nil) => Some(i)
      case _ => None
    }
}
```

More ambitious ML/Scala connectivity could augment the Scala compiler by some plugin to produce these definitions from concise specifications. Such sophisticated meta-programming is outside our present scope, though.

Note that these data views are inherently partial: dynamic type mismatch can occur, and indicate some fault of the protocol infrastructure of the prover. User code usually works with statically-typed `Markup.proof_state` in ML and `Proof_State.unapply` in Scala, to minimize programming errors.

3.2 Augmented Oppen Pretty-Printing

Proof assistants mainly operate on symbolic term structures, which are eventually displayed to the user via *pretty printing*. This often works by some variant of Pretty-Printing according to Oppen [8]: text *atoms* and potential line *breaks* are arranged as nested *blocks* (with optional *indentation*). The resulting pretty tree can be formatted to match a given margin: the pretty printing algorithm inserts physical spaces and newlines as required.

The initial indication of breaks and blocks already constitutes a physical markup language, which can be augmented for our document-oriented architecture as follows.

1. Pretty blocks may also carry *logical markup*. The Isabelle/ML function `Pretty.markup: Markup.T -> Pretty.T list -> Pretty.T` inserts markup information into pretty trees, analogous to `Markup.markup` on plain text as seen before (§2.3).
2. In ML the pretty formatting is made *symbolic*. Instead of former physical layouting, the information about breaks and blocks is turned into XML markup elements `<break width="N"/>` and `<block indent="N">...</block>`, respectively. Together with the logical markup this constitutes an XML document, which is inlined into prover output by YXML encoding.
3. In Scala the physical formatting is now done directly on XML trees. Break and block markup elements are expanded to produce appropriate spaces and newlines within the body text; the remaining XML tree only contains the logical markup. The formatting algorithm imitates the former ML implementation closely, using typed views `Break(width)` and `Block(indent, body)` to recognize symbolic breaks and blocks in raw XML trees.
4. Final rendering works via XHTML/CSS, using the JVM-based Lobo/Cobra Web browser, for example. Markup elements `<foo>...</foo>` are turned into `<span class="foo">...</span>` (ignoring attributes) and then given to the browser together with an XHTML header and prover-specific style sheet. The XHTML text is essentially treated as preformatted, to retain the indentation and line breaking performed before.

Presently, Isabelle merely uses this XHTML rendering scheme to produce traditional highlighting of certain entities within printed terms, e.g. global variables in blue, local variables in brown, bound variables in green, undeclared variables

with yellow background. Much more can be done by exploiting the full potential of XHTML/CSS, with boxes and table layouts, for example.

Since the Web browser allows to attach program code to HTML elements (either as JavaScript or JVM code, which can be produced in Scala), we could interpret certain prover markup to fold/unfold sub-terms, or produce popups that reveal information about the formal entities at that point (the place of definition, documentation etc.). Using HTML input forms, one could also support simple interaction schemes based on prover output: the proof assistant presents some templates that the user can complete and insert into the source text.

4 Document Reconstruction

With the text markup and content infrastructure of §2 and §3 available, we can now look deeper into semantic prover information. Classic command-line provers only produce *output* in the form of messages, but in a document-oriented setting we can attach valuable information to the *input* via markup.

Conceptually, the sources define the true meaning of the text. By processing the sources, the prover explores the meaning incrementally, as represented by some internal configuration. Aspects of this semantic information can be attached to the original sources to help the user understand the text, or support tools that operate on the text systematically (e.g. via "refactoring").

This means, an approximation of the true semantic prover content is *reconstructed* as externally readable document. The prover as the only instance that can really analyze the formal text reveals important aspects about its content, without requiring front-ends to imitate substantial parts of the prover itself.

Inherent Complexity of Prover Syntax. Presenting sources of formal languages with rich semantic annotations is nothing new. For example, an IDE for Java understands the syntax and static semantics of the language (with scopes and type information for identifiers) in order to support systematic refactoring of the program text. Frameworks like Eclipse already provide powerful libraries (such as Xtext) to implement such functionality for user-defined languages.

Unfortunately, hardly anything like this works for common proof assistants. Apart from some lexical analysis and superficial parsing, there are many more syntax layers until fully typed λ-terms emerge internally. Some of these phases are well-defined, like Hindley-Milner type-inference over the order-sorted algebra of type-classes in Isabelle. Other phases are open-ended, providing computationally complete plugin-mechanisms: recent Isabelle allows to (re)define the type-discipline in user space, based on arbitrary ML code. The symmetric situation happens for output of λ-terms, until some printable text is eventually produced. Any of these mechanisms can depend on local declarations according to the scoping rules of the formal language.

The latter is illustrated by the following trivial example in Isabelle/Isar, with local mixfix declarations that modify some notation within proof blocks:

```
notepad
begin
  fix foo :: 'a ⇒ 'a ⇒ 'a
  fix bar :: 'a ⇒ 'a ⇒ 'a
  { write foo (infix · 70) have x · x = foo x x .. }
  { write bar (infix · 70) have x · x = bar x x .. }
end
```

Even more, Isabelle allows to *transform* arbitrary declarations by morphisms, to move between different formal contexts. We conclude that any prover IDE or theory browser that attempts to analyze the text directly is bound to fail!

Interestingly, the situation in Coq is similar. For example, there have been approaches to direct structural manipulation via "proof by pointing" [5], with strong assumptions about the content of Coq syntax trees. These were violated later as the prover evolved, causing the PCoq front-end [1] to break eventually.

Builders of standard IDE syntax plugins might shake their heads and demand that provers are restricted to decidable syntax, or deliver abstract syntax trees with full information. We argue that this is unrealistic: existing theory libraries with sophisticated notation would have to be reduced or discontinued. Moreover, current provers like Coq or Isabelle do not even address the full complexity of mathematical notation yet, and the trend is towards even more complex notational devices as can be seen in Matita [2], for example.

Problem: Find ways to transport semantic information through many sophisticated layers of prover syntax. Avoid strong assumptions about prover syntax trees. Observe the matter of fact that the prover cannot provide complete information, only some important aspects.

Solution: Careful inspection of the syntax hierarchy of sophisticated provers like Isabelle leads to the following observations:

1. Plain text is clearly observable by external tools, and the prover can agree with front-ends about precise addressing of text positions (cf. §2.2).
2. The main elements of the theory and proof language (called "commands" in Isabelle terminology) can be understood as *transactions*, which define clearly delimited updates on internal state. Assuming that execution of a transaction proceeds linearly (with monotonic increase of internal state information and without back-tracking) we can collect a *trace* of it as part of the result.
3. Trace information can be assembled into an external *markup tree* that is associated with the original piece of text in the transaction context.

Reporting of markup for certain positions generalizes the idea of printing text messages to the user. The Isabelle/ML message channels of `error`, `warning` etc. are extended by `Position.report: Position.T -> Markup.T -> unit` for maintaining the implicit markup tree of the current transaction. The prover merely needs to pass precise positions through to a point where certain formal content is discovered, and report an externalized version back to the sources.

Figure 1 illustrates why this makes an important conceptual difference: instead of demanding full information about sophisticated phases of parsing $f_1;f_2;f_3;f_4;f_5$ and printing $g_1;g_2;g_3;g_4;g_5$, we merely preserve position information up to the point of certain reports after $f_1;f_2$ and $f_1;f_2;f_3$, for example.

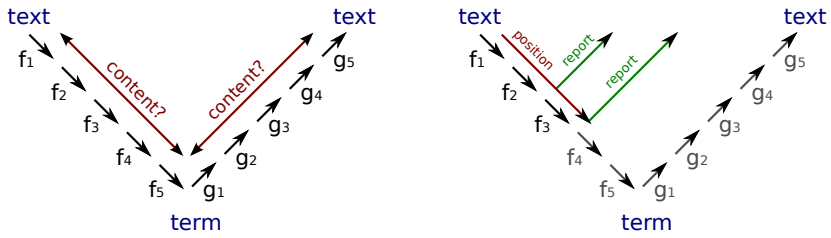

Fig. 1. Full content of input/output phases versus reports on some input phases

This means *less* information is passed through *some* of the input phases only. With this conceptual simplification in place, we now need to rework some central prover syntax modules, in order to provide useful reports. It might require extra tricks to preserve precise position information in particular phases, e.g. the move from quoted "outer syntax" to "inner syntax" in Isabelle, where we revive the ancient ASCII DEL character (127) for padding.

In Isabelle the number of syntax phases is not 5, but approximately 12. For the inner syntax of terms and types this includes computationally complete mechanisms like so-called "translation functions", which are used for example to implement derived binders or implicit state-dependence as in Hoare logic notation. The regular type-checking phase has its own plugin-in concept, which typically includes extensions of Hindley-Milner type-inference, with "type improvement" and optional "coercion functions". It is unrealistic to expect that fully annotated syntax trees can be passed through all these phases, but position reports according to our scheme of figure 1 are quite feasible, after some minor reworking of the existing code base.

Document Markup for Prover IDEs. Isabelle2011 already reports about its "outer syntax" of tokens and command spans, including various binding positions that are directly accessible (e.g. the points where *foo* and *bar* were fixed in the previous example). The "inner syntax" of types and terms is more complex: tokens, raw parse trees, scopes of free versus bound variables are already accessible for reports, but type-information and sub-expression markup is still missing. The key idea for non-trivial term annotations, which is partially implemented in Isabelle repository versions after official Isabelle2011, is as follows: source positions within the raw term language are disguised as type constraints, and passed through the main syntax translation layers (where many user-defined transformations exist), until the point where Hindley-Milner type-inference happens. By

construction of traditional lambda calculus with optional type-constraints, old syntax operations are expected to handle such extra annotations (but in practice it often means to repair a few omissions in user code).

The Poly/ML 5.4 compiler (by David Matthews) that is used with Isabelle2011 also produces significant reports about the results of static analysis of Standard ML. The following ML snippet inside Isabelle/Isar illustrates this, although the results are unavailable in the classic PDF version of this document.

```
ML ⟪
  fun foo th = Thm.prems_of (th RS @{thm refl})
⟫
```

Processing these sources with the experimental Isabelle/jEdit Prover IDE of Isabelle2011 (see also figure 2), the document markup is presented as boxed sub-expressions, tooltips for inferred types, and hyper-links for referenced identifiers (such as the local `th` and the global `RS` operator from the Isabelle/ML library).

The syntax highlighting for tokens etc. is also based on precise semantic information from Isabelle: the editor merely retrieves it from the universal document markup tree.

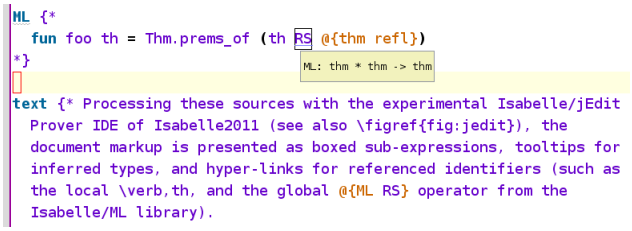

Fig. 2. IDE presentation of formal document content of these sources

Many more possibilities are conceivable, e.g. indicating variable scopes by highlighting binding and referencing positions simultaneously (as in common IDEs), or popup menus pointing to other formal documents that explain the referenced concepts (Isabelle manuals already provide suitable formal markup).

5 Conclusion and Related Work

We have shown how to augment old-school proof assistants to support text documents with rich semantic information, while minimizing the changes to the existing code base. The cultural background of provers implemented in ML (or Haskell) is preserved, we refrain from pervasive "XML-ization" of the programming style. Moreover, the increasingly popular Scala language on the JVM helps to extend the scope of the prover into a greater world.

This work can be seen as a continuation of earlier approaches to overcome the plain-text attitude of proof assistants: CtCoq/Pcoq [5], PLATΩ [11] and especially PGIP [4]. The efforts around PGIP and its (partial) implementation in Isabelle2004/2005 posed many problems that are now addressed quite differently than first anticipated: our problem statements and proposed solutions are best understood in the context of former struggles with PGIP. For example, PGIP demands to implement "proof-script parsing" on the spot without any further context. In contrast, our present approach lets the prover report information incrementally whenever it becomes available.

Matita [2] is probably the best-known proof assistant that was designed with some IDE support (based on Gtk) and advanced presentation formats (MathML) from the ground up. Thus it had the privilege of a fresh start, without a huge existing code base to take care of. Nonetheless, the Matita authors report significant efforts to re-build basic infrastructure for rich mathematical output (such as their own MathML rendering engine in OCaml/GTk). Even after substantial efforts, the Matita front-end still resembles traditional Proof General [3]. In contrast, our general approach is to maximize re-use, by augmenting existing provers with minimal effort and simplifying connectivity to powerful front-ends (IDEs, browsers, Web frameworks), especially on the JVM. Moreover, our document model of annotated prover sources goes beyond the basic IDE functionality of Matita, which is more concerned about output of mathematical formulae than annotating input sources.

Earlier work on "XML-izing Mizar" [10] is based on the assumption that complete syntax trees in XML are feasible, in contrast to our observation of "inherent complexity of prover syntax" as discussed before. Despite its own builtin complexity, Mizar is a closed language and defining fixed XML formats for its tool chain turned reasonably easy. It also allows users to implement derivative tools, using XSLT for example. This more conventional way to externalize prover content would be hardly possible for the more complex and open-ended LCF family, notably Isabelle, Coq, and HOL.

Proviola [9] shares our goals to extract document content from prover runs. This is done for Coq, without modifying the prover itself (due to lack of access to its inner circle of developers). This limits the document content to traditional proof state and message output, for example. Our work can be understood as providing the missing link on the prover side: it could simplify projects like Proviola (and its greater perspective of prover Web clients and Wikis for formalized mathematics), and also provide more interesting content.

Ultimately, only the prover itself can reveal substantial document content, derived from its true semantic state. We have shown how this can be achieved in a profound manner, without rewriting major proof assistants from scratch. In the future we hope to see more proof assistants re-use these concepts, and more front-ends exploiting the content delivered by such document-oriented provers.

References

[1] A. Amerkad, Y. Bertot, L. Pottier, and L. Rideau. Mathematics and proof presentation in Pcoq. In *Proceedings of Proof Transformation and Presentation and Proof Complexities (PTP'01)*, 2001.

[2] A. Asperti, C. Sacerdoti Coen, E. Tassi, and S. Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2), 2007.

[3] D. Aspinall. Proof General: A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*. Springer, 2000.

[4] D. Aspinall, C. Lüth, and D. Winterstein. A framework for interactive proof. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants, 14th Symposium, Calculemus 2007, 6th International Conference, MKM 2007*, volume 4573 of *LNCS*. Springer, 2007.

[5] Y. Bertot and L. Théry. A generic approach to building user interfaces for theorem provers. *Journal of Symbolic Computation*, 25(7), 1998.

[6] M. Gordon, R. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Principles of programming languages (POPL)*, 1978.

[7] M. Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne, 2004.

[8] D. C. Oppen. Pretty printing. *ACM Transactions on Programming Languages and Systems*, 2(4), 1980.

[9] C. Tankink, H. Geuvers, J. McKinna, and F. Wiedijk. Proviola: A tool for proof re-animation. In S. Autexier, J. Calmet, D. Delahaye, P. D. F. Ion, L. Rideau, R. Rioboo, and A. P. Sexton, editors, *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010*, volume 6167 of *LNCS*. Springer, 2010.

[10] J. Urban. XML-izing Mizar: Making semantic processing and presentation of MML easy. In M. Kohlhase, editor, *Mathematical Knowledge Management (MKM 2005)*, volume 3863 of *LNCS*. Springer, 2006.

[11] M. Wagner, S. Autexier, and C. Benzmüller. PLATΩ: A mediator between text-editors and proof assistance systems. In S. Autexier and C. Benzmüller, editors, *User Interfaces for Theorem Provers (UITP 2006)*, volume 174(2) of *ENTCS*. Elsevier, 2007.

[12] M. Wenzel. Isabelle/Isar — a generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. University of Białystok, 2007.

[13] M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In C. Sacerdoti Coen and D. Aspinall, editors, *User Interfaces for Theorem Provers (UITP 2010)*, ENTCS, July 2010. FLOC 2010 Satellite Workshop.

[14] M. Wenzel. *The Isabelle/Isar Implementation Manual*, 2011.

[15] M. Wenzel and A. Chaieb. SML with antiquotations embedded into Isabelle/Isar. In J. Carette and F. Wiedijk, editors, *Workshop on Programming Languages for Mechanized Mathematics (PLMMS 2007). Hagenberg, Austria*, June 2007.

[16] F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNAI*. Springer, 2006.