A Logic-Independent IDE

Florian Rabe

Computer Science, Jacobs University, Bremen, Germany
f.rabe@jacobs-university.de

Abstract. Interactive deduction systems are becoming increasingly powerful and practical. While design has historically focused on questions such as the choice of logic or the strength of the prover, modern systems must additionally satisfy practical requirements such as good IDEs or easily-searchable libraries. We hypothesize that many such practical requirements can be solved independently of the specific logic or prover, and that doing so will enable a fruitful separation of concerns between logic/prover and user interface developers.
To that end, we design and implement a logic-independent IDE-style user interface, which we base on the jEdit editor and the Mmt language. jEdit is a full-fledged and extensible text editor, and Mmt is a logic-independent representation format that already offers a variety of generic knowledge management services. Taken together, they let us realize strong user support at comparatively little cost: Example features of our IDE include hyperlinking, awareness of inferred information, library browsing, search, and change management.

1 Introduction and Related Work

Motivation Deduction systems such as type checkers, proof assistants, or theorem provers have initially focused on soundness and efficiency. Consequently, the most natural design choice for the user interface (UI) has often been a kernel that implements the logic and the theorem prover and with which users interact through a read-eval-print loop. But over time formalization projects have reached larger scales that call for more sophisticated UIs, and it has proved non-trivial to add these to existing kernels.

Users more and more expect interactive, flexible UIs, e.g., like the ones they know from software engineering. They ask for features such as highlighting of errors, fast rechecking after editing any part of the project, or a powerful search function over the library. For example, in personal communications with the author, members of the Feit-Thompson project in Coq [GAA+13] pointed out the UI as a key bottleneck (rather than the type theory or the theorem prover); and Georges Gonthier expressed his wish for a better UI for searching. Similarly, members of the L4.verified project in Isabelle [BDKK12] urged for better change management support because their work was often delayed by long rechecking cycles. And for casual users from outside the theorem proving community, the UI can be more important than the underlying logic.

However, developers' resources, tend to be stretched already by developing (and maintaining) the kernel at all. Therefore, it is no coincidence that none but the most mature systems like Coq [The14] or Isabelle [Pau94] are equipped with graphical UIs. Arguably the most powerful UI is Makarius Wenzel's Isabelle/jEdit [Wen12], and its history is telling: It required a multi-year effort by one of the lead developers, throughout which he had to reimplement parts of the kernel in order expose the needed kernel functionality to the UI component.

In this situation, it is important to investigate whether powerful UIs can be realized independently of the kernel. This is well-known to be the case in software engineering, where IDE frameworks like Eclipse [Ecl] are parametric in the programming language, which is supplied as a plugin. This separation of concerns between kernel and UI development would be even more valuable in theorem proving, where developer communities are comparatively small.

Related Work The idea of generic UIs for theorem proving has so far mainly been applied to **lightweight** user interfaces. These typically interact with an abstract kernel through a read-eval-print interface. This has the advantage that almost any kernel can be plugged into the UI at relatively little cost. Examples are Proof General [Asp00], which uses a text editor, and Proof Web [Kal07], which uses a web browser to host the UI. Similarly, [ABMU11] provides a wiki-like frontend for Coq and Mizar.

It is often possible to enrich this abstraction layer with configuration and query commands. But the lightweight approach is inherently limited by the fact that the UI is not aware of the abstract syntax tree (AST) that is computed and refined by the kernel.

We speak of **heavyweight** UIs if the UI is aware of (at least) the AST. Here the AST should include the results of, e.g., disambiguation, type reconstruction, and theorem proving, and nodes should carry cross-references to the corresponding regions in the source. This awareness enables IDE-style UI features such as AST display, error highlighting, tool tips, hyperlinks, and auto-completion. Moreover, the kernel may expose functions for rechecking changed fragments (and their dependencies) to enable change management. Aside from the one we present here, Isabelle/jEdit [Wen12] is the only heavyweight UI we are aware of, and we will relate the two in Sect. 5.

The disadvantage of heavyweight UIs is that it becomes much harder to plug in an existing kernel (especially if they are implemented in a different programming language than the UI). In fact, it is not always obvious whether it would be easier to develop a heavyweight UI for a specific existing kernel or to connect a generic heavyweight one.

A compromise can be to develop UIs for logical frameworks like Isabelle [Pau94], which can be inherited by all object logics. A similar argument applies to UIs for very rich systems like Coq [The14], whose logic can be customized by disabling language features or adding axioms. The Agda kernel [Nor05] achieves advanced UI features by dynamically generating code that is evaluated by the emacs-based UI.

Our Approach We hold that in the long run, a decoupling of kernel and UI can prove beneficial if only because there are a lot more potential UI than kernel developers. Somewhat provocatively, we observe that in monolithic systems, the few people who understand the kernel well enough to improve the UI mostly do not have the time and incentive to do so.

Therefore, we introduce jMMT, a logic-independent heavyweight user interface. Our key idea is to design a rich abstraction layer between kernel and UI. This permits the logic-independent development of UI features, which results in a valuable separation of efforts:

- Logic developers can focus on designing the logic and implementing the kernel. This frees valuable resources to improve the theorem prover.
- UI developers can focus on developing UI features without having to understand any logic or prover. They benefit twice because their work becomes easier and more reusable.
- Finally, this benefits developers of auxiliary components like proof hint generators (e.g., [KU13]) or search engines (e.g., [KŞ06]). They are enabled to make their services logic-independent and directly integrate them with the UI.

We base our abstraction layer on the MMT language [RK13], a logic-independent representation language that admits natural embeddings of virtually all declarative languages. We model a kernel as a set of components for parsing and validating the structure of a document as well as the terms within it. This lets us develop advanced UI features such as change management and search in addition to the ones we mentioned above.

On the implementation side, our IDE makes use of the MMT API [Rab13], which already offers a high-level machine interface to MMT content, as well as the jEDIT text editor, which offers a rich plugin framework for both UI and language-specific extensions. To connect them, we develop a plugin for jEDIT based on MMT, which implements our abstraction layer and the UI functionality.

To evaluate and exemplify this design, we have developed a kernel for the logical framework LF [HHP93] that instantiates our abstraction layer. This yields a powerful IDE for LF, whose UI features surpass existing implementations of comparable languages.

Moreover, our LF kernel is designed to be highly extensible. Thus, it is possible to develop additional kernels quickly by plugging in, e.g., different operators, notations, or typing rules.

Overview Sect. 2 summarizes the preliminaries we need about jEDIT and MMT. In Sect. 3, we describe the UI functionality, which amounts to the user perspective of jMMT. In Sect. 4, we describe our abstraction layer, which amounts to the kernel developer perspective of jMMT.

Source code, binaries, documentation, and additional screenshots and videos are available from the MMT homepage[1]. Early parts of this work were carried out with Mihnea Iancu and presented as work-in-progress [IR12].

[1] https://svn.kwarc.info/repos/MMT/doc/html/index.html

2 Preliminaries

2.1 The jEdit Text Editor

jEdit is a widely-used Java-based text editor [jEd]. It is particularly interesting as a UI for formal systems due to its strong plugin infrastructure that can be used to provide IDE-like functionality. Thus, it provides a lightweight alternative to complex IDE frameworks like Eclipse [Ecl].

Existing plugins already provide abstract interfaces for among others outline view, error highlighting, auto-completion, hyper-linking, and shell integration that can be implemented by format-specific plugins. Thus, relatively little glue code is necessary to connect the UI components and the language-specific data model.

2.2 The MMT Representation Language

MMT [RK13] is a prototypical declarative language. It systematically avoids committing to an individual logic or type theory and focuses on capturing their joint structural properties. Specific language features such as function types or equality are defined in separate modules, which serve as building blocks to compose languages.

Theory	Σ	::=	$\cdot \mid \Sigma,\ c[: E][= E][\#N]$
Context	Γ	::=	$\cdot \mid \Gamma,\ x : E$
Term	E	::=	$c \mid x \mid c(\Gamma; E^*)$
Notation	N	::=	$(\mathcal{V}_n \mid \mathcal{A}_n \mid \texttt{string})^*$

Fig. 1. MMT Grammar

Here we introduce only a small fragment of MMT that is sufficient to exemplify our IDE. The grammar is given in Fig. 1. A **theory** Σ is a list of **constant** declarations. A **constant** declaration is of the form $c[: A][= t][\#N]$ where c is an identifier, A is its type, t its definiens, and N its notation, all of which are optional. A **context** Γ is very similar to a theory and declares typed variables.

`type`	#	`type`
`kind`	#	`kind`
`Pi`	#	$\{\,\mathcal{V}_1\,\}\,\mathcal{A}_2,$
`lambda`	#	$[\,\mathcal{V}_1\,]\,\mathcal{A}_2,$
`apply`	#	$\mathcal{A}_1\ \ \mathcal{A}_2,$
`arrow`	#	$\mathcal{A}_1 \to \mathcal{A}_2$

Fig. 2. LF in MMT

Type and definiens are **terms**, which are formed from the constants, variables, and complex terms. We use a general form $c(\Gamma; E_1, \ldots, E_n)$ for complex terms, which subsumes binding and application: c is called the **head** of the term, the (possibly empty) context Γ declares the **bound variables**, and the E_i are the **arguments**. For example, in a λ-term $\texttt{lambda}(x : A; t)$, λ is the head, $x : A$ the bound variable context, and t the single argument.

The notation of c is used for the concrete representation of complex terms with head c: $\mathcal{V}_n$ refers to the n-th bound variable, $\mathcal{A}_n$ to the n-th argument (counting starts from the last variable), and they may be interspersed with arbitrary delimiters. For example, if we declare the constant `lambda` with the notation $\lambda\,\mathcal{V}_1\,.\,\mathcal{A}_2$, then $\lambda x : A.t$ becomes the concrete representation of $\texttt{lambda}(x : A; t)$.

The following theories will serve as running examples throughout the paper:

Example 1 (LF as an MMT *Theory).* To obtain the logical framework LF, we use the theory shown in Fig. 2. These constants do not have types. Instead, they constitute primitive concepts, which can be used to form terms, which can then occur as the types of other constants. However, their binding-arity and argument-arity can be inferred from their notations, e.g., `Pi` binds one variable and then takes one argument.

Once LF is defined, we can use it as a logical framework to define other logics:

Example 2 (A Logic in LF). Using the theory from Ex. 1, we can define a theory for a simple logic by declaring (among others) the following constants

$$\begin{array}{lll}
\mathtt{prop} & : \mathtt{type} & \#\,\mathtt{prop}, \\
\mathtt{ded} & : \mathtt{prop} \to \mathtt{type} & \#\,\mathtt{prop}, \\
\mathtt{imp} & : \mathtt{prop} \to \mathtt{prop} \to \mathtt{prop} & \#\,\mathcal{A}_1 \Rightarrow \mathcal{A}_2, \\
\mathtt{and} & : \mathtt{prop} \to \mathtt{prop} \to \mathtt{prop} & \#\,\mathcal{A}_1 \wedge \mathcal{A}_2, \\
\mathtt{andI} & : \{A\}\{B\}\,\mathtt{ded}\,A \to \mathtt{ded}\,B \to \mathtt{ded}\,(A \wedge B) & \#\,\mathtt{andI}\,\mathcal{A}_3\,\mathcal{A}_4, \\
\mathtt{impI} & : \{A\}\{B\}\,(\mathtt{ded}\,A \to \mathtt{ded}\,B) \to \mathtt{ded}\,(A \Rightarrow B) & \#\,\mathtt{impI}\,\mathcal{A}_3 \\
\mathtt{example} & : \{A\}\mathtt{ded}\,(A \Rightarrow (A \wedge A)) \ = \ [A]\mathtt{impI}\,[p]\,\mathtt{andI}\,p\,p &
\end{array}$$

This theory uses the Curry-Howard representation of judgments as types: The provability judgment of $A : \mathtt{prop}$ is represented as the term $\mathtt{ded}\,A$. The constant `example` declares and proves a (trivial) theorem using the natural deduction rules for conjunction and implication introduction. We already omit inferable variable types and declare arguments to be implicit by not mentioning them in the notation (e.g, the first 2 arguments of `impI` are implicit).

MMT abstracts from the typing relation between terms. It fixes only the judgments about terms as given in Fig. 3. These judgments can be defined arbitrarily by specific languages represented in MMT.

$\vdash_T A\ \mathtt{TYPE}$	inhabitability, i.e., A may occur as the type a constant
$\vdash_T E : A$	typing relation, in particular E may occur as the definiens of a constant with type A
$\vdash_T E = E' : A$	equality of terms at type A

Fig. 3. MMT Judgments Relative to a Theory with Name T

The MMT **implementation** [Rab13] provides an API for maintaining MMT content. It is written in Scala and thus fully interoperable with the Java codebase of JEDIT. It is not an application with its own user interface. Instead, it focuses on providing services that can be used in MMT-based applications (such as our IDE).

The MMT implementation is highly extensible and relegates many features to plugin interfaces. In particular, to instantiate MMT with specific languages, one supplies a plugin that implements the judgments from Fig. 3.

Note that via Curry-Howard, **axioms**, **theorems**, **tactics**, **inference rules**, etc. can be represented as MMT constants. In particular, the type of a theorem represents the asserted formula and its definiens the proof (as in the constant `example` of Ex. 2). In that case, the typing relation represents the correctness of proofs.

This is flexible enough to represent the tactic-based proofs used in interactive theorem provers: For example, we can declare an untyped constant `auto` and use it as a proof. In that case, the implementation of the typing judgment would include the theorem proving necessary to determine whether `auto` can be accepted as a proof.

3 Generic User Interface Functionality

3.1 Basic Features

We use the phrase "basic features" to group together features that are relatively easy to realize on top of the JMMT abstraction layer. Most of these features make use of existing JEDIT UI functionality, specifically the JEDIT plugins SideKick, ErrorList, and Hyperlinks. This is not, however, meant to imply that these features are inherently easy to realize. Instead, this easiness positively evaluates our design.

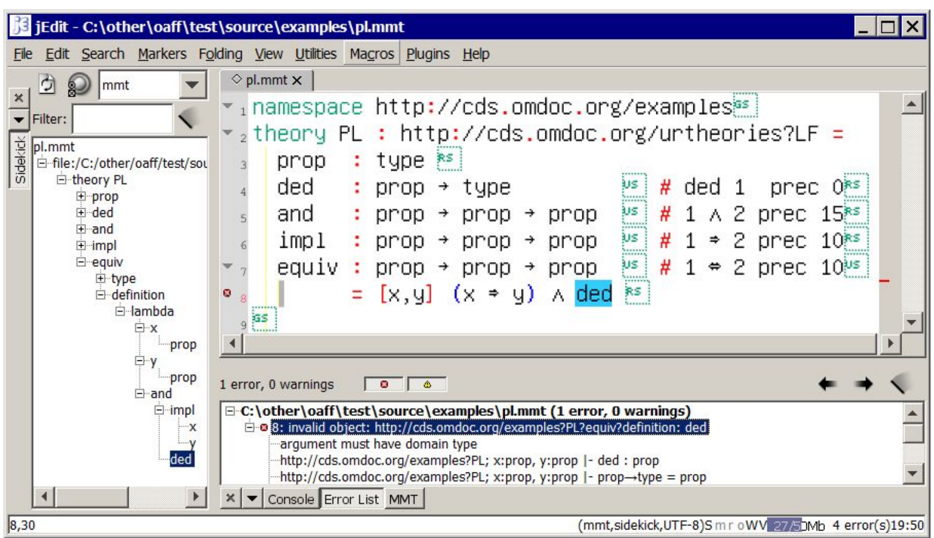


Fig. 4. JMMT Instantiated with Our LF Kernel

Abstract Syntax Display A dockable subwindow displays the abstract syntax tree (AST) of the current buffer. This is shown on the left of Fig. 4, where we define the theory from Ex. 2 in LF. (Here, the green parts are the keywords and separators of our structure parser (see Sect. 4.1.)

Users can navigate from a buffer position to the respective node in the AST and vice versa, as shown in the jointly selected occurrence of `ded`. This is achieved by using **source references**: Every node in the AST points to the region it covers in the buffer. An example for the strength of source references in the AST is that it took the author < 15 minutes to code the follwoing immensely useful feature: Double-clicking on an operator selects the smallest sub-term containing it.

The AST also includes information that is not present in the source but inferred by the kernel. This can be seen in the types `prop` of `x` and `y`, which occur in the AST but not in the source. Similarly, all inferred implicit arguments occur in the AST.

Error Highlighting All errors returned by the kernel are displayed in a dockable window and via line markers. This is shown at the bottom of Fig. 4. There the kernel detected an example typing error in the definition of the equivalence connective: `ded` is used instead of $y \Rightarrow x$. Clicking on the error message resulted in the selection of this ill-typed sub-term.

The error message also includes the kernel's log messages from the branch of the derivation that led to the error. In this case, the typing judgment `ded : prop` failed.

Note that the AST display is robust against errors. Firstly, parsing of the ill-typed term succeeded so that the AST can be displayed. Secondly, even though type-checking of the term failed, the type-checker returned a partially checked term. We can see that in the inferred types of the bound variables `x` and `y`, which are displayed correctly in the AST even though they occur in an overall ill-typed term. This is very important because UI support is needed especially when fixing errors.

Tooltips and Hyperlinking For all occurrences of variables and constants, jMmt displays their types as tooltips. This includes the inferred types of bound variables and the inferred implicit arguments of constants.

Moreover, users can navigate from every occurrence of a constant to its declaration. Because the Mmt project manager (see Sect. 3.4) serializes the AST of every source file to disk, this includes declarations in files or projects that are not currently open in the IDE.

3.2 Auto-Completion and Proof Hints

jMmt provides a basic auto-completion feature (realized based on SideKick). Using the AST, it determines the current theory and context at the cursor position and suggests identifiers that are in scope.

Moreover, we realize a general hinting feature as follows. We declare a additional Mmt constant and implement a typing rule for it such that the term $\langle\!\langle E \rangle\!\rangle$ is inferred to have type E. Both users and plugins can use such terms to represent a missing term of type E (e.g., an open subgoal). In that case, JMMT is aware of the expected type at the cursor position and can provide better auto-completion support.

Example 3. In our kernel for LF, we implement a completion rule: If JMMT asks for completions for the open goal $\langle\!\langle E \rangle\!\rangle$, this rule finds all constants or variables c of type $\{x_1\} \ldots \{x_m\} T_1 \to \ldots \to T_n \to T$ such that T unifies with E via substitution σ. If a completion is chosen by the user, the rule inserts $c\, \sigma(x_1)\, \ldots\, \sigma(x_m)\, \langle\!\langle \sigma(T_1) \rangle\!\rangle \ldots \langle\!\langle \sigma(T_n) \rangle\!\rangle$.

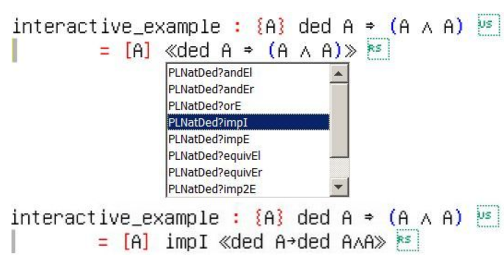


Fig. 5. Proof Hints

In the upper part of Fig. 5, the user is about to choose implication introduction towards proving the example theorem from Ex. 2. Our rule returns $\texttt{impI}\, A\, (A \wedge A)\, \langle\!\langle \texttt{ded}\, A \to \texttt{ded}\, A \wedge A \rangle\!\rangle$. The bottom part shows the situation afterwards (where the implicit arguments have been removed when inserting the hint into the source).

Using a second rule to introduce λ-terms, the whole example proof can be completed using repeated auto-completion.

More generally, any tactic, decision procedure, or proof hinting algorithm can be realized as an auto-completion in this way.

3.3 Interactive Type Inference

If a subterm is selected, JMMT queries the kernel for its type and displays it as a tooltip. This can be seen in Fig. 6 where the user selected a sub-term of the proof of the theorem from Ex. 2.

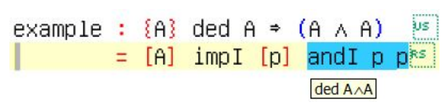


Fig. 6. Inferred Type as a Tooltip

3.4 Project Management and HTML View

MMT provides a notion of projects [HIJ+11] modeled after projects in software engineering. Among other things, this includes a build tool that calls the kernel on all files in a project and caches the resulting ASTs to disk using the OMDoc XML format [Koh06]. Therefore, JMMT can access all dependencies of the current source file without them being processed during the current session.

The MMT build tool can also produce a documentation website for the project, which includes a hierarchical and graph-based project browser. For example, Fig. 7 shows the theorem from Ex. 2 as rendered in a web browser.

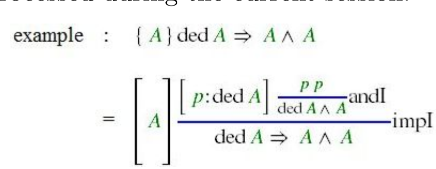


Fig. 7. Generated Web Page

Moreover, JMMT automatically starts a web server, through which the project pages can be browsed interactively. Interactive features include selecting or folding sub-terms (as in Fig. 8) or showing/hiding any inferred part of the term. The user can also control JMMT via the web browser; in particular, by clicking on a declaration in the web browser, the user can open it in JEDIT.

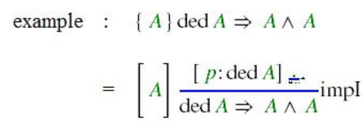


Fig. 8. Folding

3.5 Relational Navigation

The MMT build tool also produces a relational index that stores the ABox of the MMT ontology. For example, atomic relations in the index are the import-relation between theories and the occurs-in relation between parsing units. MMT includes a query language over this ontology [Rab12], which can, e.g., take the transitive closure of a relation or its restriction to theorems.

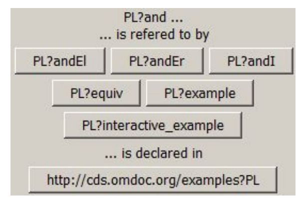


Fig. 9. Navigation

A particularly useful application in JMMT is relational navigation: It displays one navigation button for every declaration that is related to the current one. Clicking a button navigates to that declaration. This permits very fast navigation across a project. Fig. 9 shows for example all constants that refer to `and`.

3.6 Search

The MMT build tool also creates an index of all terms in a project. This index can be read by the MathWebSearch tool [KŞ06], which uses substitution tree indexing to make the set of terms available for fast searching. If an instance of MathWebSearch is running, it can be registered with JMMT, which then provides a search interface for the project.

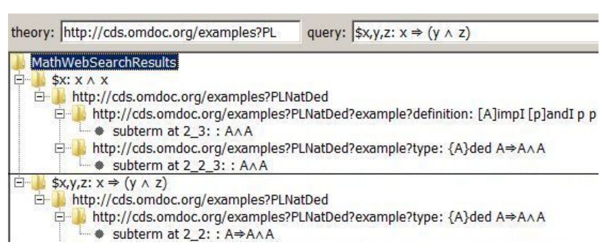


Fig. 10. Display of Search Results

In Fig. 10, JMMT displays the results of two search queries, where only results from the running example are shown. A search query is of the form $\$x_1, \ldots, x_n : E(x_1, \ldots, E_n)$ and returns all terms that unify with $E(x_1, \ldots, E_n)$. The first query asked for all terms of the form $x \wedge x$ for arbitrary x, and the second for all terms of the form $x \Rightarrow (y \wedge z)$.

Note that the results include hits in the inferred parts of the term: For example, the term $A \wedge A$ is found in the definiens $[A]$ `impI` $[p]$ `andI` $p\,p$ because it occurs in the implicit arguments of `impI`.

Clicking on a result opens the containing file in jEdit and selects the respective sub-term.

3.7 Change Management

The type checking of terms is typically the most expensive part of checking a file because it can involve theorem proving or computation. Therefore, it is desirable to recheck terms only when necessary.

$$\begin{tikzcd} t & t' & \\ \text{string} & \text{string} & \xrightarrow{\text{depends on}} \\ \text{parsed} \arrow[u] & \text{parsed} \arrow[u] & \\ \text{validated} \arrow[u] & \text{validated} \arrow[l] \arrow[u] & \cdots \arrow[l] \end{tikzcd}$$

Therefore, Mmt maintains a two-dimensional dependency graph for all terms in a project as indicated on the right. In the vertical dimension, every term is maintained in three forms: the string in the source file, the result of parsing it, and the result of validating it. In the horizontal dimension, Mmt maintains a logical dependency relation between terms.

These horizontal dependencies are known from software engineering, where IDEs recompile a source file only when necessary. For logics, it is important to use a more fine-granular dependency relation. For example, [AMU12] extract fine-granular dependencies for Coq and Mizar. We use terms as the granularity level: If the validity of a term is used while checking another one, this creates a horizontal dependency.

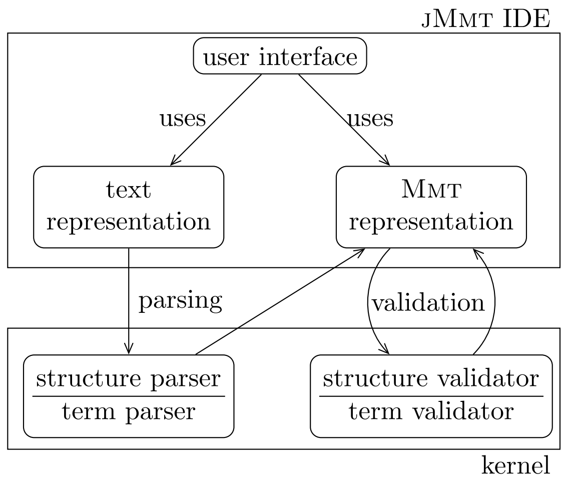
For example, consider two theorems $c : A = p$ and $d : B = q$ such that d uses c. Then type checking the proof of d, i.e., q, has to look up the formula asserted by c, i.e., A. Thus, we obtain a horizontal dependency from q to A. Let us now assume that we make a change in p without changing A, e.g., we might fix an error in the proof.

Without change management, jMmt would have to ask the kernel to recheck the whole buffer and then replace the old AST with the new one returned by the kernel. This may even require rechecking multiple files (which is what caused the delays in the L4.verified project we cited in Sect. 1).

As we will describe in Sect. 4, jMmt splits the kernel conceptually into several components such that every term can be parsed and validated individually. This permits jMmt to patch the AST by calling the kernel only on the necessary terms. More precisely, a term is reparsed only if its string representation has changed. If this causes the parsed representation to change, the term is revalidated as well. For example, this is not the case if we only changed a comment or the formatting of a term. Finally, if a term has to be revalidated and the resulting validated representation has changed, we also revalidate all terms that horizontally depended on it.

For our two-theorem example, that means that the changes in p do not trigger the expensive revalidation of q. This is crucial to achieve short edit-check cycles when working with multiple theorems at the same time.

4 An Abstraction Layer Between Kernel and UI



An **overview** of our IDE is shown on the right. jMmt maintains both the concrete text source and its abstract Mmt representation, which are used by the UI. It is parametric in operations for parsing and validation, which must be provided by the individual kernels.

The crucial difficulty is how to conceptually split the kernel into separate components. If the kernel appears as a single component (as is typical for read-eval-print kernels), the UI is effectively limited to reporting and querying the kernel's state. But if we specify too many components, it puts too many constraints on kernel design. For our abstraction layer, we choose a trade-off that splits the kernel into 2×2 components as shown on the right.

2×2 components	Parsing	Validation
Structure	Sect. 4.1	Sect. 4.2
Terms	Sect. 4.3	Sect. 4.4

Along one dimension, we distinguish the **structure** and **term** levels. The structure level comprises the names of the theory and constant declarations. This corresponds to the OCaml toplevel in HOL Light [Har96] or to the outer syntax of Isabelle. The term level comprises the type and definiens of the constant declarations. This corresponds to the HOL Light formula parser or to the inner syntax of Isabelle.

Along the second dimension, we distinguish **parsing** and **validation**. The former produces an abstract syntax tree that closely resembles the source text; the latter refines this AST using advanced operations such as type checking, theorem proving, or computation. Notably, we use Mmt to represent both the parsed and the validated representation, i.e., validation is a transformation from Mmt syntax to Mmt syntax.

A major **motivation** of this separation is to isolate the validation of terms. This has three reasons. Firstly, this is the most expensive phase. Therefore, as we described in the change management section, isolating this component permits revalidating terms as rarely as possible. Secondly, almost all errors that users have to fix are detected during term validation, whereas the other 3 components succeed most of the time. By isolating term validation, we can provide substantial UI features, which are especially helpful when term validation does not succeed. Moreover, many UI features do not depend on term validation such as auto-completion or search.

Thirdly, it opens up an alternative way to connect an existing kernel with our UI in those cases where it is not possible to have the kernel expose the AST. Then it can be feasible to reimplement the other 3 components from scratch, but it will usually be impossible to reimplement term validation.

We explain the 2×2 components in the following, using our kernel for LF as a running example. To simplify the description below, we point out two **general aspects** globally that apply to all 2×2 components. Firstly, to maximize **extensibility**, both individual kernels as well as extensions of our example kernel are supplied to jMMT as plugins. Thus, kernels can be developed and used without rebuilding jMMT itself.

Secondly, all 2×2 components can **report errors** to jMMT and can always recover from errors by returning partial or no results. For example, term parsing may return a term with some unparsed sub-terms, for which errors are reported.

4.1 Structure Parsing

Structure parsing takes a source document and returns an abstract syntax tree (AST) in the MMT language. Alternatively, the AST can be returned as XML (using the OMDoc format [Koh06]), which is useful if kernels are implemented in other programming languages. The terms occurring in the source can remain unparsed: jMMT produces a parsing unit for each unparsed object, which is passed on to the term parsing component.

Example 4 (A Keyword-Based Parser). We implement a straightforward structure parser based on keywords. First it splits a file into declarations based on a few reserved characters and computes their source references. Then the parsing of each declaration is relegated to an appropriate plugin, which is selected based on the keyword.

We choose the ASCII characters 28-31 as separators (the green boxes in Fig. 4). Thus, existing term parsers can be reused easily no matter what special characters are used in the terms.

Our structure parser works at the MMT level, and we do not have to customize it for LF at all.

4.2 Structure Validation

Structure validation is a fixed algorithm, that implements the semantics of MMT, in particular the module system [RK13]. Thus, it is not necessary to change the implementation in specific kernels.

Because MMT is parametric in the underlying type system, the terms cannot be validated generically. Instead, structure validation produces one validation unit for each term and passes them on the term validator. In particular, the constant declaration $c : A = t$ in a theory T gives rise to two validation units: $c_{\mathtt{type}}$ validates the judgment $\vdash_T A\ \mathtt{TYPE}$, and $c_{\mathtt{def}}$ validates the judgment $\vdash_T t : A$.

Even though this algorithm is fixed, it is still extensible because it implements the structure level extension mechanism we presented in [HKR12]. This is general enough to express, e.g., all toplevel declarations of Mizar (as we show in [IKRU13]) or the type definition principle of HOL systems.

4.3 Term Parsing

A **parsing unit** is a tuple of

- a string S that is to be parsed into a term,
- the source reference R of S (if returned by the structure parser),
- the theory T in which S occurs.

Term parsing takes as input a parsing unit and returns an MMT term t in context Γ. The role of Γ is to declare meta-variables for unknown sub-terms that are to be found during validation. For example, the term parser may generate a meta-variable for the omitted type of a bound variable. More generally, meta-variables can be used to represent proof that still have be found and inserted.

In MMT, each sub-term may carry its own source reference. Thus, the term parsing algorithm can return fine-granular source references for all sub-terms. The MMT implementation makes sure that source references are ignored when programs inspect terms (e.g., to check identity or for pattern-matching), but are carried along when programs transform terms by substitution or rewriting.

Example 5 (A Notation-Based Parser). JMMT includes a term parser, which uses the MMT notations of all constants that are imported into T. It returns source references for every sub-term and inserts fresh meta-variables for omitted variable types and implicit arguments. (An argument position is considered implicit if it is not mentioned in the notation.)

In fact, our term parser is a bit more general: It also supports notations with precedences, argument sequences, and bound variable sequences.

Our term parser works at the MMT level, and we do not have to customize it for LF – we only have to import the theory from Ex. 1.

4.4 Term Validation

A **validation unit** consists of

- a context Γ (as returned by the term parser),
- a judgment (as in Fig. 3) whose terms may use the free variables of Γ.

A term validator takes a validation unit; it verifies the judgment and returns a substitution for the variables in Γ, i.e., it finds the unknown parts of the term. This has the advantage that the validated term has the same structure as the parsed term so that the UI can present both at once. Alternatively, it may be reasonable that validation returns a whole new term, e.g., its normal form.

It is important that we validate at a per-term basis and not at a per-declaration basis. As described above, our structure validator produces two independent validation units $c_{\texttt{type}}$ and $c_{\texttt{def}}$ for a declaration $c : A = t$. For most type theories, $c_{\texttt{def}}$ implies $c_{\texttt{type}}$ so that validating both seems redundant. But this design is crucial for change management, as we have seen in Sect. 3.7.

Example 6 (A Rule-Based Validator). We have developed a term validator for MMT terms. It reconstructs the unknown variables from Γ by using an algorithm similar to the one of Twelf [PS99]. Moreover, type checking can be performed modulo rewriting according to user-declared equalities (which are assumed to be confluent). This term validator is described in detail in [Rab14].

In order to maximize reuse, we only implement certain structural rules like congruence and lookup. We use the heads of complex terms to relegate to language-specific rules, which can be provided by other plugins.

Example 7 (A Term Validator for LF). We instantiate the term validator from Ex. 6 with LF by providing LF-specific rules. These are inference rules that infer the types of terms with head `type`, `lambda`, `Pi`, or `apply`; a checking rule for judgments of the form $\vdash_T f : \{x : A\}B$; an equality rule for judgments of the form $\vdash_T t = t' : \{x : A\}B$; a rewrite rule to turn $A \to B$ into $\{\_ : A\}B$ and a rewrite rule for β-reducing terms of the form $([x : A]t)a$.

Moreover, we implement a solution rule, which is used to determine the values of the meta-variables in Γ: It applies to judgments of the form $\vdash_T X\ x_1\ \ldots\ n_n = E : A$, where X is a meta-variable from Γ, and the x_i are distinct bound variables, and solves X as $[x_1, \ldots, x_n]E$.

Notably, in terms of lines of code, the LF-specific rules from Ex. 7, the generic term validator from Ex. 6, our plugin for JEDIT together with MMT, and the whole JEDIT code are roughly related like $1 : 5 : 500 : 5000$. This indicates the potential synergies of logic-independent IDEs.

5 Conclusion

We have developed JMMT, a generic user interface (UI) for formal logics, by combining the MMT representation language and the JEDIT text editor. Our UI differs from most current ones by employing a high-level abstraction layer that permits a heavyweight UI. This permits a number of advanced features including auto-completion, error highlighting, search, navigation, and change management. Future work can benefit from the existing abstraction layer and extend these features substantially.

But we have to pay for this feature richness: Our IDE cannot be directly used with existing logic implementations. Even though these largely include the algorithms that our abstraction layer assumes, these algorithms are not easy to expose. Depending on the individual system, this may require even partial redesigns.

Among IDEs for *existing* logic implementations, our closest relative is Isabelle/JEDIT [Wen12], which was developed independently of ours. Both systems share the idea of a heavyweight UI based on JEDIT as well as the basic features (Sect. 3.1). The main difference is that JMMT is a JEDIT plugin, whereas Isabelle/JEDIT uses a fork of JEDIT that is integrated with the Isabelle system. Consequently, Isabelle/JEDIT can offer additional Isabelle-specific features, in particular in regard to theorem proving. jMMT, on the other hand, offers several advanced features and focuses on logic-independence and extensibility.

New logic implementations can easily reuse jMMT by providing plugins that instantiate our abstraction layer. We demonstrated that by developing a mature implementation for the logical framework LF from scratch. Moreover, new logic implementations can customize our LF implementation by adding operators, notations, or typing rules – in that case, developers obtain the double benefit of a strong UI at very small implementation cost. Additionally, they can benefit from the existing MMT infrastructure, in particular the module system.

References

ABMU11. J. Alama, K. Brink, L. Mamane, and J. Urban. Large formal wikis: Issues and solutions. In J. Davenport, W. Farmer, J. Urban, and F. Rabe, editors, *Intelligent Computer Mathematics*, pages 133–148. Springer, 2011.

AMU12. J. Alama, L. Mamane, and J. Urban. Dependencies in Formal Mathematics: Applications and Extraction for Coq and Mizar. In J. Jeuring, J. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *Intelligent Computer Mathematics*, pages 1–16. Springer, 2012.

Asp00. D. Aspinall. Proof General: A Generic Tool for Proof Development. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for Construction and Analysis of Systems*, pages 38–42. Springer, 2000.

BDKK12. T. Bourke, M. Daum, G. Klein, and R. Kolanski. Challenges and Experiences in Managing Large-Scale Proofs. In J. Jeuring, J. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *Intelligent Computer Mathematics*, pages 32–48. Springer, 2012.

Ecl. Eclipse IDE. `http://www.eclipse.org/`.

GAA+13. G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 163–179, 2013.

Har96. J. Harrison. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

HHP93. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

HIJ+11. F. Horozal, A. Iacob, C. Jucovschi, M. Kohlhase, and F. Rabe. Combining Source, Content, Presentation, Narration, and Relational Representation. In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 212–227. Springer, 2011.

HKR12. F. Horozal, M. Kohlhase, and F. Rabe. Extending MKM Formats at the Statement Level. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 64–79. Springer, 2012.

IKRU13. M. Iancu, M. Kohlhase, F. Rabe, and J. Urban. The Mizar Mathematical Library in OMDoc: Translation and Applications. *Journal of Automated Reasoning*, 50(2):191–202, 2013.

IR12. M. Iancu and F. Rabe. (Work-in-Progress) An MMT-Based User-Interface. In *Workshop on User Interfaces for Theorem Provers*, 2012.

jEd. jEdit: Programmer's Text Editor. `http://www.jedit.org/`.

Kal07. C. Kaliszyk. Web Interfaces for Proof Assistants. In *User Interfaces for Theorem Provers*, pages 49–61, 2007.

Koh06. M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer, 2006.

KŞ06. M. Kohlhase and I. Şucan. A Search Engine for Mathematical Formulae. In T. Ida, J. Calmet, and D. Wang, editors, *Artificial Intelligence and Symbolic Computation*, pages 241–253. Springer, 2006.

KU13. C. Kaliszyk and J. Urban. Automated Reasoning Service for HOL Light. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 120–135. Springer, 2013.

Nor05. U. Norell. The Agda WiKi, 2005. `http://wiki.portal.chalmers.se/agda`.

Pau94. L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

PS99. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.

Rab12. F. Rabe. A Query Language for Formal Mathematical Libraries. In J. Campbell, J. Carette, G. Dos Reis, J. Jeuring, P. Sojka, V. Sorge, and M. Wenzel, editors, *Intelligent Computer Mathematics*, pages 142–157. Springer, 2012.

Rab13. F. Rabe. The MMT API: A Generic MKM System. In J. Carette, D. Aspinall, C. Lange, P. Sojka, and W. Windsteiger, editors, *Intelligent Computer Mathematics*, pages 339–343. Springer, 2013.

Rab14. F. Rabe. A Generic Type Theory. see `http://kwarc.info/frabe/Research/rabe_mmttypetheory_14.pdf`, 2014.

RK13. F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

The14. The Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2014.

Wen12. M. Wenzel. Isabelle/jEdit - A Prover IDE within the PIDE Framework. In J. Jeuring, J. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *Intelligent Computer Mathematics*, pages 468–471, 2012.