FDL: A Prototype Formal Digital Library

– Description and Draft Reference Manual –

Stuart Allen Mark Bickford Robert Constable Richard Eaton
Christoph Kreitz Lori Lorigo

Department of Computer Science, Cornell-University, Ithaca, NY, 14853-7501

{sfa,markb,rc,eaton,kreitz,lolorigo}@cs.cornell.edu

Preface

This manual describes the *first prototype* of a new kind of system which we call a Formal Digital Library (FDL). We designed the system and assembled the prototype as part of a research project sponsored by the Office of Naval Research entitled

Building Interactive Digital Libraries of Formal Algorithmic Knowledge.

A key purpose of the prototype library is to demonstrate that it is possible to build a system with many of the properties called for in the project proposal and to illustrate important scenarios for its use. Experience with the prototype library will influence the design and construction of an improved system. The current prototype includes some expediences that made it possible to create a working system in less than a year.

The prototype FDL is one part of the overall project. There are other theoretical and experimental efforts that are described in other publications.

The library described here contains definitions, theorems, theories, proof methods, and articles about topics in computational mathematics and books assembled from them. Currently it supports these objects created with the theorem proving systems MetaPRL, Nuprl and PVS. We intend to include material from other implemented logics such as Minlog, Coq, HOL, Isabelle, and Larch in due course.

In addition to the purely formal material, the Library supports mathematically literate hypertext articles that cite and use the formal concepts. These include explanations of reference algorithms and explanations of formal mathematical models used in applications.

Many operations on the Library are automated and extensible. The basic operations are to find and read material, organize it, and submit new material. New operations can be defined algorithmically.

This manual is intended to help users understand the operation of the Library and to demonstrate to those interested in the project what else we intended to build and how it will be used.

Contents

1 Introduction

Achievements of mathematicians, logicians and computer scientists over the past fifty years have created the practical means to formalize vast amounts of mathematical knowledge. Moreover, the value of algorithmic mathematics and the need to validate computer software and hardware provided financial support to actually carry out this formalization on a large scale worldwide. The result is a large collection of formal material that arises from applications; included therein is a large number of general mathematical results needed to support those applications. The volume of material increases daily.

This formal material presents extraordinary opportunities and challenges. The opportunity is to organize the material so that it is more widely usable and shared. It is valuable in creating more reliable hardware and software and thus valuable to all the activities that depend on reliable computing. It is valuable in expanding the capacity of many formal tools needed to protect the software infrastructure of the nation and of the global communication system.

As an artifact in itself, the collection of formal material has exceptional properties. It is *digital*. It is *logically organized* and highly structured. It codes vast amounts of mathematical knowledge, especially algorithmic knowledge. It represents the highest standards of correctness and accuracy that we know how to achieve as a technical society. It captures the precise thinking of a large number of excellent scientists who have spent hundreds of person years in creating this as yet unorganized collection with limited accessibility.

One long term goal of our project is to organize this formal knowledge and provide software tools for using it in a variety of ways. The first tools we produce will be simple, allowing people to read, organize, search, annotate and incorporate the material in other digital documents. More advanced tools will be provided on this basis.

It is clear that there will be large organized collections of mathematical and scientific knowledge in digital form that will be intelligently accessed with computer assistance. The FDL will contain such material and it will be integrated with the formal content.

Another long term goal is to enable a worldwide user community to contribute new formal material and to contribute original articles that incorporate this material in aid of ordinary scientific and educational discourse.

1.1 Goals

In order to create the massive amount of content needed in a general global resource and to transfer the methods to other disciplines, it must be possible for a significant number of people worldwide to contribute. Likewise to prove formal properties of a large software system, it must be possible for many people to contribute. Thus it must be possible to share results among formal theories developed with different theorem provers; and it must be possible to account for logical correctness in an environment that tolerates many different theories, some incompatible with others. In this context it is critical to know what depends on what.

As we have thought about how our Library might become a *distributed open global interactive information resource*, we have identified key technical challenges and specific intermediate objectives. Specifically we propose approaches to the problem of accounting for correctness and truth in a library that allows multiple logics and multiple theorem provers, for knowing exactly what a result depends on, for combining sublibraries and for performing a variety of routine operations on libraries such as searching and browsing. We also hope to provide very advanced operations on theories such as soundly translating among them, generalizing, specializing and reflecting them. These will be operations on theories as objects stored in the Library and operations on code in these theories. We plan to use the computational contents of proofs as components of programs in other programming languages in a consistent way and to provide interaction with the Library using the Web. These goals generate many interesting technical problems, several of which we discuss below.

1.2 Use Scenarios

From a user's perspective, a digital library serves three different purposes

- As a *library* it provides a repository for *information* that is neutral about its content and mainly supports the efficient publication and retrieval of information.
- As an *archive* it provides records of *facts* and accounts for the integrity of these records. Furthermore it ensures the longevity of these records, which makes it possible to trace the justifications for facts back to their very origins.
- As a *workspace* it enables clients to make use of the stored information and facts and to reorganize them in new ways. It also supports the creation of new contributions for archiving, which includes the creation of justifications that the archive may check before accepting the contribution.

The library that we are developing is *formal* in the sense that significant parts of the stored data have a precise meaning, which may be checked by a computer. We often speak of a *logical library*, to indicate that we use *formal logics* (as opposed to rigorous mathematical approaches in natural language) to check the validity of arguments and justifications.

In the following we describe a few typical scenarios for using a digital library of formal algorithmic knowledge. Additional scenarios can be found in the FDL design documents at `http://www.cs.cornell.edu/Info/People/sfa/XDL_scenarios1.html`

1. A programmer wants a precise explanation of a standard algorithm to know why it works in order to implement a variant of it. He or she finds several reference algorithms using different representations of the data. There is one part of one of the more familiar algorithms that has always seemed unnecessarily complex; by digging into the formal proof of its correctness the programmer finally sees what he was missing.
2. Two large libraries are maintained by parties that learn they can trust each other's library maintenance, and decide to accept each other's certificates without always reverifying them locally. This is not a deadly embrace because that a certificate is borrowed from another library is part of the certificate.

It is discovered that one institution maintaining a library has not been following the protocols that all the library maintainers agreed to. Because certifications passed from one library to another are recorded as such, they can be located, and all things that depended on them can be identified and scrutinized (and hopefully re-certified).

3. A software company has developed a package of programs whose sources it wants to keep secret. How can it assure customers of facts about the programs? There must be a trusted impartial third party to certify the claims. This party would be a library process trusted to implement its published policies for certifying proofs. The company would maintain its own private library of source and object code and proofs of correctness. The impartial library process would certify it by employing a public logic, uploading the source code and object code and proofs, then itself checking the inferences by the public inference engine. If it succeeds then it deletes the source code and proof, and creates a certificate that refers to the object code, to the statement of correctness, and to the public inference engine, and claims that there once was a proof of the statement about the object code which the impartial library checked (then deleted)

4. Researchers working on mobile code security determine that properties of assembly level code must be verified. As a first step they want a prototype highly-automated procedure similar to an extended type checker for specific properties, delivered in a six month time frame.

   The Library contains a formal model of the virtual machine (VM) with properties established in PVS. Complete reference material is available in the library along with rewrite rules and formal theorems from a public PVS section of the FDL.

   The CIP/SW researcher codes an extended type-checking algorithm by modifying a documented type checker in the Library. A small inference engine is created as a tactic in MetaPRL which is extremely fast. It is made available in the library as XCheck.

   A related project is proving properties of a type checker using reflection. Components of XCheck have been verified, and the group quickly establishes an unexpected feature of XCheck, that it fails to guarantee memory safety under certain conditions. The designer modifies XCheck to produce version 2, leaving a trace of the development.

   Tactic optimization procedures can be applied to XCheck under certain standard conditions. An optimized XCheck is proved equivalent to the original. All this is done in four months, with documentation in a series of articles archived in the library. These articles allow the CIP/SW mobile code security team to use the new XCheck code.

1.3 Relationship to National Needs

It has been well established that the United States needs better programming technology to assure the safety and reliability of the nation's software infrastructure. The National Research Council study on information system trustworthiness concluded that the current science and technology base is not adequate for building systems to control critical software infrastructure [Sch99]. The President's commission on critical infrastructure protection and the PITAC report reached the same conclusions [Pit99]. These reports placed special emphasis on finding new ways to build more reliable and secure software and stressed the need to conduct fundamental research on the problem with a long range view.

> *But it has become clear that the processes of developing, testing, and maintaining software must change. We need scientifically sound approaches to software development that will enable meaningful and practical testing for consistency of specifications and implementations. This requires long-term research in languages, theories, simulation, analysis, and testing that could lead to standardized multilevel mechanisms similar to those which have created the success in computer-aided design for digital hardware.*

The PITAC report [Pit99] finds that the nation faces these key problems in software.

- demand for software exceeds our ability to produce it
- the nation depends on fragile software
- technologies to build reliable software are inadequate

The interactive logical library that we are developing contributes to mechanisms for guaranteeing the *reliability of large software systems*. It can be used to develop software systems that are *correct-by-construction* and *documented by the context* and makes it possible to connect textual documentation to formal documentation.

Scientific and social benefits

Providing a logical library will result in many significant benefits to scientific practice as well as to the social impact of science. First, we will be able to *increase the reliability of reference material* at a low marginal cost and provide a starting point for the evolution of these mechanism to dramatically lower cost. We can know that collections of definitions and theorems are correct according to specific designated criteria and are consistent. The correctness can be established at the highest levels of assurance known, namely proofs checked by both humans and machines. The process of progressively providing computer certifications for more and more claims asserted in a collection is a process that we call *hardening* the collection, and it applies to the software systems stored in the library as well. The library provides *an arena for gradual formalization.*

We contribute to formal *mechanisms for guaranteeing the reliability of large software systems*. An interactive logical library can be used to develop algorithms and even systems that are correct-by-construction and documented by the context. Moreover the logical library provides mechanisms for integrating textual and formal documentation.

A logical library will *complement the mechanisms of electronic publishing* and open the way to verify journals that specialize in formalized mathematics [Miz, QED]. In such journals every result will be checked by certified theorem provers, including those for which there is a small proof checker that can be publically scrutinized (this is a system that obeys the so-called *deBruijn principle*).

There is *significant educational value* in formal reference material. We have used such material in teaching and have studied its impact [Con96]. In particular one can learn about a particular system in a context where the design, the specifications, the algorithms and

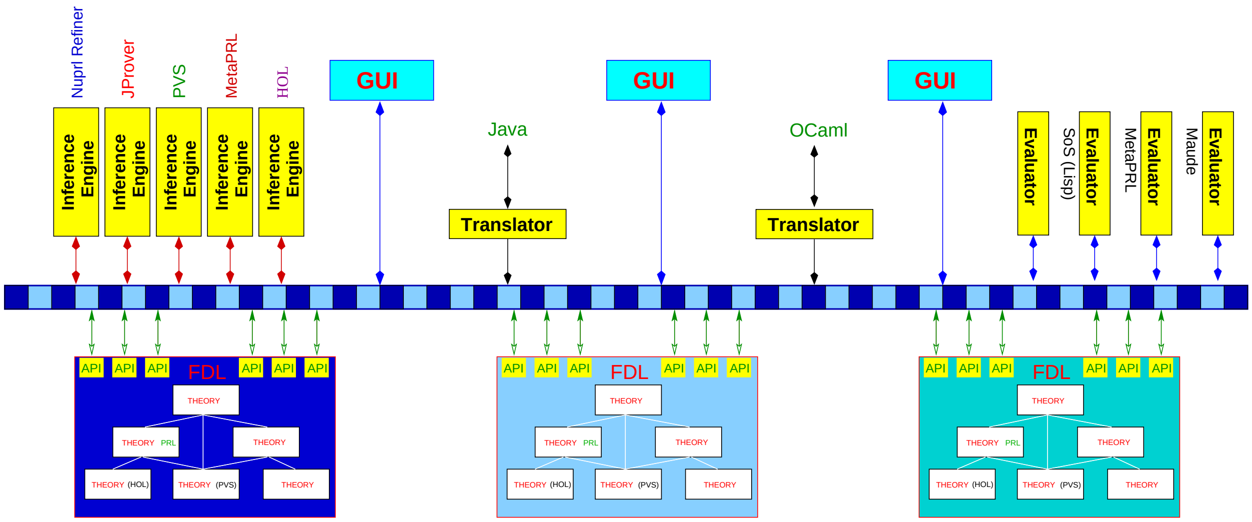


Figure 1: Using the FDL in a heterogeneous network of services

the proofs are all linked to the relevant literature. Significant benefits accrue from having static formal material as is now posted on the Nuprl web site [NuPb], but even greater advantages come from allowing users to *interact with proofs and algorithms*. Readers can explore the consequences of deleting an assumption or strengthening a conclusion. They can watch an algorithm execute on concrete data and symbolically. They can ask whether one result depends on another; they can see exactly how or whether a proof breaks by changing definitions, lemmas, inference steps and justifications. They can also decompose a high level inference step, say built from tactics or derived rules, into its constituent parts, layer by layer as subjective understanding dictates.

The growing database of formal computational mathematics is a new resource for *studies in artificial intelligence*. As one example, members of the AI group at Cornell are generating natural language proofs from parts of the Nuprl corpus [HMBL99]. Interesting ideas have been proposed for automating more of the process of formalizing articles and textbooks.

Public access to this global interactive digital library of algorithmic mathematics will benefit the non-experts who must use technical results, and it will empower students and lay persons to explore mathematics interactively and to contribute to these libraries. It will create what we call a *formal forum* connecting those interested in formal methods. A much wider group of people will be able to participate in adding to scientific knowledge, and we might create communities of volunteer contributors in the same way (but on a smaller scale) that advances in databases have allowed 20 million naturalists and bird lovers to contribute to the study of nature through interactions with Cornell's laboratory of ornithology.

2 FDL Design

The use scenarios for the formal digital library suggest that its design be open in several dimensions, as illustrated in Figure 1. The library will connect to *multiple clients* with different needs and correctness criteria wrt. the facts they deal with. The information that a client needs may be distributed over *multiple libraries*. Finally, the design must allow for *multiple implementations* of the formal digital library, which may provide different additional features and may employ different implementation techniques.

Our research on the development of FDL serves two major purposes. First, we develop a general model that describes the core functionalities and features of digital libraries of formal algorithmic knowledge as well as a suitable architecture for building formal digital libraries. Secondly, we provide a specific implementation of a formal digital library and explain the design decisions and extra features incorporated in the FDL prototype. In the following we will use examples from the latter to illustrate some of the principles of the general model.

2.1 Design Objectives

The design of a formal digital library is based on the following objectives

Connectivity: The FDL must be able to connect to multiple clients (proof tools, users, etc.) independently, asynchronously, and in parallel.

Usability: Clients of the FDL must be able to browse library contents, search for information by a variety of search criteria, and contribute new knowledge to the library.

Interoperability: The FDL shall support the cooperation of proof systems in the development of formal algorithmic knowledge. Different proof systems will be based on different formal theories and on different internal representations of knowledge. The representation of knowledge in the FDL has to be generic, so that it can be translated into a large variety of formats when providing knowledge to clients or receiving formal knowledge from them.

Accountability: The FDL needs to be able to account for the integrity of the formalized knowledge it contains. As it supports interoperability between very different proof tools, there cannot be an "absolute" notion of correctness. Instead, the FDL has to provide justifications for the validity of proofs, which will depend upon what rules and axioms are admitted and on the reliability of the inference engines employed. Furthermore, these justifications must be exposed to determine the extent to which one may rely upon the provided knowledge. We call these justifications *certificates*.

Information Preservation: The FDL has to guarantee that existing knowledge and justifications cannot be destroyed or corrupted by clients or system crashes.

Archiving: The FDL has to support the management of knowledge on a large scale such as merging separate developments of large theories and performing context-specific tasks. This requires the use of abstract references to knowledge objects, as traditional naming schemes do not scale.

One of the main objectives of our project is to identify a minimal set of design policies and necessary components that every implementation of an FDL must support and to develop a reference implementation of the FDL that satisfies these requirements.

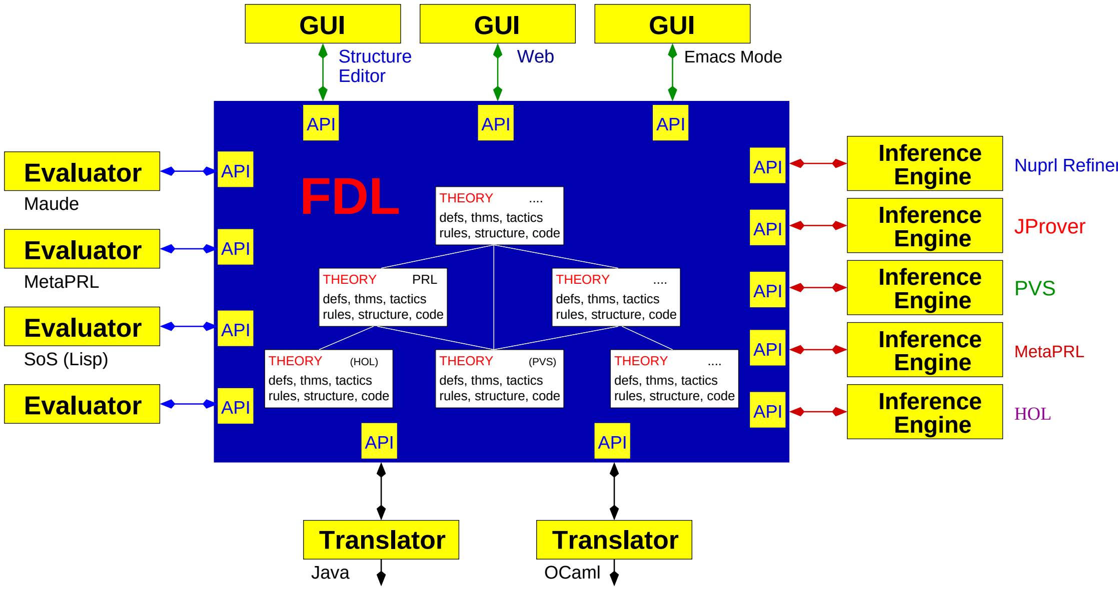


Figure 2: Interaction between the FDL and its clients

2.2 Reference FDL Structure

The FDL data base, also called a *library table*, is an association list of objects together with object identifiers. Objects are abstract terms that can accommodate almost any kind of formal content (see Section 3.1). Object identifiers are also abstract and cannot be accessed without going through the library.

The library provides a small set of primitive *library operations*, such as *binding* an object identifier to an object (i.e. adding an object), *unbinding* an object identifier (deleting the object), creating new identifiers, and looking up objects. There are several primitives for modifying the content of an object. However, these functions do not overwrite an object content but create new content, which then is bound to the corresponding identifier. From the primitive operations we define many library operations (see Section 3.2) in a similar way complex proof techniques are built from from basic inference rules and tacticals.

We build *closed maps* (Section 4) representing the work space for client sessions on top of the basic operations and logical *accounting and security* Section 5) on top of that.

2.3 Current FDL Prototype

Figure 2 illustrates the architecture for an interaction between the FDL and its clients. All clients are independent processes that communicate with the library as central repository. The library contains all the definitions, algorithms, theorems, axioms, inference rules, theories, objects relating theories based on different formalisms, meta-level code for proof tactics and decision procedures, and other forms of justification, to which a client may refer when processing formal algorithmic knowledge or developing new contributions for the library. Even descriptions of how to present formal knowledge in various formats used by different interfaces will be stored as structure objects within the FDL.

The library can communicate simultaneously with arbitrarily many clients, such as various user interfaces for browsing and editing formal knowledge, inference engines for proving facts, rewrite engines and evaluators for transforming and evaluating algorithmic knowledge, translators for generating code in a specific programming language, etc.

This makes it possible to build formal knowledge using a variety of proof systems such as Nuprl [CAB+86, ACE+00], MetaPRL [Met], PVS [ORR+96], SPECWARE [SJ95], HOL [GM93], Coq [Dea91], Isabelle [Pau90], or Ωmega [BBS99], first-order provers like JProver [SLKN01], Otter [WWM+90], EQP [McC97], or Setheo [LSBB92], proof-based program generators like MinLog [BBS+98], rewrite engines like Maude [CDE+99], computer algebra systems [Wol88, Map], decision procedures [NO79, Sho84, SVC], and model checkers [McM93, Dil96, Hol97]. These systems may even cooperate through the library, which enhances their reasoning capabilities in the production of formal algorithmic knowledge.

Supporting a variety of interfaces commonly used in proof systems, such as structure editors, emacs modes, web browsers, enables several users to work in parallel on the same formal theory while using their favorite interface.

2.4 Programming Practice

Our conceptual path to the library design follows the need to maintain a flexible development method, permitting divergent partially independent developments, and yet to be able to justify claims of validity, exposing the assumptions of such justifications.

We chose an incremental approach to the development of our FDL prototype. We begin with a simple implementation that provides the basic functionality and a few algorithmic theories as standard library content. This allows deploying it to "daring" users who are interested in experimenting with the FDL, browsing its content, developing new formal content, and connecting their own clients to the FDL. New functionality will be added incrementally, which makes sure that there is always a working prototype that can be tested and evaluated. New library contents will be added incrementally as well, either by explicit interaction with the FDL or by migrating the contents of existing formal digital libraries into the format of the FDL.

3 Library Data and Operations

3.1 Basic Data

Theorems, definitions, algorithms, tactics, comments, articles, and other library contents are represented by a common basic data structure called *objects*. Objects are abstract terms that are associated with a *kind*, a variety of *properties*, and possibly with *extra data*.

Abstract terms provide a *uniform data structure* for representing almost any kind of formal content. Abstract terms consist of an *operator identifier*, a list of *parameters*, and a list of *subterms*.

The abstract term syntax makes sure that no predefined structure is imposed on the contents of the library and makes parsing unnecessary. All visible structure and notation is generated within the work space by consulting display forms (i.e. library objects with kind DISP) that describe how to "read" an abstract term. Display forms are processed by the API's for user interfaces and other clients when displaying or modifying an object.

This separation between internal representation and external presentation makes it possible to present library contents in the native language of almost any proof tool without having to convert between different data structures. Furthermore it makes formal notation extremely flexible and expressive, as it supports an almost arbitrary syntax and allows information to be presented differently depending on context and the preferences of the clients or users of the FDL.

The *kind* of an object is a description of the intended role of the abstract term. It allows making a distinction between theorems, definitions, tactics, comments, etc., and identifying structure information when assembling theories in a client's work space. Currently the following kinds are defined in the FDL.

- `ABS` for *abstractions*,
- `DISP` for *display forms*,
- `STM` for *statement* objects,
- `INF` for *inference* objects,
- `PRF` for *proof* objects,
- `RULE` for inference *rules*,
- `COM` for *comments*,
- `CODE` for tactic and other *code*,
- `PRC` for *precedence* objects,
- `DIR` for *directories*,
- `TERM` for *objects* of unspecified kind.

The *properties* contain status information that is helpful for maintaining the object, tracking dependencies, building justifications etc. The most common properties are

- A *liveness bit*, indicating whether the object may be referenced to by others
- A *sticky bit*, indicating whether the object may be removed from the library table during garbage collection
- A *description of clients* to which the object shall be made visible
- A memnonic *name* which is commonly used for presenting the object identifier.
- The *language* in which a code object is programmed.
- A *reference environment* describing the context of the object.

Extra data are used to collect information that accounts for the validity of an object's content. Statements include a list of (links to) proof objects as extra data, proofs include a tree of inferences, and inferences include primitive inference steps.

In the *library table*, objects are also associated with *abstract identifiers* that are bound to the contents of the object. All references to objects have to use these abstract identifiers, which in turn are linked to names for objects in a client's closed map.

Object contents are viewed as non-destructive. To change the content of an object, one has to create a new object content and rebind the abstract identifier of the object to the new content. To remove the object from the library, one simply removes the binding between the abstract identifier and the content. Object contents are usually not removed from the library table except by garbage collection.

All library operations are built from a small collection of primitive operations on object contents and library tables. These operations are

- *Binding* an object identifier to an object and *unbinding* an object identifier.
- *Looking up* object contents bound to an abstract identifier.
- Generating *new object identifiers*.
- *(De)activating* an object (changing the liveness bit).
- *(Dis)allowing* garbage collection for the object (changing the sticky bit).

There are also several primitives for *creating new object contents* from existing object contents and new data. The most basic primitive creates a new abstract term for the object. Other primitives modify extra data related to building proof structures by changing the list of proofs linked to a statement, modifying the inference tree of a proof, or changing the inference step of an inference object.

3.2 Basic Library Operations

Basic library operations are services such as inserting, removing, and looking up and searching for data as well as supporting the development and modification of definitions, theorems, proofs, algorithms, and informal descriptions. These services are fundamental for most client applications and should be supported by all implementations of formal digital libraries.

In contrast to the primitive library operations described in Section 2.2, basic services describe the interaction with the client through the client's current closed map, although some of the also affect the contents of the library itself.

Below is a list of operations that we have implemented in our FDL prototype.

- *Basic operations on library objects*
  - Name and create objects of various kinds, such as rules, definitions (abstractions and display forms), theorems, comments, etc.
  - Arrange objects in folders and theories
  - Move and rename object
  - Create links to objects
  - Deactivate and re-activate objects
  - Remove objects and links
  - Browse the library and its theories

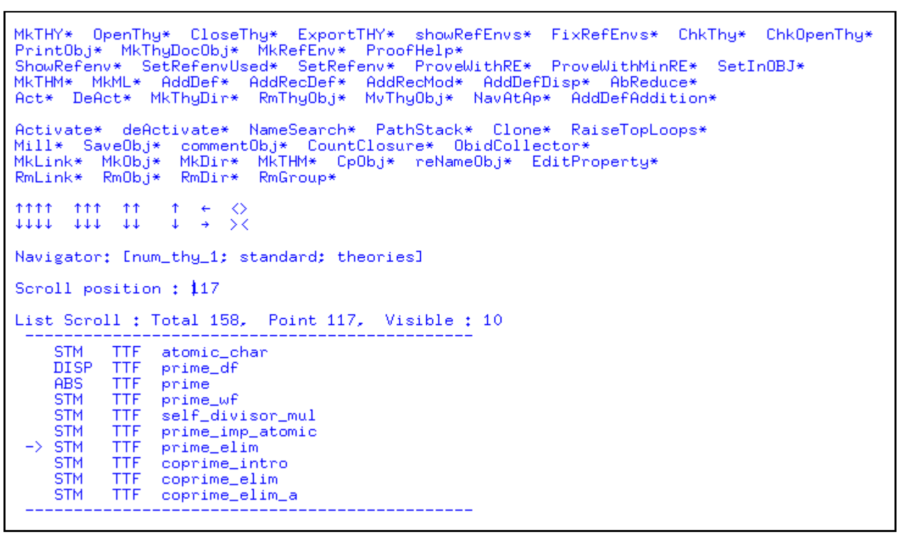
  - Search for objects by name
  - Link formal objects to text
  - Present and print objects in various formats (TeX, HTML)

- *Support for Content Development and Modification*
  - Prove a theorem, using multiple proof tools
  - Logically account for inference steps in a proof
  - Explicitly store justifications of inference steps
  - Edit objects (proofs, definitions, code objects, ...)
  - Create new proof tactics and decision procedures

- *Theory Operations*
  - Export and import a theory
  - Check a theory
  - Restrict a theory to objects relevant to a specified list of objects in it
  - Search for lemmata containing a specified list of object names
  - Search for objects modified within a given time specification
  - Migrating an externally developed theory into the FDL (currently only for Nuprl 4 format)
  - Milling: a framework for developing tools for importing and migrating data.

Let us illustrate how some of these operations work in the current FDL prototype.

- To *browse the library*, the FDL prototype provides a visual interface that arranges objects and theories in folders (also called directories) of the user's work space.



The interface window shows information on a segment of the library at the bottom and above that a few statistics and a zone with buttons for issuing basic library commands. A user may move a *navigation pointer* through the current folder by using arrow keys, the mouse, or clicking on one of the arrow buttons ↑↑↑↑, ↓↓↓↓, . . . , ↑, ↓. To move into a subfolder or to open an object for editing, one uses the right arrow key (or middle-clicks on it with the mouse), to move out of a directory, one moves the navigation pointer to the left.

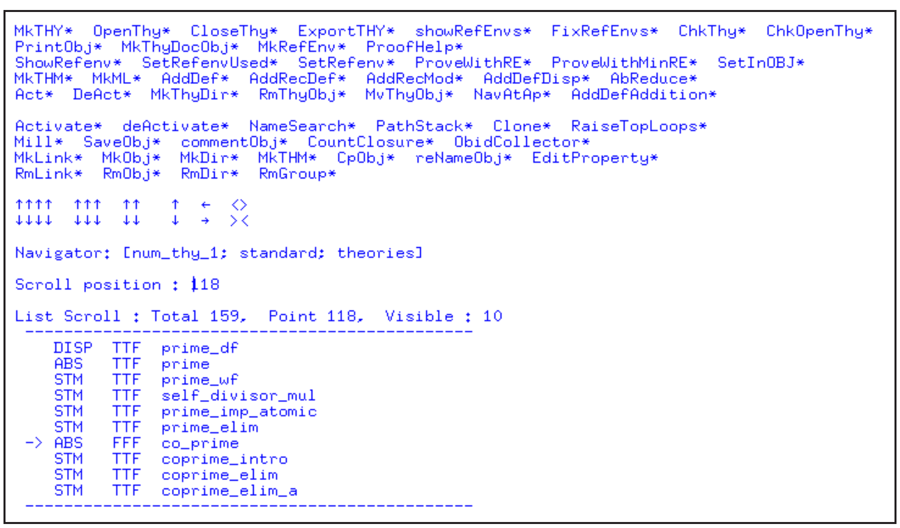
- *Naming and creating objects* is a combination of two, more fundamental operations. In the first step, a function `mk_obj` creates a default object of a given kind and adds it to the library table. The object will be bound to a new object identifier, which will be returned as the result of the function.

  In the second step, the object identifier will be linked to a name in the user's work space and assigned a position in one of the user's folders, usually immediately after an already existing object. To identify this object, a user has to rely on library mechanisms that detect the corresponding abstract object identifier from information provided by the user. In the current prototype this mechanism is provided by the visual interface: the object referred to is the one pointed at by the navigation pointer.

  Both steps are combined into a single user command, which requires the user to provide the name and the kind of the object to be created. Executing the function "`dyn_mkobj` ***kind name***" will create a new object with the given kind and name. Executing

  "`dyn_mkobj ‘abs‘ ‘co_prime‘`",

  for instance, will create an abstraction object named `co_prime` and position it immediately after the current object, as indicated below.



  Usually, this command is issued interactively by clicking the `MkObj*` command button, which will open two templates into which the user may type in the name and kind of the new object.

- *Renaming an object* means linking the object to a new name in the user's work space and changing the object's name property. To do so, one has to determine the object's abstract identifier and assign a new name to it. As the former is identified by the navigation pointer, a user only has to execute the function "`rename_obj` *`new-name`*" or issue the same command interactively by clicking the `RenameObj*` command button.

- *Exporting and importing theories* is important for moving theories between libraries in a controlled fashion. Theories are usually associated with specific folders in the user's work space. To export a theory, a user moves the navigation pointer out of the current folder, such that it identifies the folder's object identifier and then issues the command `dump_thy` (or clicks the `ExportTHY*` command button). This will collect all the objects in the marked folder and dump them to a file in a default location.

  To import a dumped theory from a file, one has to provide the path name of the file by issuing the command "`replace_objects` *`path-name`*". This will create a folder containing all the objects of the dumped theory and place it at the same location in the user's work space. If the folder already existed, objects of the dumped theory will be added to the folder. In case of name clashes, the name of the old object will be modified if its content is different. If the contents are identical, the new object will be ignored.

3.3 Native Library Language

Clients of the library must be able to stipulate programs executed by the library process. Request for execution of such programs and returning their results is a basic interaction between clients and the library. Most work of certifying inference steps is expected to be done outside the library by inference engines (see Section 5.1). The library simply invokes those engines and records the results.

A native language should provide generic computational methods as well as some basic library-specific operations for manipulating ones current closed map (Section 4), managing a small external name space, control of access to objects by other clients, and for communicating with external processes.

The execution of native language programs is implemented as part the library and forms the basis of certification (Section 5.2). The facts to which a certificate attests are simply that certain native language programs were executed to certain effect.

There may be multiple native languages, suitable for different styles of programming by customers. For example, a higher-order functional style (as used in our current prototype implementation of the FDL) and a conventional imperative style language would be basic candidates, and perhaps a virtual machine for use by those clients who prefer to develop their own languages for execution by the library.

3.4 Library State

The library state contains a description of current library policies, ongoing interactions with clients, (temporarily) unfinished work, and other information that is necessary to guarantee the consistency of the library. Specifically, the following will be included in the state.

- Certificate policies, i.e. the criteria for a library object being a certificate.
- The collection of all closed maps (see Section 4.1).
- The current set of *alterable* submaps, i.e submaps of the repository that a constitute the current closed maps of the various sessions. Information includes the "owner" of the map and limitations for sharing the map with other clients.
- The current set of client sessions, which for each client includes the identity of the client and the method for communicating terms with it.
- A collection of *session journals* that are used for determining the current working environment of a client (i.e. its current closed map) when it connects to the library. The standard policy would be to select the most recent stable working environment, but clients may also chose to resume earlier sessions.
- The current set of external library sessions, which includes information about how libraries communicate among each other.

In addition to the above parts of the state the library state is also expected to include temporary information that is needed for achieving a consistent state of the library after modifications to library objects. This temporary part of state is expected to include

- The set of stale certificates (Section 4.3)
- The changed objects referred to by each stale certificate and their prior content.
- The objects referred to by each stale certificate that were distinct and are now to be identified together.
- Objects marked for deletion upon successful reconsideration of all stale certificates.

4 Sessions and Current Closed Maps

The usual method of interaction with the FDL is to build and develop a *client work space*, i.e. a collection of named object contents that provide a specific view of the data and can be tailored to the specific needs and permissions of a client.

In a work space, abstract object identifiers are linked to concrete names chosen by a user. This allows the user to organize objects in folders, to use the same name in different folders, and to establish "private" links between objects. The work space may also restrict a client's access to certain library objects. Most importantly, however, it protects internal identifiers and object contents from being modified without going through the FDL, helps preventing name collisions, and makes proof mechanisms independent of particular naming schemes.

The *library manager* provides clients with utilities for building, storing, and sharing collections of *session objects*. It maintains the work spaces and thus enforces a discipline for building named collections, thus preserving the *coherency* of the collections.

4.1 Closed Maps

Work spaces are represented by *maps* from a finite set of names to library objects. These maps have to be *closed* in the sense that the objects they refer to do not contain any references to objects that have no name in the map. Thus the basic model of interacting with the library is to maintain a *current closed map* as a part of state that is updated repeatedly as one works.

In general, a *closed map* is a function of type $D \rightarrow \texttt{Term}(D)$, where D is a finite discrete type of indices and $\texttt{Term}(D)$ is the type of terms whose subterms only contain abstract identifiers in D. Usually we identify objects in a closed map with their index (or *name*).

In practice the class D will be varied continually. For example, extending a closed map requires selecting a larger index class. Deleting members of a closed map requires a smaller index class. In both cases, we have to make sure that the resulting map remains closed.

If the restriction of a closed map $m \in D \rightarrow \texttt{Term}(D)$ to a subclass $X \subseteq D$ is itself a closed map (i.e. is in $X \rightarrow \texttt{Term}(X)$), then we call it a *submap* of m. Similarly a *supermap* of m is a closed extension of m to a class $Y \supseteq D$. Two closed maps $m \in D \rightarrow \texttt{Term}(D)$ and $m' \in D' \rightarrow \texttt{Term}(D')$ are *equivalent*, if they are simply renamings of each other.

Closed maps are essential for defining the notion of *dependency*. Objects depend on others if they directly or indirectly refer to them. An expression $t \in \texttt{Term}(D)$ *refers* directly to an object (index) $x \in D$ if x occurs within a subterm of t.

The notion of dependency is the key to defining *correctness*. While it is possible to define useful notions of correctness with respect to state, the enduring ones can only be formulated in terms of closed maps: the correctness of an object should only depend on the correctness of the object it refers to but not on library objects that are not within the current closed map.

4.2 Operations on Closed Maps

The library is a repository not of closed maps per se, but is rather a repository of data and instructions for building closed maps modulo choice of abstract identifiers. In a session the current closed map is initialized from the library, transformed through a sequence of operations, and then stored back into the library for later retrieval. Some basic operations that can be defined on closed maps are

- *Uniform renaming* of abstract identifiers.
- *Contracting* around a set S of objects, i.e. restricting the closed map to objects in S together with objects referred to by the objects in S.
- *Focusing* on S, i.e. restricting the closed map to objects relevant to S (elements of S and object referred to by objects in S or referring to them).
- *Deleting* S along with all objects that refer to elements of S.
- *Merging* two closed maps in a way that objects can be identified.
- *Cloning* S, i.e. replicating the objects in S and replacing references to elements of S within the clones by references to the corresponding clones.
- *Splitting* wrt. S, i.e. cloning S together with all objects that refer to objects in S.

- *Reassigning* the indices of a closed map to new contents.
- *Folding* a closed map by identifying certain objects within it with each other.

All but the last two operations are *conservative* in the sense that they do not invalidate certificates (see Section 5.2). Reassigning and folding, however, does affect certificates as well and therefore has to be coupled with operations that modify and rehabilitate these *stale* certificates rather than simply deleting them.

4.3 Stale Certificates

The presence of stale certificates in a closed map corresponds to an inconsistent state in a database, and part of completing a closed map operation is to eliminate staleness. As different kinds of certificates can be implemented, closed maps rely on procedures for creating new certification objects of that kind, and procedures for reconsidering a certificate, i.e. modifying its contents. These *certification procedures* may also create, alter, or delete other objects.

However, some basic operations may have cascading consequences on the library that are beyond the control of any specific certification procedure, as the content non-certificates can be changed almost arbitrarily. When any object's content is altered other than by conservative operations each certificate object referring to it will be reconsidered according to the procedure specified for its kind. If reconsidering a certificate alters its content, then certificates referring to it must themselves be marked for reconsideration, etc. Similarly, when multiple objects are identified with each other, any certificate that contains references to more than one of them gets marked for reconsideration.

5 Accounting mechanisms

One of the central aspects of a formal digital library is to account for the integrity of its contents and to support arguments for claims of the following form:

> Because the library contains a proof of theorem T that refers to a given collection of proof rules and proof engines, theorem T is true if those rules are valid and those engines run correctly.

Accounting mechanisms determine how to execute *inferences* as specified by a proof tactic depending on the actual contents of the library and produce *certificates*, which attest that certain actions were taken at a certain time to account for the validity of an object.

Accounting mechanisms are also needed to determine whether a proof built from a collection of certified inferences is acceptable for a given purpose. This would be trivial if the FDL would be restricted to a single uniform logic and to a a single inference engine. But inferences that may employ a variety of logics and inference engines cannot be simply combined. Instead, certain stipulations limiting proofs to a given set of rules, axioms, and perhaps other objects on which these may depend must be expressed and checked.

We use what we call *proof sentinels* to express these stipulations and to indicate a reduction of validity to those things whose criteria of correctness lie outside the formal system.

This assures the proof to be correct as long as the rules in the sentinel are and makes it possible to distinguish claims that are accounted for from those that are not.

5.1 Inferences

One of the most fundamental mechanisms to account for the validity of library contents such as theorems and proofs is the application of logical inferences from a finite number of premises to a conclusion. Inferences are represented by trees of *inference steps*, which in turn are represented as library objects. A library process checks or generates an inference step by applying *inference engines*, which create proofs in some formal calculus according to user specified methods.

The fact that an inference step has been verified by a given inference engine is represented by an external certificate that refers to the inference step. There may be multiple certificates for the same inference, certifying that the inference has been checked by different inference engines. Depending on the contents of the available certificates, the inference may be considered valid or not in a specific context.

Inference engines support the development of new formal knowledge by providing mechanisms for interactive, tactical, and fully automated reasoning in a specific formal language. As the formal digital library supports almost any formal language, it can be connected to a variety of inference engines that will provide justifications for its formal content.

5.2 Certificates

Certificates are the basis for logical accounting. They attest that certain library actions were taken at a certain time to validate the contents of, or identity between, objects. A certificate will be realized as an object, which can then be referenced and accessed like other objects save for certain constraints. A certificate cannot be created or modified except by the library process following a procedure specific to the kind of certificate in question.

Although certificate contents are expected to often be rather compact, largely consisting of Object references, they will often also be rather expensive to establish. By realizing certificates as objects the library can build certificates that depend on others whose correctness is independently established. Thus one process of certification can contribute to many other certifications without having to be redone.

The paradigmatic certificates are those created to validate proofs. An *inference step* certificate attests to the fact that a specified inference engine accepted that a certain inference. It is built by applying the engine to the inference, and includes references to the inference step as well as to the instructions for building or deploying the inference engine.

A proof is a rooted dag of inference steps. A *proof certificate* is created only when there is an inference certificate for the root inference, and there are already proof certificates for all the proofs of the premises of the root inference.

A certificate may fall into doubt when any object it refers to is modified and needs to be reconsidered and modified in this case (see Section 4). Certificates may also be reconsidered by explicit demand.

As certificates only attest to the fact that the objects they refers to satisfy certain policies for creating these certificates, they provide justifications for object contents that are more general than formal proofs in a specific target logic. Nevertheless, they are equally rigorous in the sense that they providing the exact reasons that were used for declaring an object valid.

This aspect is particularly interesting when one considers the possibility that certain inference engines may not be generally accepted. People who do trust a certain inference engine will accept the certificates produced by it while others will insist that the same inferences have to be checked by inference engines they trust. The connection between the FDL and JProver (Section 7.1) accounts for these two levels of trust: one may either trust that matrix proofs produced by JProver are valid, or one may require that the algorithm for translating matrix proofs into sequent proofs be executed and that the results will be checked with a proof checker for the intuitionistic sequent calculus.

5.3 Proof Sentinels

Proof sentinels are used to direct certification of inferences, assembly of proofs from inference steps, and in records identifying inferences and proofs as having been certified accordingly. A sentinel is a term, intended to represent a class of *basic* logical resources and methods.

For example, one might build an inference engine that takes a primitive rule set as a parameter. A sentinel expression appropriate to inference certificates invoking this engine would then indicate the kind of inference engine invoked and the primitive rule set it used.

Another part of the sentinel expression is an indication of when an inference engine itself is acceptable. Therefore, it has to include a method for finding or building individual inference engines of the appropriate kind, as the reason that this inference engine process was trusted in the first place is really that it was identified according to certain proof methods.

To provide for possible extensions of a logic, sentinel expressions should also determines which other sentinel expressions shall be accepted in assembling certain proofs, i.e. they inherit all the inferences passed by those other sentinels. A search for certificates according to a sentinel should normally also find those certificates whose sentinels are inherited by it.

When an inference step is certified, the sentinel expression according to which it was certified is stored as a distinguished component of the inference certificate. When a proof is certified the sentinel expression determines whether the certificate for the step may be incorporated into the certificate for the whole proof.

Because of the external significance attributed to a sentinel expression by a person, persons will normally work with familiar sentinels, which means they need to be sufficiently small as to make it possible for a person to become familiar with those they understand, and not to mistake one for another. A suitable degree of abbreviation can be achieved by allowing liberal use of packaging complex material into objects then referred to by object identifiers, and by allowing liberal use of native language macros.

6 Features of the FDL prototype

A key purpose of building an FDL prototype is to demonstrate that it is possible to build a system with many of the properties called for. Experience with the prototype library will

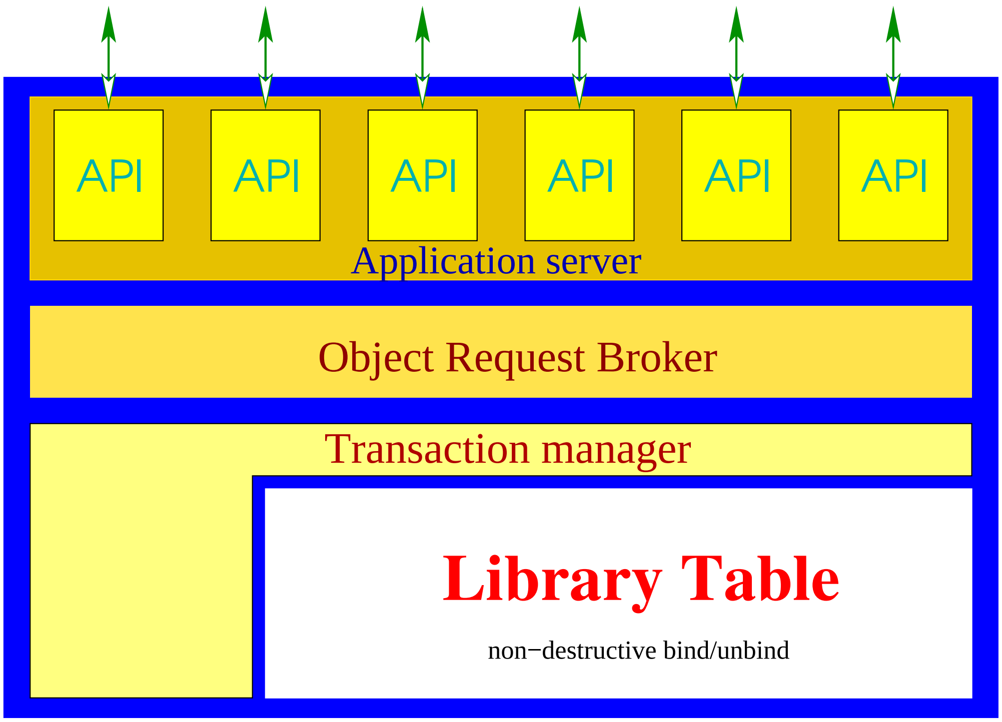


Figure 3: Architecture of the FDL prototype

influence the design and construction of an improved system.

Our prototype implementation of the FDL is organized as a *persistent object store* that adheres to the standards of today's data base technology [Dat02], i.e. to the principles of *atomicity*, *consistency*, *isolation*, and *durability* (*ACID*).

The FDL prototype is centered around the *library table* (Section 3.1). The library table serves as repository for all library content and is responsible for managing access to objects and their abstract identifiers. Its abstract organization prevents clients from accessing and modifying objects without invoking a library process that accounts for their validity.

A *transaction manager* (Section 6.2) supports delete and undo operations in client work spaces and makes it possible to recover from failures. An *object request broker* handles request for accessing the library table. The *application server* (Section 6.3) provides the infrastructure for communicating with external clients and is used to build application specific interfaces for them.

In addition to the basic FDL implementation our prototype also includes a few standard utilities (Section 6.4) that enable users to interact with the library and to develop new formal content for it.

6.1 Library Tables and the File System

After creation, all library contents are stored immediately on the file system. Objects, their properties, and the library table are stored in individual files, whose abstract name corresponds to the internal name of the objects. As object contents cannot be changed, a library file will never be deleted or modified. Instead, modifying an object's contents will cause a new and updated copy to be created and the object's abstract identifier in the library table will point to the new object file.

Thus all previous versions of objects will be preserved unless they are removed by an explicitly enforced garbage collection process. This approach ensures durability of information and replayability of proofs that were accepted by the library. A version control mechanism makes it possible to recover previous versions of an object. This protects user data from being corrupted or destroyed erroneously and enables a user to keep several versions of the same object, while developing the contents of a formal algorithmic theory.

6.2 Transactions

Transactions are a well-established technique to ensure the ACID property of data bases [Dat02]. They provide a model for controlling the outside access to the actual library contents, make sure that the library is always in a consistent state, and provide mechanisms that make it possible to recover from failures and system crashes.

To accommodate the special needs of a formal digital library we have refined the transaction model, so that it can enforce stronger consistency conditions and deal with larger atomic units such as the creation of certificates together with every modification of an object.

All operations that commit changes to the library are based on a small set of *directives* and primitives for *creating new object contents*. Directives determine whether an object is bound in the library table (*bind / unbind*), considered alive (*activate / deactivate*), or permanent, i.e. to be excluded from garbage collection (*allow / disallow*). Primitives for creating new object contents either create new objects from scratch or modify the contents of existing object and store the result in a new object. There is a variety of these primitives for each kind of object, particularly for statements, inference steps, and proofs.

Updating an object in the library table thus involves five basic steps, which have to be finished before the transaction is considered complete: deactivating the object, unbinding its abstract identifier, creating an updated copy, binding the abstract identifier to the new version, and activating it. As each directive has an inverse to undo its effects, an update can simply be undone by reversing the sequence of directives involved, i.e. by rebinding the abstract identifier to the old object, which is much simpler than undoing the actual update operation. Since the updated object content is retained in the library store, redoing a transaction is equally simple.

During a transaction all directives are journaled together with a list of yet uncompleted directives as they are evaluated. Object contents are written to disk at bind time. This makes it possible to recover from crashes by replaying the committed directories in the journal.

6.3 The Application Server

One of the central features that a practically useful FDL prototype has to provide is the ability to connect to clients in an efficient way. In addition to handling requests and answers, this means supplying methods that allow clients to access necessary data efficiently and simply. To accomplish simplicity, the library needs to hide the workings of the transaction system while still presenting a consistent view to the client and allowing a series of request to occur within a single transaction. For efficiency, the client need only see the data which is meaningful to it (e.g. a proof engine may only have to see lemmas that are relevant to proving a theorem) and the client needs to be able cache the data.

In a distributed library the data may be cached by application server supporting the client or by the application itself. If the client caches the data then the library needs to push all modifications made to the data so that the cache is accurate, which results in a large amount of communication. Therefore, our application server contains a *distributed cache*, which for each client maintains a copy of object data that are broadcast by the library and filtered by an application-specific interface. Inactive objects will not be broadcast and are thus invisible to the client.

Usually, the application server presents a serialized interface to a client. The client and server interact via a single thread of requests, notices, and responses. Clients subscribe to certain kinds of information and receive notifications about changes so that they may pull new information from the library if needed. The advantage of this approach is that it supports connections with low bandwidth.

A tighter form of interaction, where a client calls the application server each time new object data are needed, requires high bandwidth connections and an application server written in the client's native language. In this case the application server may be compiled into the client, which enables the client to pre-calculate dependencies and to make very specific requests.

Requests to the library are handled by the object request broker, which separates actual transactions from the interaction with applications. Requests are usually stored in the ORB's queue and processed in FIFO order. This separation allows two modes of operation. In a *synchronous* mode, a client submits its requests and waits for results before proceeding. For instance, a proof engine may need the contents of a specific lemma before it can continue with its proof. In *asynchronous* mode the application only waits for an acknowledgement that its request has been received and periodically checks for notifications. For instance, a user may request a certain proof tactic to be applied to all theorems within a specific theory while continuing to work on other parts of the library, or may want to run several proof engines on the same proof problem in parallel. The difference between these two modes can be expressed by slightly different requests. For synchronous mode the client submits a request to execute a certain process. For asynchronous mode, it submits a request to schedule this process and to send notification upon completion.

To interact with the application server of the FDL, clients have to follow a certain communication protocol that describes allowed sequences of communication as a simple context-free language over "send" and "receive" expressions. All requests, responses, and notices have to be expressed as library terms (see Section 3.1), which have to obey the grammar described in Table 1. Requests to the library, such as looking up the content of a specific object have to be formulated as expressions that can be evaluated and interpreted by the object request broker. Requests to clients, such as performing the inference step described by a tactic, will be formulated in the same format. Responses may be values resulting from evaluating a request, print commands, failure messages, or acknowledgements. The latter acknowledges the receipt of a request that does not expect an immediate answer or is evaluated only for

```
<comm-seq>      :=  (<P1> | <P2>)*

<P1>            :=  SEND<request> <P2> RECV<response>
                  | SEND<notice>

<P2>            :=  RECV<request> <P1> SEND<response>
                  | RECV<notice>

<interrupt>     :=  !interupt{<sequence>:n}

<request>       :=  !req{<sequence>:n, <type>:t}(<expression>)

<response>      :=  !rsp{sequence>:n}(<result>)

<notice>        :=  !msg{<sequence>:n}(<message-term>)
                  | !add{kind}(<object-update> list)
                  | !delete{kind}(<object-id> list)

<object-update> :=  !update{<object-id>:<o>}(<term>)

<expression>    :=  !expression{}(<ap>)
                  | <configure>

<result>        :=  !value(<term> <message-terms>)  { expression }
                  | !print(<term> <message-terms>)  { expression }
                  | !fail(<term> <message-terms>)   { expression }
                  | !ack()

<ap>            :=  !ap<ap-bits>(<term{func}>; <term{arg}>*;)
                  | !unit_ap<ap-bits>(<term{func})

<ap-bits>       :=  {<result-p>:b}
                  | {}

<configure>     :=  !configure(!inform(<rspinfo>))
                  | !configure(!request(<reqinfo>))
                  | !configure(!revoke(<info>))

<reqinfo>       :=  <address{environment}>   { symbolic address }
                  | <start{broadcasts}>      { subscribe }

<revinfo>       :=  <address{environment}>   { symbolic address }
                  | <start{broadcasts}>      { unsubscribe }

<info>          :=  <disconnect>
                  | <start{broadcasts}>      { initial state dump }

<address>       :=  !environment_address{<tok>:t list}

<disconnect>    :=  !disconnect{}
```

Table 1: Grammar for requests, responses, and notices

its side effects.

The FDL application server supports several data formats. In *compressed ASCII format*, terms are converted into their ASCII representation and then compressed to reduce the overhead of communication. Clients that communicate with the FDL in this ASCII format must provide conversion and compression functions that match the algorithms of the FDL. A more efficient format for communicating mathematical data is the *MathBus standard*[1] [Mat], which currently is used in many connections between the FDL and proof engines (see Section 7.1). The FDL also supports a representation of terms in XML, which is more convenient for building web interfaces and is used by approaches like MathWeb [FK99] and HELM [HEL] that aim at building standardized interfaces for communicating theorem provers.

The communication between clients and the FDL requires establishing a connection through standard TCP/INET sockets, which is supported by most programming languages. Clients that do not already include a communication module only have to be extended by a small module that can open and close sockets, read from sockets and write to them (see [Soc, Ste92] for an introduction into writing such modules).

6.4 Utilities

Our FDL prototype comes with a small collection of utilities that are not considered essential but help demonstrating its practical usefulness. These utilities are implemented separately as standard clients of the FDL and may be connected to it on demand.

There are two major interfaces. The FDL *editor* enables a user to inspect and modify formal content at any level of detail. It includes a structure editor for entering and modifying library terms, a proof editor for interactive and tactic-based development of verified knowledge, a visual navigator for browsing, searching, and modifying the library interactively (see our examples in Section 3.2). The editor can interpret library contents and can be customized by adding display forms to the library.

A *web interface* processes library information for publication on the web (Section 8). It analyzes links between formal objects like definitions and theorems to informal text objects and creates articles that present the material on several levels of detail. On the top level, a user will look at a formatted technical report. Clicking on the formal (top-level) text in this document reveals the further details at all levels of precision – from the rough sketch of a theory or proof down to the level of the underlying logic.

Several *proof engines* are connected to the FDL: a sequent style refiner that creates inference steps by interpreting tactics and rule objects, a complete theorem prover for intuitionistic and classical first-order logic, and the proof engine of PVS. The connections to these proof engines are described in detail in Section 7.1.

We also developed a package for connecting the libraries of Nuprl 4, PVS, and MetaPRL to the FDL and for migrating their formal contents into certified FDL theories. The links to these external libraries and the techniques for migrating their contents are described in Section 7.2.

[1] The development of the MathBus format has been supported by previous ONR grants

6.5 Computational Content

One of the main purposes of the FDL is to provide formal content that can be applied in software design and construction. Formalizing a standard body of computational mathematics is a task that has been approached by a many research groups. As the FDL supports the native formal language of almost any proof system it is possible to accumulate the formalized content of a variety of proof environments in a single repository. With the advanced theory mechanisms that will be developed in the future one will then be able to develop new algorithmic content that can use the entire repository in its justifications.

Importing existing formal content into the FDL requires migrating the formalizations of definitions and theorems as well as the formal proofs into the more general FDL format and developing display forms that make the FDL representations look like expressions of the original formal language.

Our FDL prototype initially included only the complete type theory of the Nuprl 5 system [ACE$^+$00, NuPa, ML84] together with its standard theories. In the past year we have migrated all user theories developed with its predecessor Nuprl 4 into the more general FDL format. These include elementary number theory, discrete mathematics, general algebra, finite and general automata, basics of Turing machines, and the formal development of hybrid communication protocols. A complete documentation of these theories can be found at `http://www.cs.cornell.edu/Info/Projects/NuPrl/Nuprl4.2/Libraries/Welcome.html`

More recently we have developed mechanisms for importing the theories of the PVS system [ORR$^+$96, PVS] into the FDL. We have used these methods for migrating all 79 theories of the PVS prelude and the complete graphs library. Further theories will be imported in the near future. Having PVS content available in the FDL makes the FDL accessible to the large community of PVS users.

We have also built a formal representation of constructive ZF set theory (CZF) and linked it to Nuprl's type theory. This makes it possible to represent many mathematical theories in a formalism that mathematicians are familiar with while making them available in formal proofs that involve computational type theory.

In MetaPRL [Met] we have developed a framework for establishing semantical links between different formal theories. This makes it possible to proof theorems by referring to "acceptable" formal knowledge developed in a different theory or proof system without having to re-prove that theorem. Using the mechanisms for migrating formal content (Section 7.2) we will integrate this framework into the FDL library.

7 Connecting Theorem Provers and Logical Frameworks

One of the central goals of implementing a digital library of formal algorithmic knowledge is to provide an infrastructure for interoperability between different proof systems that will enable people who work with similar, but different formalisms to cooperate in the development of new certified knowledge.

To integrate different proof systems and the formal theories that have been developed with them, we have to connect them to the FDL by providing the appropriate API modules, and to develop mechanisms for automatically migrating formal content into the FDL library.

7.1 Proof Engines

Proof engines provide mechanisms for interactive, tactical, and fully automated reasoning in a specific formal language and generate justifications for the formal content stored in the FDL. Currently we have connected the following inference engines to our FDL prototype.

- The Nuprl refiner [ACE+00, NuPa] supports interactive and tactic-based reasoning in computational type theory within a sequent calculus framework. To connect it to the FDL we have developed an API module that communicates proof goals, tactic names and their parameters to the Nuprl refiner and modifies the current proof object according to the results it receives from the refiner. Mathematical data are communicated in MathBus format, which makes it possible to reconstruct terms from the data received without having to parse them.

  We also have imported all the basic rules, tactics and theories on which they depend into the FDL, which makes it possible to control the refiner's behavior through code objects in the library.

- JProver [KOS96, SLKN01] is a complete theorem prover for intuitionistic and classical first-order logic. Its proof search procedure [OK95, OK96, KO99] is an extension of *matrix methods* developed by Andrews and Bibel [And81, Bib81]. When JProver successfully proves a formula F, it produces a *reduction ordering* for F that consists of the formula tree together with ordering constraints induced by substitutions. JProver also includes an algorithm for converting this reduction ordering into a sequent style proof [SK95, KS00], in which each individual proof step is justified by a basic inference rule.

  To connect JProver to the FDL, we have developed an API module that converts FDL contents into the language of JProver and vice versa. The module also translates FDL language features (like type information) that are outside the range of first-order logic into abstract predicates that can be handled by JProver and reconstructs these features when rebuilding the sequent proof it receives from JProver. On the side of JProver, this module is complemented by a small code module that establishes a communication with the FDL over the net in MathBus format.

- The PVS system [ORR+96, PVS] provides mechanized support for formal specification and verification based on a classical, typed higher-order logic. It supports interactive reasoning and proof scripts that build sequent style inferences from primitive inference rules, induction, rewriting, and decision procedures.

  To connect PVS to the FDL we have implemented an API module that uses display forms to present FDL terms in PVS notation, sends sequents and PVS commands to the PVS proof engine, and builds FDL terms from the results. Mathematical data are communicated in text format, which makes it possible to use PVS without any modifications.

In the future we will connect further proof engines such as the HOL proof system [GM93], the proof-based MinLog [BBS+98] program generator, Isabelle [Pau90], and even Larch [Lar] and Automath [Bru80].

7.2 Linking and Migrating Libraries

Over the past decade a substantial body of formalized mathematical and algorithmic knowledge has been developed with proof systems like PVS [ORR+96], Nuprl [ACE+00], MetaPRL [Met], MinLog [BBS+98], HOL [GM93], Coq [Dea91], and Isabelle [Pau90]. Each of these systems uses a different formalism and none of them contains all the currently available formal knowledge.

In order to use the FDL as a common repository, we have to link these proof systems to the FDL and to migrate the content of their libraries into the FDL library, i.e. converting formal expressions into the FDL format and constructing inference trees, proofs, and certificates from the proofs build with the respective systems. Currently we have developed migration packages for the following systems.

- Since our FDL prototype evolved out of the library of the Nuprl 5 system [ACE+00, NuPa], the structure of Nuprl 5 terms is similar to the one of basic FDL data (Section 3.1). Migrating Nuprl 5 theories into the FDL can therefore be based on the mechanisms for exporting and importing theories described in Section 3.2, which only have to rebuild certificates.

- The MetaPRL [HN00, Met] is a logical framework that supports interactive and automated reasoning. MetaPRL supports multiple logics (i.e. CZF, ITT), and each logic is organized into theories, or modules, and each theory contains theorems, rules, and display objects.

  To migrate content from MetaPRL into the FDL, we convert each system's data to a common MathBus interchange format, and send the MathBus terms over TCP sockets. The MetaPRL logics that a user is interested are specified during the build of MetaPRL. After the FDL is connected to MetaPRL, one can retrieve the modules of those logics, and their contents. Commands and their arguments are sent to MetaPRL from the FDL, which specify what to import, how, and additional evaluation requests. Example commands include listing all modules, retrieving a particular proof in a module, calling the proof engine on a particular proof step, or migrating an entire module, or logic.

  For the purpose of the FDL, we typically desire to migrate all data. Then, we check the proofs by calling the MetaPRL proof engine and build the appropriate certificates.

- Users of the predecessors of the current Nuprl system have developed a substantial amount of formalized knowledge. To integrate this knowledge into the FDL we have developed a migration package that converts Nuprl 4 data into the more general FDL format, checks formal proofs with the Nuprl refiner, and builds the appropriate certificates.

- The PVS system [ORR+96, PVS] supports formal reasoning in a classical, typed higher-order logic. Users of PVS have developed large repositories of formal knowledge about the formal specification and verification of software. To migrate PVS theories into the FDL library, we connected the PVS proof engine to the FDL as described above and connected it to a "PVS parser", that builds FDL terms from ASCII representations of

PVS expressions. PVS theory files are converted into FDL theories by converting definitions and statements with the PVS parser and re-executing proofs with the connected PVS proof engine in order to build the necessary certificates.

In the future we plan to migrate formal content developed with other proof systems such as HOL, Coq, MinLog, Isabelle, and Larch.

8 Publishing and Reading

One of the key services that a library must provide is an interface that makes formal algorithmic knowledge accessible to users. As we envision a variety of users who would benefit from being able to inspect the contents of a digital library of formalized mathematicl and algorithmic knowledge, we have developed a *publication mechanism* that enables external users to access the logical library and to browse its contents without having to run a local copy of it.

In [Nau98] we have made a first step towards publishing our formal mathematics on the web. But our interface goes beyond being a simple web browser, which allows users only to browse through some pre-formatted text version of the formalized knowledge. It also supports viewing formal contents at all levels of precision – from the rough sketch of a theory or proof down to the level of the underlying logic. We expect that the ability to unveil formal details on demand will have a significant educational value for teaching and understanding mathematical and algorithmic concepts.

We have built a large utility for converting related collections of objects to HTML, along with automatically generated structure-revealing documents and auxiliary annotations. The structure of the web material is thoroughly explained on the several documentation pages to which one may link via the rightmost `Doc` link on every page so created. This link is `http://www.cs.cornell.edu/Info/People/sfa/Nuprl/Shared/doc.html`

This preparation of the HTML documents consists largely of taking scripts for posting library sections and adapting them to the a new section in question. The basic parameters stipulated are:

- What is the section called?
- Where are the pages to be put?
- What sections logically precede this one?
- What is the principle context above the section?
- What are the section's objects, i.e. what objects are to be included among the web pages as originating in this section? (There are a number of filtering utilities that help weeding out "unimportant" objects)
- Do any objects need to be modified specifically for presentation and how?
- Are there any remarks to be made for the reader as to how the section originated?
- Are there special links you want added to every page?
- Are there certain objects whose (postscript) print form should be left ungenerated?

- Should the listing of the section be used as the front page or has a more readable page been provided as an introduction to the section?

When the utility is run, the various browsable pages and some print forms are generated, missing links are reported diagnostically, and a decision is rendered about whether the result is suitable for posting. As most of the above stipulations can be specified as display forms and command objects of the library, the process of generating HTML documents has been automated to a large extent.

Currently, the publication mechanisms are capable of handling formal content created by Nuprl. Showing new kinds of content, such as PVS libraries, requires appropriate modification and specialization. Not only might the presentation of each object content need adjustment, but the relations to be revealed between documents must be determined and accommodated.

9 Acknowledgments

This work was supported by ONR Grant N00014-01-1-0765 (Building Interactive Digital Libraries of Formal Algorithmic Knowledge) and in part by DARPA grant F 30620-98-2-0198 (An Open Logical Programming Environment)

We are very thankful to William Arms, Alan Demers, Johannes Gehrke, Paul Ginsparg, John Kleinberg, and Carl Lagoze for valuable discussions and feedback.

A Glossary

The following one-sentence descriptions will be explained in detail below.

Abstract Identifiers: identifiers treated abstractly and as atomic, serving as object names in a closed map.

Assertion: an expression used as a conclusion or premise of an inference step.

Basic Value Injections: the constituents of terms that are not themselves terms.

Certificate: an object attesting to some fact established by the library process.

Closed Map: a finite collection of named object contents; object can refer to objects.

Current Closed Map: the main part of the state in a session with a client of the library, being a distinguished, variable closed map.

Definition: an explicit eliminable definition of a mathematical operator or concept.

External Name: a concrete name whose association to abstract identifiers is maintained by the library.

Formal: having precise meaning or objective criteria of correctness, ideally computer verifiable, based simply upon "syntactic" form.

Inference Engine: a process that can verify (or generate) an inference Step

Inference Step: expression of an inference from zero or more premises

Justification: data provided to an inference engine in an inference step in addition to its assertions.

Library: a repository of a certain kind with a process for using it.

Native Language: a programming notation executable by the library process.

Object: the unit of library content; abstractly named expressions.

Proof: a complex of appropriately related inference steps.

Refiner: an inference engine that computes premises from a conclusion and justification.

Sentinel: an expression occurring in a certificate validating an inference step identifying which primitive logical resources are permitted in justifying it.

Tactic: a program describing an inference by composing primitive inferences.

Term: the main form used in the library for structured data such as expressions.

A.1 Abstract Identifiers

Abstract Identifiers are identifiers that are treated abstractly and as atomic, serving as object names in a closed map. The only basic operations on abstract identifiers are testing equality between them, incorporating them into terms where they serve as object references, and operations on them in their capacities as object references such as object content lookup. An abstract identifier cannot be constructed from any other values or be distinguished except by atomic comparison to other abstract identifiers.

A.2 Assertion

Assertions are expressions used as a conclusion or premise of an inference step. An assertion might paradigmatically be thought of as a statement with a truth condition that is understandable independently of inferences in which it occurs, but it is also possible that the significance of an assertion may depend on an inference in which it occurs.

It is not part of this library design what the structure of an assertion must be, except that it must be an expression of the general form used in the library, i.e. a term. When a new kind of inference engine is introduced into the library one must design the form of assertions to be used in inferences by that engine. If one were concerned solely with the individual inference then one would have an awful lot of freedom in deciding what should go into the assertion and what should go into the justification . The basic constraint on this design is imposed by the rather natural fact that inference steps are assembled into a proof based upon the match between assertions used as premises and conclusions, and if the same inference engine is used on a collection of inferences that can be formed into a rooted dag based on conclusion/premise matching, then the root conclusion is deemed to be a consequence of all the leaf premises.

A.3 Basic Value Injections

Basic Value Injections are the constituents of terms that are not themselves terms. Examples would be integers, strings and abstract identifiers.

A.4 Certificate

As the name suggests, a certificate (Section 5.2) attests that certain library actions were taken at a certain time; they are the basis for logical accounting. A certificate will be realized as an object which can then be referenced and accessed like other objects save for certain constraints. A certificate cannot be created or modified except by the library process following a procedure specific to the kind of certificate in question.

Although certificate contents are expected to often be rather compact, largely consisting of object references, they will often also be rather expensive to establish. By realizing certificates as objects the library can build certificates that depend on others whose correctness is independently established. Thus one process of certification can contribute to many other certifications without having to be redone.

The paradigmatic certificates are those created to validate proofs. An inference step certificate is built by applying a specified inference engine to an inference, and including in the certificate references to the inference step as well as to the instructions for building or deploying the inference engine; the certificate attests to the fact that that such an engine accepted that inference. A proof is a rooted dag of inference steps. A proof certificate is created only when there is an inference certificate for the root inference, and there are already proof certificates for all the proofs of the premises of the root inference.

A.5 Closed Map

A closed map (Section 4.1) is a map of type $D \rightarrow \texttt{Term}(D)$ from some finite index set D to object contents. Reference between objects consists of the occurrence of the referent's index in the referring objects content. Object indices are treated as abstract identifiers. The library is used as a repository for closed maps, and the usual method of interaction is to build and develop a current closed map, which may be thought of as a closed map variable serving as the focus of a session with the library.

A.6 Current Closed Map

The Current Closed Map (Section 4) is a distinguished, variable closed map that characterizes the main part of the state in a session with a client of the library. A client can preserve the current closed map for later access, modulo uniform change of object identifiers.

A.7 Definition

A definition an explicit eliminable definition of a mathematical operator or concept that does not have any further epistemic content. The meaning of an expression should remain unchanged when an a definiendum is replaced by it definiens. In a program source this is meant to be like a macro; in a mathematical discourse, the intention is that the truth value or provability of any assertion is preserved by definition elimination or introduction, although the proof itself may require modification with regard, for example, to whether the definiendum is mentioned in inference specifications.

While the creation of a definition provides a new, though semantically shallow, resource for possible use in expression or argument, no substantial epistemic content is implied by the definition; this is in contrast to the addition of axioms or primitive rules of inference. The same goes for theorems and derived rules of inference.

A.8 External Name

An External Name is a concrete name whose association to abstract identifiers is maintained by the library.

While the principle name space used by the library consists of abstract identifiers, a relatively small number of identifiers must be maintained that make sense outside the library process in order to identify library entities outside the library process. When connected to the library, clients can refer to abstract identifiers and communicate them via the library to other connected clients. But in order for clients to communicate identifiers outside the library, one must have a concrete substitute that can be later resolved when connected to the library. For example, one person may wish to tell another person via e-mail how to find an object in the library (from which many other objects can be found); to do so the first person simply needs to have the library attach a concrete name to the object, and communicate it to the second person, who then requests the object associated (perhaps temporarily) with the received concrete name.

A.9 Formal

In the context of this project "formal"means having a precise meaning or objective criteria of correctness, ideally computer verifiable, based simply upon syntactic form. This is the sense of the word most relevant to our endeavor, and is perhaps the most common sense of "formal" used in the literature of logic and analytic philosophy.

This usage stands in contrast with other common meanings as being rigid, ceremonious, solemn, customary or not casual. It is closer to meaning exact, methodical or orderly, but for the purposes of supporting precise understanding or procedures that can be performed by machines. By "informal" we simply mean not formal, in this sense.

A.10 Inference Engine

An Inference Engine is a process that can verify or generate an inference step. The library process builds and applies inference engines according to instructions stored in the library and specific to the kind of engine. One extreme would be creating a process from scratch on a local machine according to instructions; another extreme would be to simply communicate with an already existing process over the internet. Naturally, different inference engines can be trusted to different degrees not only because of the varying inferences they are intended to check and who programmed them, but also because of the varying reliability of the mechanisms used to run and communicate with them.

Often one tends to think of formal inference steps as small and schematic, but inference engines, such as tactic based engines, can be built that verify arbitrarily complex and non-schematic inferences.

A.11 Library

We imagine a Formal Digital Library as a repository for formally verified material along with other complementary material,much of which is essential for practical use of the formal. The complementary material is either unverified, only partially verified, or perhaps not subject to verification by machines. Further, we conceive of the library as a process for maintaining, accessing, and modifying the repository. One should expect that multiple independent libraries of this sort will be created, some maintained long term, others quite temporarily, and that they should be able to communicate their content usefully.

Among the content of a library are specifications for how to verify formal content. It is not part of the library design to govern what counts as legitimate validation, and indeed libraries are intended to be radically open. This requires accounting for what has been verified and by what means.

Different implementations of libraries are possible, and they may provide different services beyond the minimal, as well as have their own policies for access and contribution.

A.12 Native Language

The Native Language is a programming notation executable by the library process. Clients of the library must be able to stipulate programs executed by the library process. Request for execution of such programs and returning their results is a basic interaction between clients and the library. Most work of certifying inference steps is expected to be done outside the library by inference engines, the library simply invoking those engines and recording the results. A native language should provide generic computational methods (the glue) as well as some basic library-specific operations for manipulating ones current closed map, for managing a small external name space, for control of access to ones own objects by other clients, and for establishing and communicating with external processes.

The execution of native language programs, which are terms, is implemented as part the library, and forms the basis of certification; the facts to which a certificate attests are simply that certain native language programs were executed to certain effect.

There may be multiple native languages, suitable for different styles of programming by customers. For example, a higher-order functional style and a conventional imperative style language would be basic candidates, and perhaps a virtual machine for use by those clients who prefer to develop their own languages for execution by the library.

A generic native language macro facility is provided as well. These macros are expanded by simple match against a left-hand-side term whose immediate subterms, and perhaps some constituent basic value injections, are then substituted into a right-hand-side.

A.13 Object

An Object is the unit of library content consisting of abstractly named expressions. The notion is technically dependent on that of closed maps. With respect to a closed map, an object is identified with an index value. The principal division of objects is into Certificate objects and all others. The content of an object is a term and the object is identified by an index in the closed map.

A.14 Proof

A Proof is a complex of appropriately related inference steps. We assume the basic form of proof is a dag (directed acyclic graph) of inference steps, where the conclusion of a child inference is a premise of its parent inference. An essential part of what makes a proof convincing is that all the individual inferences are verified by a convincing inference engine or a compatible collection of inference engines. The notion of assertion is that it is the unit of matching adjacent inferences in a proof. If the proof is a rooted dag then we may construe the proof as deriving the root conclusion from the leaf premises.

That some dag of inferences is a proof constrained to certain methods is a matter of certification. It is intended that multiple, diverse, even incompatible criteria for acceptable inference be permitted to coexist in the library, and so one must be careful when certifying a proof that one limits ones inferences by stipulation to acceptable and compatible methods.

One can devise secondary forms of proof with more structure from which a basic proof (dag) can determined. Such a proof can be considered as a presentation of a basic proof. For example, one might want to use a block style natural deduction organization of inferences, where the assertions at each inference are derived partly from the whole block-form proof as a context.

A.15 Refiner

A Refiner is an inference engine that computes premises from a conclusion and justification. Refiners are used to develop proofs top-down, but of course they can be used after-the-fact to validate a whole inference step by computing what the premises should be according to refinement, then comparing the actual premises to the expected ones. Many refiners are tactic based provers.

A.16 Sentinel

A Sentinel (Section 5.3) is an expression occurring in a certificate validating an inference step identifying which primitive logical resources are permitted in justifying it. The connotation of "sentinel" is that it guards against the intrusion of untrusted entities into an inference, which is essential in a library that comprises multiple incompatible logics. The sentinel expression constitutes the criterion of coherency among inferences organized into a proof.

A.17 Tactic

A Tactic is a program describing an inference by composing primitive inferences. Some inference engines are tactic based provers. They are significant as examples of systems that can accept wide variety of complex inference steps , and that can be rather expensive to run due to the fact that tactics may be programs built in a full general purpose programming language. When such an inference engine is built, one typically builds a state with a lot of procedures and data built in.

In the following explanation of what tactics are, the notion of proof is internal to the tactic prover, and is not presumed to that of proof as used in the library . The principal use of the tactic prover by the library process is simply as a source of individual inferences.

Given a collection of prespecified primitive inference forms, a tactic is a program for reducing a desired proof goal to premises by composing primitive inferences. A tactic is essentially a program for constructing such an inference tree, and one chooses which tactics to apply according to how you want to generate subgoals from the goal. The execution of the tactic gives rise to an inference step, the premises being all the unproved leaf premises of the primitive proof tree. Further, to count as a tactic, although its execution might not terminate or might raise an exception, if it does terminate without exception, then it must be guaranteed to generate subgoals justifiable by primitive inferences.

Returning to the concept of library proof, an alternative use of a tactic prover by the library would be to call the tactic prover's bluff and demand that the tactic prover produce for the library the smaller inferences that it claims existed. A tactic prover providing this alternative access by the library process could then be double checked in order to provide an independent verification of the original complex inference.

The appropriate form of justification used in an inference step to be submitted to a tactic prover is the tactic code for it to execute. An inference engine that generates its premises from conclusion and justification, as in the process described above, is called a Refiner .

A.18 Term

A Term is the main form used in the library for structured data such as expressions. Terms used as library content are simple recursive structures, i.e. they are abstract structures rather than text strings. We take issues of parsing strings into structures and displaying structures to users to be matters ordinarily extraneous to criteria pertaining to formal properties of expressions such as criteria for correctness.

The term structure is iterated operators on subterms. In addition to its subterms, a term contains a sequence of labelled values presupposed by the construction of terms, which we call basic value injections; character strings and abstract identifiers are among these basic values. An individual term consists of a sequence of zero or more basic value injections together with a sequence of zero or more immediate subterms; further, any such pair of sequences constitutes a term. The sequence of basic value injections may be construed as identifying the operator of which the term is an instance, or sometimes this sequence together with the number of subterms is construed as the operator.

Binding structure, however, i.e. which expressions are variables and which become bound in which terms, is not considered part of the term structure as far as the library is concerned. Binding structure is attributed to terms by the clients of the library. The reason for this is that there are significantly divergent approaches to binding structure, and to build one in would be an unacceptable bias. We believe that issues of binding structure remain open among different groups or prospective library clients, that they will be the subject of further innovations, and that disputes should be waged among library clients and not between library designers and clients.

References

[ACE+00] Stuart Allen, Robert Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In D. McAllester, editor, *17th Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[And81] P. Andrews. Theorem-proving via general matings. *Journal of the Association for Computing Machinery*, 28(2):193–214, 1981.

[BBS+98] Holger Benl, Ulrich Berger, Helmut Schwichtenberg, Monika Seisenberger, and Wolfgang Zuber. Proof theory at work: Program development in the minlog system. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II, chapter II.1.2, pages 41–71. Kluwer, 1998.

[BBS99] C. Benzmüller, M. Bishop, and V. Sorge. Integrating TPS and Ωmega. *Journal of Universal Computer Science*, 5, 1999.

[Bib81] Wolfgang Bibel. On matrices with connections. *Journal of the Association for Computing Machinery*, 28:633–645, 1981.

[Bru80] N. G. De Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, New York, 1980.

[CAB+86] Robert L. Constable, Stuart F. Allen, H. Mark Bromley, W. Rance Cleaveland, J. F. Cremer, Robert W. Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, Jim T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL proof development system*. Prentice Hall, 1986.

[CDE+99] Manuel Clavel, Francisco Duran, Steven Eker, P. Lincoln, N. Marti-Oliet, Jose Meseguer, and J.F.Quesada. The Maude system. In P.Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications (RTA'99)*, number 1631 in Lecture Notes in Computer Science, pages 240–243. Springer Verlag, 1999.

[Con96] Robert L. Constable. Creating and evaluating interactive formal courseware for mathematics and computing. In Magdy F. Iskander, Mario J. Gonzalez, Gerald L. Engel, Craig K. Rushforth, Mark A. Yoder, Richard W. Grow, and Carl H. Durney, editors, *Frontiers in Education*, Salt Lake City, Utah, November 1996. IEEE.

[Dat02] C. J. Date. *Introduction to Database Systems*. Addison Wesley, 2002.

[Dea91] G. Dowek and et. al. *The Coq proof assistant user's guide*. Institut National de Recherche en Informatique et en Automatique, 1991. Report RR 134.

[Dil96] David Dill. The Murphi verification system. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV'96)*, Lecture Notes in Computer Science, pages 390–393, 1996.

[FK99] A. Franke and M. Kohlhase. MathWeb, an agent-based communication layer for distributed automated theorem proving. In H. Ganzinger, editor, *16th Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, 1999.

[GM93] Michael Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[HEL] HELM: Hypertextual electronic library of mathematics. `http://www.cs.unibo.it/~asperti/HELM`.

[HMBL99] Amanda M. Holland-Minkley, Regina Barzilay, and Robert L.Constable. Verbalization of high-level formal proofs. In James Hendler and Devika Subramanian, editors, *Sixteenth National Conference on Artificial Intelligence*, pages 277–284. AAAI Press, 1999.

[HN00] Jason Hickey and Aleksey Nogin. Fast tactic-based theorem proving. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 252–266. Springer Verlag, 2000.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[KO99] Christoph Kreitz and Jens Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5(3):88–112, 1999.

[KOS96] Christoph Kreitz, Jens Otten, and Stephan Schmitt. Guiding program development systems by a connection based proof strategy. In M. Proietti, editor, *Fifth International Workshop on Logic Program Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Science*, pages 137–151. Springer Verlag, 1996.

[KS00] Christoph Kreitz and Stephan Schmitt. A uniform procedure for converting matrix proofs into sequent-style systems. *Journal of Information and Computation*, 162(1–2):226–254, 2000.

[Lar] Larch home page. `http://www.sds.lcs.mit.edu/spd/larch`.

[LSBB92] Reinhold Letz, Johann Schumann, Stephan Bayerl, and Wolfgang Bibel. Setheo: A high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.

[Map] Maple home page. `http://www.maplesoft.com/`.

[Mat] The MathBus Term Structure. `www.cs.cornell.edu/Info/Projects/NuPrl/mathbus/mathbusTOC.htm`.

[McC97] W. McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19:263–276, 1997.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Met] Metaprl home page. `http://metaprl.org`.

[Miz] Mizar home page. `http://www.mizar.org`.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory Lecture Notes*. Bibliopolis, Napoli, 1984.

[Nau98] Pavel Naumov. Publishing formal mathematics on the web. Technical Report TR98-1689, Cornell University. Department of Computer Science, 1998.

[NO79] Greg Nelson and Derek. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NuPa] Nuprl home page. `http://www.cs.cornell.edu/Info/Projects/NuPrl`.

[NuPb] Nuprl web libraries. `http://www.cs.cornell.edu/Info/Projects/NuPrl/Nuprl4.2/Libraries/Welcome.html`.

[OK95] Jens Otten and Christoph Kreitz. A connection based proof method for intuitionistic logic. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *4^{th} Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, volume 918 of *Lecture Notes in Artificial Intelligence*, pages 122–137. Springer Verlag, 1995.

[OK96] Jens Otten and Christoph Kreitz. T-string-unification: Unifying prefixes in non-classical proof methods. In U. Moscato, editor, *5^{th} Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, volume 1071 of *Lecture Notes in Artificial Intelligence*, pages 244–260. Springer Verlag, May 1996.

[ORR+96] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 411–414, 1996.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Piergiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[Pit99] President's information technology advisory committee report to the president. Information Technology Research: Investing in Our Future, February 1999. `http://www.ccic.gov/ac/report/`.

[PVS] PVS home page. `http://pvs.csl.sri.com`.

[QED] The QED project. `http://www-unix.mcs.anl.gov/qed`.

[Sch99] Fred Schneider. *Trust in Cyberspace*. National Academy Press, 1999.

[Sho84] R. E. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984.

[SJ95] Y. V. Srinivas and Richard Jüllig. SPECWARE: Formal Support for composing software. In *International Conference on the Mathematics of Program Construction*, 1995.

[SK95] Stephan Schmitt and Christoph Kreitz. On transforming intuitionistic matrix proofs into standard-sequent proofs. In P. Baumgartner, R. Hähnle, and J. Posegga, editors, *4^{th} Workshop on Theorem Proving with Analytic Tableaux and Related Methods*, volume 918 of *Lecture Notes in Artificial Intelligence*, pages 106–121. Springer Verlag, 1995.

[SLKN01] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Alexey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In R. Gore, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 421–426. Springer Verlag, 2001.

[Soc] Introduction to socket programming. `http://www.linuxgazette.com/issue47/bueno.html`.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison Wesley, 1992.

[SVC] The Stanford Validity Checker home page. `http://verify.stanford.edu/SVC/`.

[Wol88] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison Wesley, 1988.

[WWM$^+$90] L. Wos, S. Winker, W. McCune, R. Overbeek, E. Lusk, R. Stevens, and R. Butler. Automated reasoning contributes to mathematics and logic. In M. E. Stickel, editor, *10^{th} Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 485–499. Springer Verlag, 1990.