Knowledge management for Twelf

Project Report

Alin Iacob

February 17, 2011

1 Introduction

The LATIN (Logic Atlas and Integrator) project [LAT] focuses on developing a foundationally unconstrained framework for knowledge representation that allows to represent the meta-theoretic foundations of the mathematical knowledge in the same format and to interlink the foundations at the meta-logical level.

Currently, the project consists in building an Atlas of Logics used in automated reasoning, mathematics, and software engineering concentrating on paradigmatic logics from first-order logic (TPTP, CASL, and Mizar), higher-order logics (PVS, Isabelle/HOL, HasCASL), and logical frameworks (LF and Isabelle). In order to provide a comprehensive logical framework, LATIN extends the Twelf language with a module system inspired by MMT [RK10].

As the Atlas of Logics grows in size and complexity, the need for adequate knowledge management becomes apparent. The next section presents a simplified version of the Twelf language with the MMT extensions, and sections 3 and 4 describe two kinds of changes that allow for easier understanding of the Atlas and increase sustainability of a large code base.[TIS]

In the last part of the report we present the details of the parser implementation (section 6) and the changes to the propositional logic tree (section 7)

2 Preliminaries

A file in the Atlas has the extension `.elf` and is a collection of *signatures* and *views*.

1. A *signature* is a list of *symbol declarations* and *structures*

   ```
   %sig signature-name = {
     (symbol-declaration | structure-declaration)*
   }
   ```

   where `(A|B)*` indicates 0 or more occurrences of A or B.

   (a) *symbol declarations* are the core of the original Twelf language. They introduce new well-typed symbols.

   ```
   symbol-name [: symbol-type] [= symbol-definition].
   ```

where square brackets indicate that their contents is optional. At least one of {symbol-type, symbol-definition} must be given for each symbol. Each of {symbol-type, symbol-definition} is a well-formed expression with atoms declared earlier.

(b) *structures* have the (simplified) form

```
%struct struct-name : domain = {
   (symbol-name := expression.)*
}.
```

A structure is equivalent to redeclaring all the symbols from the domain signature into the current signature, possibly with different definitions, as specified by the `expressions`.

The following restrictions are in place:

- `domain` is an already declared signature name.
- the set of `symbol-names` is a subset of the names without definiens declared in `domain`. The symbols from the domain signature which are not assigned to anything in the `structure` are simply imported as-is into the current signature.
- each `expression` is formed using only symbols declared earlier in the parent signature.

A symbol available as `symbol-name` in the `domain` can be used within expressions in the current signature as `struct-name.symbol-name`.

(c) An *import* is syntactic sugar for a structure declaration with empty body, where every symbol from the domain is imported as-is into the current signature.

```
%include domain.
```

A symbol available as `symbol-name` in the `domain` can be used within expressions in the current signature as `domain.symbol-name`.

2. A *view* between two signatures has the (simplified) form

```
%view view-name : domain -> codomain = {
   (symbol-name := expression.)*
}.
```

The following restrictions are in place:

- `domain` and `codomain` are already declared signature names.
- the `symbol-names` are precisely the names without definiens declared in `domain`.
- each `expression` is formed using only symbols from `codomain`.

Unlike structures, a *view* does not add symbols to a signature. Its role is independent from those of the structures. If we regard signatures as mathematical theories, where axioms are encoded as symbols according to the Curry-Howard isomorphism, then a *view* encodes an `is-a` relation between `codomain` and `domain`.

Once we have a view from `domain` to `codomain`, every expression over the `domain` can be translated (via structural induction, using the view assignments for the atoms) into an expression over the `codomain`. Mathematically, any constructions or theorems of the `domain` theory can be automatically translated to constructions and theorems of the `codomain`.

As signatures import symbols via `%struct` and `%include` statements, it is fitting that views and structure bodies have a way to directly map these inclusions to some existing morphism (i.e. composition of views and/or structures), instead of having to map each induced symbol to an expression. Morphisms are thus defined as: `(struct-name | view-name)*`, where concatenation means composition.

Accordingly, one can use the construct

```
%struct struct-name := morphism.
```

inside both views and `%struct` declarations, and

```
%include view-name.
```

inside views.

For examples and more explanations, see Section 3 in [RS].

3 Shallow improvements

3.1 Problems with existing knowledge management in the Logic Atlas

The Logic Atlas is organized via a standard file system directory tree. Each component (or module) resides in a file, while the hierarchy of organization is encoded via the directory structure of the project. Each resource in the atlas is assigned a global URI[1] by an external convention (it coincides with the URL on a university server on which an official copy resides). Once the Atlas is checked out via SVN, the physical location becomes file-system specific[2] and does not coincide with the (globally-unique) URI any more.

The current structure of the Logical Atlas consists of a multi-tier hierarchy, where:

1. On the top level we distinguish between the different scientific domains that define their own, interrelated system of logics and theories. At the moment of writing, these fields are `logics`, `math`, `category_theory`, `set_theories`, `type_theories`, and `translations`. A problem with the current organization at this level is the inclusion of non-atlas content into the repository, namely `admin`, `projects` and `unsorted` directories. Ideally, those should be kept in a separate repository as they violate the hierarchy principles and integrity of the LATIN library.

2. The next level (depth 2) models a classification of subdomains, for the specific field given at depth 1. For example, for `logics` there exist `first-order` and `higher-order` subclasses, while for `math` currently there are `algebra` and `relations`.

   However, there is no limitation on the number of intermediate levels that can be modeled by this directory tree - while the `math` field has a single level of subdomains, the `logics` field has two. Naturally, the depth of the intermediate directories will grow alongside the refinement of granularity of the library.

3. The leaf level contains the logics (or fragments thereof) as `.elf` files, as explained in Section 2. Each of those fragments is, however, not self-contained, but has dependencies in different files from the same logic, in other logics, and even in other depth 1 folders of the hierarchy - set theories and type theories.

[1] such as `http://cds.omdoc.org/logics/first-order/syntax/base.elf`
[2] such as `file:/c:/courses/twelf/logics/first-order/syntax/base.elf`

3.2 Alleviatory measures

3.2.1 Metadata in comments

In order to facilitate a management-friendly workflow of creating, maintaing and reorganizing modules, we propose the practice of adding commentary headers to every physical file, containing the following information:

- List of authors
- File license
- Description of the file purpose
- Anything else the author deems important

Comments like this, containing useful metadata, will be enclosed by `%* ... *%` instead of the usual multi-line `%{ ... }%` or single-line `%%`. Each property like author or license will be written as a key-value pair:

```
@key_name  value_string
```

where `key_name` is user-definable, in order to accommodate for as much diversity as possible.

Furthermore, each module shall have a similar comment before. The first 0 or more lines of the comment contain key-value pairs. Starting from the next line, the first sentence is considered to be a "short description" of the module that follows and, together with the rest of the sentences (if they exist), it forms the "long description". Similar to Javadoc, this distinction allows code editors to show both small captions with information about a module and larger help boxes if needed.

3.2.2 Over-modularity

On level 3 and 4 of the logic graph, most logics have a folder each for syntax, proof theory, model theory and soundness. Furthermore, each folder contains many files, most of which have a small number of modules, but which are highly dependent on each other and on other files and folders. This has proved to be exceedingly confusing for beginners trying to understand the structure of the Atlas, or even trying to understand the dependencies of a single module, which can already be as many as 8 levels deep.

An improvement for understanding is to group the syntax modules in one file, the proof theory in another, etc. Section 7 shows how this has been done for Propositional logic, greatly reducing the complexity of the dependency graph, while keeping the readability of the code.

3.2.3 Under-modularity

In some places within the logic graph, two logics are implemented in the same folder, each of them split into several files, making it difficult to see at a glance which files are part of which logic. An example of this is Minimal propositional logic and "normal" propositional logic. In this case, we have split the two logics into different folders. Nevertheless, this is not always the best approach, as seen with Intuitionistic propositional logic and Classical propositional logic, the latter of which is a small extension of the first. Both of them belong in the same place, since one is only defined in terms of another.

Same problems are often encountered with two model theories mixed in one folder. Since they belong to the same logic, they should stay there, but in separate subfolders, as they do not depend on each other.

4 Deep improvements

4.1 Read statements versus namespaces

4.1.1 Problem

`%read` declarations make all the imported module names visible in the entire file scope, which leads to possible name clashes, since modules written in different parts of the graph may have the same name. The existing approach for avoiding this is to append prefixes and/or suffixes to each module name, in order to make them as globally unique as possible, so as to avoid any possible clash when someone uses them in conjunction with another file. The result is names that are too long and artificial, with modules like `Forall`, `ForallPF`, `ForallPFExt` or `Bool` vs `Boolean`. In the future, as the Atlas expands, it becomes unfeasible for authors to invent unique names for their modules, since they would have to look through the entire repository. Even if they have tools to automate this, the resulting long names will make the code unreadable.

4.1.2 Solution

Namespaces offer a solution: each file shall have a "current" namespace declaration immediately after the initial comment:

```
%namespace "absolute namespace uri"
```

This namespace is required to be globally unique URI and for the time being, this is accomplished by having it reside in the subdirectory tree of `http://cds.omdoc.org/`, where it will presumably be uploaded via SVN. Documents can only reference each other by knowing this URI.

The `%read` statements are replaced by

```
%namespace alias = "foreign uri"
```

where `alias` is an identifier and `uri` is a relative uri based on the last "current" namespace declaration. From the point of the alias declaration onwards, a module `A` declared within the scope of the "foreign uri" `U` can be referred to as `U.A`.

Namespaces are already defined conceptually, but there is no tool to support them yet.

4.2 A catalogue service

4.2.1 Problem

In the current setup, the namespace alias declarations present a conceptual dead-end for automated knowledge management, because the author-defined namespace hierarchy does not necessarily correspond to the physical, system-understandable file URLs. The synchronization of the two types of orderings can be achieved only by tedious efforts, which is a strong motivation for automatizing it.

To make the problem worse, each file can contain references to any number of other namespaces, allowing for various kinds of inconsistencies, such as circular imports and hierarchy violations. Detection can be done with graph algorithms.

A catalogue infrastructure that translates from URI (namespaces) to URL (filenames) has been designed, but not implemeneted.

- URIs are logical references. The Twelf files contain only logical references, not physical references
- URLs are physical references. Such a catalogue cannot be in the SVN repository, hence every user who checks out from the server needs to set their own catalogue.

We provide tool support to perform this translation and to create the catalogue.

4.2.2 Solution

We implemented a parser called `crawler` which harvests the information given by `%namespace`, `%include`, `%struct`, domains and codomains of views and generates an XML file that offers, for each module, its URI, URL and a list of dependencies given as URIs. All the URI and URLs are absolute. Thus, the generated file can function as a catalogue translating URIs to URLs.

The parser also gathers metadata information from the comments described in 3, for the files as well as for the modules.

While the parser is described extensively in section 6, below are a few example of Twelf input with the corresponding XML output.

Example 1. The Twelf+MMT code present from file `L/classic/proof_theory.elf`

```
%*
   @author Fulya Horozal, Florian Rabe
   @license LATIN
   Signature for proof theory of Classical propositional logic
*%

%namespace "I/classic/proof_theory.elf".
%namespace syntax = "syntax.elf".

%* true introduction *%
%sig Truth = {
  %include syntax.Truth    %open.
  trueI  : ded true.
}.

%* false elimination *%
%sig Falsity = {
  %include syntax.Falsity  %open.
  falseE : ded false -> {A} ded A.
}.
...
```

is mapped to

```
<?xml version="1.0" encoding="UTF-8"?>
<skeleton>

   <document URL="L/classic/proof_theory.elf">
      <meta>
```

```
         <author>Fulya Horozal, Florian Rabe</author>
         <license>LATIN</license>
         <short>   Signature for proof theory of Classical
                       propositional logic</short>
      </meta>
      <dependency URI="I/classic/syntax.elf" />
   </document>

   <sig URI="I/classic/proof_theory.elf?Truth"
          URL="L/classic/proof_theory.elf#position(10;0;13;1)">
      <meta>
         <short> true introduction </short>
      </meta>
      <dependency URI="I/classic/syntax.elf?Truth">
   </sig>

   <sig URI="I/classic/proof_theory.elf?Falsity"
          URL="L/classic/proof_theory.elf#position(16;0;19;1)">
      <meta>
         <short> false elimination </short>
      </meta>
      <dependency URI="I/classic/syntax.elf?Falsity">
   </sig>
   ...
</skeleton>
```

In order for the examples to fit in the page, we used the shortcuts

- `I` instead of `http://cds.omdoc.org/logics/propositional` and
- `L` instead of `file:/C:/courses/Twelf/logics/propositional`.

Example 2. The Twelf+MMT code from file `file:/C:/courses/Twelf/logics/meta/bool-zf.elf`

```
%*
   @author Florian Rabe
   @license LATIN
   View from (Booleans in STTIFOL) to (Booleans in ZF)
*%

%namespace "http://cds.omdoc.org/logics/meta/bool-zf.elf".
%namespace bool_sttifol = "bool.elf".
%namespace bool_zf = "../../set_theories/zfc/bool.elf".
%namespace sttifol-zf = "sttifol-zf.elf".

%view Bool-ZF : bool_sttifol.Bool -> bool_zf.Boolean = {
   %include sttifol-zf.STTIFOLEQ-ZF.
   bool' := cbool.
   0     := Ł.
   1     := Ł.
   cons  := ccons.
   boole := ctnd.
   ...
}.
```

is mapped to

```
<?xml version="1.0" encoding="UTF-8"?>
<skeleton baseURI="http://cds.omdoc.org/logics/meta/" baseURL="">

   <document URL="file:/C:/courses/Twelf/logics/meta/bool-zf.elf">
      <meta>
         <author>Florian Rabe</author>
         <license>LATIN</license>
         <short>   View from (Booleans in STTIFOL) to (Booleans in ZF)</short>
      </meta>
      <dependency URI="http://cds.omdoc.org
                                              /logics/meta/bool.elf" />
      <dependency URI="http://cds.omdoc.org
                                             /set_theories/zfc/bool.elf" />
      <dependency URI="http://cds.omdoc.org
                                             /logics/meta/sttifol-zf.elf" />
   </document>

   <view URI="http://cds.omdoc.org/logics/meta/bool-zf.elf?Bool-ZF"
            URL="file:/C:/courses/Twelf/logics
                             /meta/bool-zf.elf#position(11;0;21;1)">
      <dependency URI="http://cds.omdoc.org/logics
                                        /meta/bool.elf?Bool">
      <dependency URI="http://cds.omdoc.org/set_theories
                                        /zfc/bool.elf?Boolean">
      <dependency URI="http://cds.omdoc.org
                                   /logics/meta/sttifol-zf.elf?STTIFOLEQ-ZF">
   </view>

</skeleton>
```

5 Interface

5.1 Web server

The `crawler.Run` object starts a RESTful[3] web server on localhost.

Command-line arguments

`crawler.Run` accepts the following command-line arguments:

```
crawler.Run (--port port_number)? (location|+pattern|-pattern)*
```

[3] http://en.wikipedia.org/wiki/Representational_State_Transfer#RESTful_web_services

- `--port` *port number*
  The port number on which the server runs. This argument is optional and must be the first. The default port number is 8080 (historically a common choice for web servers operated in non-root mode[4]).

- *location*
  Absolute path to a file or directory in the local file system. The program continuously and recursively crawls these locations and updates the hashes with the information from them.

- *+pattern*
  An inclusion pattern, where *pattern* is a file or directory name, without its path. The star (*) is the only special character and matches any sequence of characters.

- *−pattern*
  An exclusion pattern.

  Inclusion and exclusion patterns affect which files and folders are crawled recursively, starting from the locations provided as command-line arguments and, later, given by web requests:

  1. Folders are crawled if they do not match any exclusion pattern.
  2. Files are crawled if they match at least one inclusion pattern, but no exclusion pattern.
     If no inclusion patterns are provided, only the second condition remains.

Administration web page

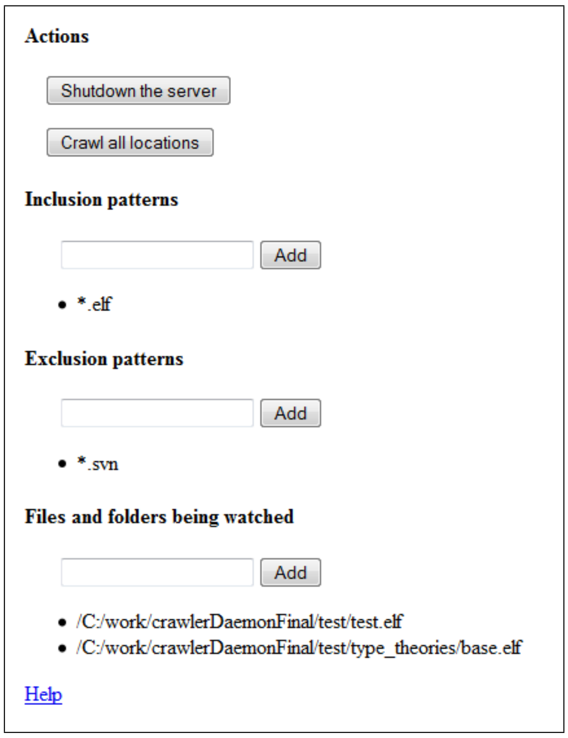
The web server provides a page (Figure 1) where the user can add inclusion/exclusion patterns and locations, and see the existing ones. There is also the possibility to re-crawl all the locations and to shutdown the server.

GET requests and responses

1. **Get metadata as text** `/getMetaText?uri=`*entityUri*

   - Print verbatim the semantic comment associated with an entity. *entityUri* can be either a file path, in which case the document metadata is returned, or the URI of a module, constant or structure declaration. If the entity does not have a semantic comment associated with it, an empty text is returned. If *entityUri* is neither a valid URI, nor a file path already in the list of locations, then an error message is returned.
   - *Example:*
     - Request URL: `http://localhost:8080/getMetaText?uri=C:/modules-lambda.elf`
     - Response:
       ```
       Some orthogonal features of type theories occurring in the lambda cube
          @author Fulya Horozal, Florian Rabe
       ```
   - *Example:*
     - Request URL: `http://localhost:8080/getMetaText?uri=http://cds.omdoc.org/type_theories/modules-lambda.elf?DepFun?@`
     - Response:
       ```
       elimination
       ```

[4]GRC - Port Authority, `http://www.grc.com/port_8080.htm`



- *.elf

- *.svn

- /C:/work/crawlerDaemonFinal/test/test.elf
- /C:/work/crawlerDaemonFinal/test/type_theories/base.elf

Figure 1: Web administration interface (http://localhost:8080/admin)

2. **Get metadata as OMDoc** `/getMeta?uri=`*entityUri*

   - Print the semantic comment associated with an entity, as OMDoc. The comment is parsed as follows: The first line of the comment is the "short" comment, which can be used for example as a hover text in an IDE. Starting from the second line is the "long" comment, which is intended to be an elaboration of the short comment.

     The first line that starts with "@" is a key-value property line. "@" must be followed by a word of non-zero length, which will be the key. The key and value are separated by whitespace, and the value starts at the first non-whitespace character and ends before the line separator. After a key-value property line, all the lines must be key-value properties. The key-value properties may start on the first or second line, i.e. the "short" and "long" comments are optional.

     entityUri can be either a file path, in which case the document metadata is returned, or the URI of a module, constant or structure declaration. If the entity does not have a semantic comment associated with it, an empty text is returned. If *entityUri* is neither a valid URI, nor a file path already in the list of locations, then an error message is returned.

   - *Example:*
     - Request URL: `http://localhost:8080/getMeta?uri=C:/modules-lambda.elf`
     - Response:

       ```
       <metadata>
         <metadatum key="short">
           Some orthogonal features of type theories occurring in the lambda cube
         </metadatum>
         <metadatum key="author">Fulya Horozal, Florian Rabe</metadatum>
       </metadata>
       ```

   - *Example:*
     - Request URL: `http://localhost:8080/getMeta?uri=http://cds.omdoc.org/type_theories/modules-lambda.elf?DepFun?@`
     - Response:

       ```
       <metadata>
         <metadatum key="short">elimination</metadatum>
       </metadata>
       ```

3. **Get text** `/getText?uri=`*entityUri*

   - Print verbatim the text of an entity. *entityUri* can be either a file path, in which case the entire document is printed, or the URI of a module, constant or structure declaration. If *entityUri* is neither a valid URI, nor a file path already in the list of locations, then an error message is returned.

   - *Example:*
     - Request URL: `http://localhost:8080/getText?uri=http://cds.omdoc.org/type_theories/modules-lambda.elf?DepFun?@`
     - Response:

       ```
       @    : scope.exp ( [x : domain.exp A] B x) -> {a : domain.exp A} scope.exp
       ```

4. **Get OMDoc** `/getOmdoc?url=`*path*

   - Print the Omdoc skeleton of a document given by its disk address. *path* must be a file path. If *path* is not a file path already in the list of locations, then an error message is returned.

- *Example:*
  - Request URL: http://localhost:8080/getOmdoc?url=C:/modules-lambda.elf
  - Response:

```
<?xml version="1.0" encoding="UTF-8"?><omdoc
base="http://cds.omdoc.org/type_theories/modules-lambda.elf" xmlns:om="http:
  <metadata>
    <metadatum key="short">
      Some orthogonal features of type theories occurring in the lambda cube
    </metadatum>
    <metadatum key="author">Fulya Horozal, Florian Rabe</metadatum>
  </metadata>
  <theory
  uri="http://cds.omdoc.org/type_theories/modules-lambda.elf?DepFun" name="D
    <metadata>
      <metadatum key="short">generalized dependent abstraction</metadatum>
    </metadata>
    <structure
    ...
```

5. **Get namespaces** `/getNSIntroduced?url=`*path*

- Print the namespaces introduced by the document given by its disk address. *path* must be a file path. If *path* is not a file path already in the list of locations, then an error message is returned.
- *Example:*
  - Request URL: http://localhost:8080/getNSIntroduced?url=C:/modules-lambda.elf
  - Response:

```
http://cds.omdoc.org/type_theories/modules-lambda.elf
```

6. **Get dependencies** `/getDependencies?uri=`*entityUri*

- Print the URIs of the dependencies of an entity, one per line. *entityUri* must be the URI of a module, constant or structure declaration. If *entityUri* is not a valid URI, then an error message is returned. The dependencies of a module are the imported modules, the domains of the structs declared and, for views, the domain and codomain. The only dependency of a structure declaration is its domain, and constant declarations have no dependencies.
- *Example:*
  - Request URL: http://localhost:8080/getDependencies?uri=http://cds.omdoc.org/type_theories/modules-lambda.elf?DepTypes
  - Response:

```
http://cds.omdoc.org/type_theories/base.elf?KindsTypesTerms
http://cds.omdoc.org/type_theories/modules-lambda.elf?DepFun
```

7. **Get position** `/getPosition?uri=`*entityUri*

- Print the file path and position within the file of an entity. All the position information is encoded in an URL consisting of the file path, followed by a fragment part of the form

`#((`*firstLine*`,`*firstCol*`),(`*secondLine*`,`*secondCol*`)).`

entityUri must be the URI of a module, constant or structure declaration. If *entityUri* is not a valid URI, then an error message is returned.

- *Example:*
  - Request URL: `http://localhost:8080/getPosition?uri=http://cds.omdoc.org/type_theories/modules-lambda.elf?DepTypes`
  - Response:
    `/C:/modules-lambda.elf#((40,0),(46,1))`

8. **Print capabilities** `/`, `/help`
   Print the list of requests accepted by the server and their meaning.

In addition to these requests conforming to the REST philosophy, the server accepts a number of requests which modify the state of the server. All of them redirect to the admin page and any possible errors are logged to standard output.

1. Re-crawl all locations `/crawlAll`
   Crawl again all locations recursively. Only the files that have changed since the last crawl are taken into consideration. This request is normally not needed, since the server crawls all locations periodically.
2. Add a location `/admin?addLocation=`*path*
   Add a disk location to the list of watched locations. This operation is equivalent to adding a location via program arguments when starting the server.
3. Add an inclusion pattern `/admin?addInclusion=`*pattern*
   This operation is equivalent to adding an inclusion pattern via program arguments when starting the server.
4. Add an exclusion pattern `/admin?addExclusion=`*pattern*
   This operation is equivalent to adding an exclusion pattern via program arguments when starting the server.

5.2 Logging and error reporting

The following events are reported to stdout:

- user adds a location
- a newly added location is an ancestor or descendant of an existing location
- user adds an inclusion/exclusion pattern
- a file or folder in a watched location is modified, crawled for the first time or deleted
- a syntax error is reached when crawling a file. Currently, any syntax error in a file causes the crawling of that file to be aborted and all the information already extracted to be dropped. In order to be useful for IDE operations like auto-completion or metadata in hover text, the crawler must instead be resilient to errors and at least keep the information extracted until the error. This is planned for the immediate future, as the modular structure of the parser makes it well-suited for this change.

5.3 Scala/Java API

5.3.1 Storage

Object `Storage` is the entry point of the API. It stores the information extracted from the files and has control methods for adding locations and patterns, crawling the stored locations, and getting the information.

1. `addStringLocation(locationName: String)`

The scaladoc generated at `doc/index.html` in the project folder has a more detailed documentation of the API.

6 Code

The basic role of the parser is to take as input a Twelf file or a directory with Twelf files, scan them recursively and generate an XML "skeleton" document with metadata and dependency information about each file and each module inside it. While reading each file, the parser checks for syntactical (but not type-theoretical) validity. Two important metadata generated is the URI and exact position within the file (URL) of each module. The rest of the metadata is taken from comments associated with the file or with individual modules and is generated by key-value pairs like `@license LATIN`, where the keys are user-definable. The file and module dependencies are given as absolute URIs. The dependencies are taken from structures and imports and, in the case of the views, also from the domain and codomain.

The crawler is divided into a `FileCrawler` class and a `crawler` object, which can be run to crawl a directory or a file, by creating a new `FileCrawler` for each file.

6.1 FileCrawler class

6.1.1 Data structures

We will describe the datastructures for organizing Twelf blocks defined in `FileCrawler` below:

1. `BoundingBox`: stores initial and final coordinates of a block of text
2. `Block`: a generic block that will be extended with different case classes for modules, declarations, read statements, etc.
3. `StringBlock`: stores coordinates of a string surrounded by double quotes, used for URIs given in the files
4. `NamespaceBlock`: stores data about namespace declaratons, such as the namesace, URI and its alias
5. `CommentProperty`: stores a key value property starting with an "@" symbol in a comment
6. `CommentThrowBlock`: stores position of a comment with the syntax "%{ ... }%"
7. `CommentKeepBlock`: stores a list of dynamically generated properties of a semantic comment, i.e a comment with special syntax, which means it provides metadata for the block, symbol or document, following it.

8. `SigBlock`: stores information about a signature: its metadata, a list of its dependencies, its URI and URL

9. `ViewBlock`: stores information about a view: its metadata, a list of its dependencies, its URI and URL

For each such data structure, we overwrite the method `ToString`, which converts all the information inside to an appropriate XML node.

The `FileCrawler` class stores, for each object of this class, the following public data structures for organizing a file:

- `file: File` the Twelf file handle for reading
- `out: BufferedWriter` the file handle for buffered writing into `dependencies.xml`
- `baseURI: URI` the URI against which the first namespace of the file is relative.
- `baseURL: URI` the URL (folder location on local disk) which is the root of the file tree at `baseURI`.
- `associatedComment: Block` the `%* ... *%` comment at the beginning of the file. If it doesn't exist, `associatedComment` remains `null`.
- `fileURL: URI` file address on disk relative to `baseURL`.
- `currentNS: URI` current namespace as absolute URI, against which each module URI is relative to. It is updated every time a `%namespace "uri"` declaration is encountered.
- `modules: ArraySeq[Block]` an array of `SigBlocks` and `ViewBlocks` extracted from the file
- `deps: ArraySeq[URI]` an array of URIs for each dependency of the file, i.e. the set of namespace URIs declared in alias declarations. `deps` is equal to the set of values of the hash map `prefixes` (see below).

In addition to these, the class maintains several private data structures, useful for avoiding excessive message passing between various methods:

- `prefixes: HashMap[String, URI]` maps aliases to namespace URIs, as read from the `%namespace alias = "uri"` declarations.
- `lines: Array[String]` an array with the lines of the file in raw form
- `flat: String` the flattening of `lines` into one String. Newlines are converted into spaces.
- `lineStarts: ArraySeq[(Int, Int)]` keeps track of where each line starts. (i, j) ∈`lineStarts` means that `flat`(i) is the first character in the jth line.
- `keepComment: CommentKeepBlock` stores the most recent `%* ... *%` comment parsed. When the comment has been associated with a block, `keepComment` is set to `null`, in order to avoid associating comments to non-adjacent blocks.

Every file must start with a namespace declaration, either absolute or relative to `baseURI`. The software checks which case it is and, depending on whether a `baseURI` was given as program parameter, it computes the absolute URI or issues an error.

6.1.2 Methods

In addition to the datastructures, `FileCrawler` contains "crawling" methods which roughly correspond to each block type. Each such method reads a fragment of text starting from a position and returns a `Block` object with information extracted from the text. The methods check for syntactical well-formedness of the block and, in case of failure, throw a `ParseError` with a string explaining the reason of the error and the location inside the file where the error occurred. However, the methods assume that the correct block actually starts at the given position in the file; hence this must be ensured by the caller.

1. `crawlString(Int): StringBlock` reads a double quoted string with the initial quotes at the given `Int` position. The `StringBlock` which is returned stores the string itself in the `name` field.

2. `crawlIdentifier(Int): Int` expects the first character of an identifier at the given position. An valid Twelf+MMT identifier:

   - is at least one Unicode character long and can contain any Unicode characters, subject to the restrictions below
   - may contain dots, but not in its final position
   - doesn't contain any character from the following: `()[]{}":%`
   - ends before a whitespace character or one of the forbidden character from the list above. It can end before a dot, only if that dot is followed by either a whitespace or one of the forbidden characters.
   - is not identical to a reserved Twelf identifier: `-> <- _ = type`

   If the character at the starting position is already a forbidden character or a dot, a `ParseError` is thrown. If the identifier is a reseved Twelf identifier, a warning is printed, but the parsing continues.

   Unlike the other crawler methods, `crawlIdentifier` doesn't returns a special `Block` with the identifier, but simply returns the position immediately after the end of the identifier. The caller can then obtain the identifier by taking the substring between its initial and final position.

3. `crawlNamespaceBlock(Int): NamespaceBlock` reads both types of namespace declarations. For a declaration of type

   ```
   %namespace "uri"
   ```

   where `uri` is either an absolute uri (i.e. it is the first such declaration in the file), the method stores the name into `currentNS`. If `uri` is a relative uri based on the current namespace uri, the method updates `currentNS` by transforming the relative uri into an absolute one.

   In the case of an alias declaration

   ```
   %namespace alias = "uri"
   ```

   where `alias` is an identifier and `uri` is a relative uri based on the current namespace uri, the method converts the uri into an absolute one and updates the `prefixes` dictionary with the `(alias, absolute_uri)` pair.

4. `crawlCommentThrowBlock(Int): CommentThrowBlock` reads a `%{ ... }%` comment and stores its start and end position.

5. `crawlCommentKeepBlock(Int): CommentKeepBlock` reads a `%* ... *%` comment and, for each line at the beginning starting with `@`, it parses the key and the value into a `CommentProperty` object, which is added to `CommentKeepBlock`'s `children` array.

6. `crawlSigBody(Int, SigBlock): Int` takes the position of the initial opening bracket of a signature body and the partially formed `SigBlock` object and parses each symbol declaration, `structure` and `import`. For the latter two, it computes the absolute uris of the domain signature and adds them to the `deps` (dependencies array) field of the parent `SigBlock` object. `crawlSigBody` returns the position immediately after the final closing bracket.
7. `crawlSigBlock(Int): SigBlock` starts reading at the `%` character from `%sig`, checks for syntactical correctness and builds a `SigBlock` object by calling the `crawlSigBody` method.
8. `crawlViewBody(Int, ViewBlock): Int` takes the position of the initial opening bracket of a view body and the partially formed `ViewBlock` object and parses each assignment to symbol, assignment to structure and each import into the view. For the latter two, it computes the absolute uris of the views that are used and adds them to the `deps` (dependencies array) field of the parent `SigView` object. `crawlSigView` returns the position immediately after the final closing bracket.
9. `crawlViewBlock(Int): ViewBlock` starts reading at the `%` character from `%view`, checks for syntactical correctness and builds a `SigView` object by calling the `crawlSigView` method. In addition to the dependencies added via this call, `crawlViewBlock` also adds the domain and codomain's uris in the dependencies array.

The method that parses an entire file is `crawlFile`, which is called automatically in the initialization phase of a `FileCrawler` object. `crawlFile` first uses the method `getLines` to get the lines to store the file in both private class members `lines` and `flat`. Then, by calling `crawlNamespaceBlock`, `crawlSigBlock` and `crawlViewBlock` alternatively, it parses the file and fills up the data public structures. Furthermore, it prints out the XML nodes for the file itself and afterwards for its modules.

6.2 crawler object

`object crawler` acts more as a wrapper to `FileCrawler`. The object has two methods, `main` and `crawlDirectory`. There are no data members, so all the information is passed as argument when `main` calls the other method.

`crawlDirectory` scans a directory recursively for `.elf` files, ignoring `.svn` subdirectories. The method takes the directory handle (which it assumes to be valid), a output file handle, `baseURI` and `baseURL`. The latter three arguments are simply passed on to `FileCrawler` and are not used anywhere else.

The `main` method takes as arguments

- the file/directory name that the user wants to parse
- (optional) `BaseURI` and `BaseURL`. If these are not provided, then
  - the first namespace declaration in a file has to be an absolute URI.
  - `BaseURL` is considered to be the directory itself (if a directory is given as the first argument), or the directory in which the file is (is a file is given).

Then, `main` proceeds to check the validity of the file/directory name and calls `FileCrawler` or `crawlDirectory` accordingly. It also writes the beginning and end tags of the XML document.

7 Redesigning and documenting PL

In order to illustrate the measures introduced in sections 3 and 4, we redesigned and commented the Propositional logic subgraph of the Atlas situated at URI `http://cds.omdoc.org/logics/propositional/`. This served as a testing ground for the `crawler` as well, which was developed in parallel.

The original hyerarchical structure of Propositional logic had both the shortcomings of over- and under-modularity, as discussed in section 3.

```
syntax/                              model_theory/
         base.elf                              base.elf
         derived.elf                           base-zf.elf
         minimal.elf                           bool.elf
         minimal-prop.elf                      bool-zf.elf
         modules.elf                           modules.elf
         prop.elf                              modules-zf.elf
proof_theory/                                  prop.elf
         base.elf                              propmod.elf
         derived.elf                           prop-zf.elf
         iprop.elf                     soundness/
         minimal.elf                           base.elf
         minimal-prop.elf                      iprop.elf
         modules.elf                           modules.elf
         prop.elf                              prop.elf
         tableaux.elf
```

After the redesign, the structure is as follows:

```
classic/
       syntax.elf
       proof_theory.elf
       proof_theory_derived.elf
       model_theory.elf
       model_theory_old.elf
       soundness_old.elf
minimal/
       syntax.elf
       proof_theory.elf
minimal-classic/
       syntax.elf
       proof_theory.elf
```

In addition to these changes, two files have been moved to `logics/meta`: `bool.elf` and `bool-zf.elf`, They are more general than simply a model theory for propositional logic, hence they are useful in other logics as well and deserve being categorized as “meta”-logic.

8 Conclusion and future work

8.1 Crawler in other tools

One immediate use for the parser is within interactive viewers like JOBAD[5], which can thus get code snippets and file or module dependency analysis. For example, when somebody makes a change in a module, they can use the crawler to see what modules depend on it, so that they can track down the influence of their change. Another useful idea is to call the parser within a post-commit hook in SVN, in order to check which parts of the logic graph depend on the recently modified components, and are therefore possibly broken. They can then be compiled automatically and a list of modules in need of revision can be sent back to the user.

8.2 Symbol-level parsing

The reasonable next step for the parser is to provide the same information (metadata and dependencies) for each symbol declaration (or assignment) in each module. This would enable tools like JOBAD to provide the user with much more fine-grained (symbol-level) dependency analysis and source code definitions. With symbol-level parsing, we can do this for each symbol. For example, after modifying the type of a connective, we can see which expressions contain that connective and whether they compile.

8.3 Management of change

A light-weight, syntactic management of change can be implemented, which keeps track of the file system URIs and can compute the files affected whenever a reordering is performed. If the reordering is realized via an `svn move` operation, such a tool can even compute a rewriting rule with which it can update all related files.

8.4 Redesigning the logic graph

As the Propositional logic experiment has shown, refactoring according to the plan discussed in this paper is beneficial for code readability and extensibility. The next step is to apply the changes to the entire Atlas, with cooperation from the authors of each piece of graph.

References

[LAT] Latin: Logic atlas and integrator. http://trac.omdoc.org/latin/.

[RK10] Florian Rabe and Michael Kohlhase. A web-scalable module system for mathematical theories. Manuscript, to be submitted to the Journal of Symbolic Computation, 2010.

[RS] F. Rabe and C. Schürmann. A practical module system for LF.

[TIS] TIny SCAla Framework. http://gaydenko.com/scala/tiscaf/httpd/.

[5] http://jomdoc.omdoc.org/wiki/JOBAD