Steps Toward a World Wide Digital Library of Formal Algorithmic Knowledge [1]

Robert L. Constable
Stuart Allen
Mark Bickford
James Caldwell
Jason Hickey
Christoph Kreitz

[1]This work was supported by the DoD Multidisciplinary Research Initiative (MURI) program administered by the Office of Naval Research under grant N00014-01-1-0765

Contents

Preface

This book describes the results and plans for future work of the Cornell/ Cal Tech/Wyoming Multidisciplinary University Research Initiative (MURI) project entitled:

Building Interactive Digital Libraries of Formal Algorithmic Knowledge.

The book addresses these topics: the objectives of our research, the scientific and technical significance of the project, our technical approach and accomplishments, scenarios for use of such digital libraries, and the relevance of our work to DoD and Navy missions.

This MURI project was created as part of a Department of Defense (DoD) initiative to increase the reliability and adaptability of software. The topic is important because unreliable and insecure software poses a significant national security risk. The nation's critical software infrastructure is open to terrorist attacks, which could be coordinated with those against the population and the physical infrastructure on which they depend; simultaneous, well-planned attacks of this kind could have a devastating impact.

The liabilities of unreliable software have been recognized since the "software crisis" of the 1970's. The cost to the economy, and the cognitive disruptions caused by poor software, remain serious problems, now more significant because of the ubiquitous nature of software. Society's response since the 70's has included: an elaborate testing methodology, now well established but limited in effectiveness; a call for improved education, now heeded and also of limited effectiveness; a call for better software engineering and more science to support it, also heeded; and finally, modest exploration of advanced development and checking tools such as model-checkers, program analyzers, and interactive theorem provers.

In some cases, these advanced tools have become established in industry, especially in hardware design and in the front ends of compilers. Gradually the outlines of a "verification technology" are emerging as a result of steady

DoD, NASA, and NSF funding; and there are deep results from computer science and mathematics that give it direction and justification. Now that the stakes are higher, the need to explore advanced technologies is more urgent. It is clear that the United States must lead in such a technology, and the more advanced it is the better protection it will afford against terrorists. In some ways, the protection of cyberspace is similar to the exploration of outer space; it requires very advanced science and technology, and it offers enormous strategic advantages.

The authors of this book have worked on verification tools and the science behind them for many years. We have known for some time that a key piece of technology is missing – namely a large machine accessible information resource. The DoD also recognized this gap and our MURI project addresses the problem directly.

The important role of knowledge, and the information resources from which it arises, is clear from the behavior of the best system designers and programmers. They bring extensive knowledge to bear in their reasoning about designs and code. Large teams share knowledge, and reason in a coordinated yet distributed manner. However, the tools used to help them reason and check their inferences do not have access to significant amounts of knowledge. For instance, the interactive theorem provers that are designed to assist in reasoning have access to only a few thousand facts, a stark contrast to the computing cycles they could (but seldom) use. They should have access to hundreds of thousands of facts. The few people who drive the provers are the main source of knowledge, and they feel compelled to formalize much of it before they are comfortable using it.

The time is right to begin creating a massive information resource. First, the time is right because we need it, but additionally, the past work of multiple groups world wide has accumulated about 50,000 formal definitions and formally proved theorems. Much of the material is about data structures, algorithms, and computer systems. In addition, the Web and the emergence of information science have given us new tools for this hard task. Currently the knowledge is fragmented; no one system or one verification group has access to more than a small fraction of it. It is a terrible waste of time and huge cost for each group to try to recreate every result for themselves. For many reasons, it is not practical to limit the choices of formalism to apply. Instead, we believe that we must facilitate logically sound combinations of results, and we must be able to account for the correctness of these hybrid results. Our goal is to make it possible for people and machines to share this knowledge. Achieving this goal is a hard problem with a large payoff — worthy of MURI funding.

The problem is hard because there are no standard parts, and we don't know how to make them. Indeed it may be that for many aspects of reasoning no unique standards will ever become established. In addition, the mechanisms for efficient machine processing of knowledge make accounting for evidence and truth difficult. It should be noted that there might be advantages to having mutliple independent logics and systems involved in checking critical arguments.

Our approach is to enable different systems to share results from distinct theories and logics, and to account for these differences precisely. We also recognize the need to connect the formal knowledge to intuitive human knowledge in transparent ways. This connection introduces mechanisms that are flexible enough to allow sharing, yet precise enough to guarantee correctness. The connection to intuition at all levels of abstraction is a safeguard, as well as an opportunity to use the formal knowledge to support education and software engineering, and thus testing as well.

This book explains our technical approach and research results from the past 27 months, and it lays out our plans for the next 30 months.

Chapter 1

Introduction

1.1 Emergence of Information Science

Our work involves building and experimenting with a digital library of machine checked algorithmic mathematics; we call machine checked mathematics *formal.* The appeal of the research agenda for this field is that computers help us create knowledge and accelerate discovery by processing the information on which knowledge is based. Computers reveal patterns invisible to an unaided mind; they can check claims against vast collections.

This enterprise was only vaguely imaginable a decade ago because there was so little formal mathematics available. Now, in 2003, there are on the order of 50,000 theorems along with their proofs and definitions. Nearly 5,000 new ones are added per year. Much of this material results from work on specifying and verifying computer hardware and software systems, thus about half of the existing content is closely related to algorithms, data structures, and computer systems. (There might be another 20,000 uncollected results that we call *dark matter.*) This corresponds to over 500 books about computer science, mathematics, and the science of programming under the assumption that a typical book has a hundred theorems with proofs. As far as we know, these results are correct in every detail with respect to particular logics.

The recent emergence of our enterprise and similar ones to collect all genomes, all protein structures, and "all articulated human knowledge" in digital format signals the emergence of a new science which is being called *information science* (it is a NATO science category as well as a new major at some universities). The subfield of computer science called information organization and retrieval was a precursor, yet even its leading practitioners did not imagine the scope of the activity and its galvanizing power across science and scholarship.

Computers can not only assemble evidence and check extremely detailed arguments, but now they can suggest new ideas by surveying more information than even very large groups of people can. In the case of computer science and mathematics, they can not only check every step of a long complex argument, but they have been used in essential ways in making discoveries by symbolic computing [89, 135, 123, 71] and they have transformed proofs of one kind of assertion into proofs of other kinds. They have systematically transformed small theories by propagating changes of definitions through all theorems and proofs. This is a template for systematically modifying a system module or class along with its full formal documentation.

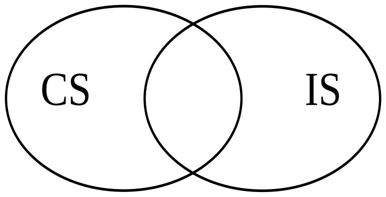
Computers have been executing algorithms, their procedural knowledge, from the beginning. Gradually this procedural knowledge has been turned to support the declarative knowledge that we use to organize our thoughts. For example, the use of hypertext is now pervasive because it is a remarkably effective way to support the associative structure of ideas intrinsic to human thought. This form of interdocument reference is critical to our formal digital library. It is especially suitable to the structure of mathematical documents.

Interdocument reference structure turns out to provide a semantic basis for processing information, e.g. mathematical symbols can be references to documents explaining their meaning. The reference structure is a *latent semantics* for a collection, and it enables algorithms to create knowledge automatically on the collection. In the case of mathematics, the formalization creates another rich associative knowledge structure — a formal one. Here symbols are connected to their defining axioms, formulas and theorems. Relating the formal knowledge structure to the latent one is becoming a source of important research questions previously unasked for lack of a framework in which they were precise. Already these investigations have yielded new information science techniques that we are using in our project. Note, work on the *semantic web* [22] shows that these questions are significant outside the realm of formal mathematics.

The digitization of mathematical knowledge has opened the way to quantifying features of the mathematical landscape. We can measure the depth of formal proofs and count all proofs of more than n lines. We can numerically quantify the value of a reorganization of concepts and the numerical value of a new primitive or a new logical feature, say in producing fewer concepts or shorter proofs. These ideas apply to theories in our formal digital library that are used to explain a software system. In due course we will have enough numerical data to make statistically significant judgements about mathematical knowledge and about software.

The relevance of our work to critical software infrastructure protection is most clear in the context of improving the reliability and security of hardware and software systems. We know that testing is important, but testing is mainly a way to detect bugs. It cannot insure the absense of errors. Knowing that chips and code behave as we intend depends on knowing more; it depends on discoveries of the 70's that algorithms and systems can be seen as mathematically precise abstract objects as well as physical objects.

The highest levels of assurance that we can provide about systems comes from proving the assertions that we make about them during the specification and design process. It comes from checking carefully what we think we know about them stated in declarative language and in abstract algorithmic language. We discuss this approach in more detail in the next section.



1.2 Applications to Verification Technology

Threats The Internet and the Web have created extreme opportunities (Web services) and extreme vulnerabilities — dependence on insecure and unreliable software infrastructure. The cost of errors and insecurity is at least $59.5 billion annually [161]. This situation has been known on a smaller scale since the mid 1970's when it was called the "software crisis". Popular books such as *Fatal Defect* [151] and other scholarly studies by McKenzie [115] have studied the problems and funding initiatives proposed to correct them.

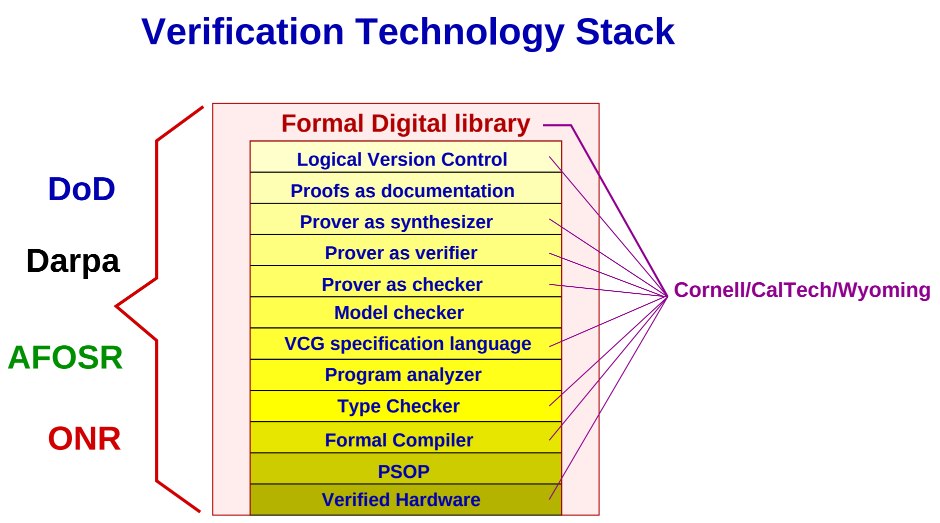
We are all familiar with computer viruses, they have caused serious disruption and inconvenience, but we have not yet seen a coordinated and serious viral attack on our software infrastructure. It is not difficult to imagine what a terrorist hacker cell could do to us if they took the time and gathered the information needed. Such an attack would not look like a simple virus, it might be more like the 1918 influenza pandemic in the software world. There might be a silent phase of 18 months when backup systems are being infected with no trace. Then an attack could be launched that would delete files and mangle data formats. It would deny service to many industries including the financial markets and military systems. As backup systems came on line, the virus would reemerge.

We could see damage on an unprecedented scale in government services, health care services, institutional functioning across the economy from universities to insurance companies to the power grid. Damage to disaster recovery software might be timed to coincide with a terrorist attack on people or on physical structures, thus greatly compounding the impact of an attack.

Protection and security Our goal is to use the opportunities (Internet and Web) to reduce the vulnerabilities. In particular, we are building a formal digital library (FDL) as a lens to focus information on the vulnerabilities. We are using information science and technology to provide an *information network* of facts about algorithms, data structures and systems. The FDL is a piece of that network accessible to people and to interactive theorem provers. These provers will draw computing power from *the grid* and information from the FDL and apply it to guarantee that the reasons we think a software system is reliable and secure are justified. The FDL will add a missing capability to a new technology for hardening and securing computer systems.

A verification technology is emerging as a result of 30 years of funding from the government (DoD, NASA, NIST, NSA, NSF). This technology can solve the problems of the 70's, but the target has been moving rapidly — from small programs to concurrent systems to distributed web services. Verification technology must be correspondingly scaled and advanced. Our FDL will help do that.

One way to understand the elements of verification technology is to focus on the so-called *stack* — the layered collection of hardware and software of which a complete system is composed. Parallel to this stack is an *emerging verification stack*. Its components can be integrated into the programming tools used at various levels of the stack to help make the resulting layers much more reliable. The DoD has invested in several components of the verification stack and has funded some of the research that has brought it into being.



At every level of the software stack, we see elements of verification technology being integrated — extended static type checkers, program analyzers, specification languages, model checkers, and interactive theorem provers. All of them will benefit from a semantic backbone to the stack provided by our FDL and Logical Framework for systems.

Our formal digital library (FDL) is an experimental new component starting at the top of this stack and providing a semantic backbone for the whole thing. It is critical to reliability in itself, and it also improves other elements of the verification stack. It also facilitates a programming methodology for large software systems. The FDL holds vital knowledge about a particular system, and it provides an interface to general knowledge needed to understand, modify it, and relate it to other systems. The FDL holds knowledge about the algorithms and protocols in the system, about what they do and how they connect. This knowledge is formally linked to critical components of the system, and it is linked to general knowledge about algorithms, data structures, and other systems.

The FDL will enhance the standard technology of trust in several ways. It will allow a software supported presentation of entire software systems as *reference systems*. We are demonstrating this for distributed systems. It will make proof technology more efficient as we illustrate later. Most critically, it will make this technology more flexible. It is a lack of flexibility that is one of the limitations in deploying these tools more generally and at a larger scale.

Just as lower levels of the stack provide networking links that connect a system to other systems, the FDL connects system knowledge to the network of knowledge that is essential to understand it and support it. It is also a basis for quantitative data about the system design.

The FDL is the interface between knowledge that is checked and generated by machines — *formal knowledge* — and the kind that can only be checked and generated by humans — some call it *intuitive knowledge*, or by contrast, informal knowledge when it is in the context of formal knowledge.

We describe the role of our tools and results in the chapter on Critical Infrastructure Protection (Chapter 3).

1.3 Research Issues

The main question we face is how we can assemble and coordinate very large digital information resources so that computing power can help us create formal symbolic knowledge and use it to advance science and technology broadly. In particular, we want to take advantage of a significant body of formal computational mathematics and use it to substantially improve the reliability and security of the nation's software infrastructure in a timely manner. How might this be done?

Here is a scenario for using our formal digital library and logical framework that suggests one way this will be done.

> A programmer must modify a protocol in the event processing protocol stack of a system, due to changes in the upstream processes. The formal documentation of this protocol in the FDL system/theory entry for the stack states assumptions about the input stream and invariants of the protocol. It is not clear to the programmer that the new message stream will satisfy the input assumptions.
>
> Further investigation of the upstream process, also in the FDL, reveals that all of the necessary assumptions are satisfied, except for some assumption P. The programmer asks to reexecute the proof that the invariant holds under the new assumption. The tactic is in the FDL and it is executed, revealing the need for an additional fact about streams of messages of a certain type T.

> The programmer proposes a simple modification to the code that appears to establish the required additional fact, but there is some uncertainty. The programmer searches the FDL for theorems about streams and finds a fact that would justify a slightly different code modification, but the proof uses a different logic than the one the tactic is written for. The programmer asks for the fact to be translated into the tactic's logic and then runs the prover on this fact, which is established automatically. The programmer is able to make the slightly different modification to the code and be much more certain that the change is provably correct even though he or she does not understand formal logic in depth.

To support such a scenario, we need a way to connect formal knowledge about a system to the algorithms, protocols, datastructures, classes and interfaces that make up the executable procedural knowledge. This procedural knowledge is vast, stacked layer upon layer, spanning many levels of abstraction. The corresponding formal knowledge base would be even more vast — two to five times larger — and it is connected to a large amount of a basic knowledge about general mathematics and computing (numbers, strings, arrays, lists, trees, graphics, stacks, queues, events, automata, etc.).

No one person or one development team or one set of programming tools will comprehend all this knowledge. It must be connected, even though it is not homogeneous. For example, formal definitions of an intuitive idea might not be equivalent in different logics; specifications of a computing task are not equivalent. Not even the formal general mathematical knowledge is homogeneous in this sense.

Fortunately, not all elements of a software system are equally critical, and not all knowledge supporting a system needs to be checkable by computer, i.e., formal. We need to know *what knowledge is critical and thus requires a dependency analysis — what knowledge depends on what other knowledge.*

Also fortunately, well-educated systems analysts and programmers think clearly and precisely about the elements and structure of software systems. We know how to design well and explain the design. We know how to solve computational problems, explain our solutions and use them to guide the generation of code. We know how to share ideas and share high level algorithms or protocols. How can we share extremely precise ideas, and how do we share formal ones?

One class of reliability problems arises because we don't save our thoughts, solutions, ideas, and designs in digital form. We don't connect them to the code, and we can't easily bring ourselves to systematically scrutinize these ideas for small errors. A related problem is, we make lots of little mistakes — at every level. Indeed, in formal methods projects run by NASA, every system scrutinized by formal methods revealed previously unknown errors [37]. Another related problem is that the programmers who had the original ideas are not the ones who fix and extend the code. These new people might introduce incompatible ideas and solutions, and this incongruity is not discovered until later.

On a small scale, all of these problems can be solved with the current verification technology (for functional programs). Moreover, we see ways to scale up this technology from functional algorithms to systems. This requires scaling up the amount of formal knowledge available to both the automatic tools and to people, and keeping it related to the informal ideas that guide the system building. We are attacking that problem.

To create more formal knowledge, it will be possible to share data among verification systems, not only definitions and theorems, but proofs that can be replayed in a new context and repaired automatically by local tactics. We have already done this on a small scale [30]. How can we extend these methods? Such sharing will significantly lower the cost of creating formal knowledge, in part by eliminating massive duplication of effort. It will also be easier to derive the formal knowledge from intuitive knowledge.

To keep the formal declarative knowledge connected to the system, the executable procedural knowledge, there must be automatic linkages so that when one element changes, the other does as well. Our project does not address all these problems, though we have thought about many of them and have done original work on several. We focus on the means of collecting, connecting, and significantly expanding formal knowledge.

In our proposal, we described the following categories of work. We proposed to look at library infrastructure issues — how to "automate a logically sound information management service" for content from diverse theorem provers, in particular how to allow sound sharing of content. We proposed to explore this topic experimentally and theoretically.

The second category of our work is content creation, especially content pertinent to software infrastructure protection. We proposed to create it in ways that would illustrate the key uses; so we wanted reference algorithms and datastructures. We also wanted formal classes and promising approach to reference systems, especially a distributed system. In this category we would also experiment and develop theory. Thus, our work is characterized by this matrix:

	Library Content	**Library Infrastructure**
Foundational Theory	(1) CF	(3) IF
Experiment	(2) CE	(4) IE

Here are some of the issues in each of the four categories.

Library Content

For those investigations, especially in the first three years, we proposed looking at three representative logics and systems producing formal content. In the United States and in the European Community (EC), there are two kinds of type theory used: Church's classical *Simple Type Theory* (STT) [41, 69], and two versions of constructive type theory. To keep things simple, we mention only *Computational Type Theory* (CTT) [118, 119, 48], and type theory based on Impredicative Inductive Constructions (IIC) [66, 52, 53]. For each of these theories, there are major interactive theorem provers in both the US and EC. In the EC, the STT prover is HOL, and the constructive prover for IIC is Coq. In the US, the classical type theory prover is PVS, and the constructive prover for CTT is Nuprl. Another major constructive prover in the US is ACL2 [94, 95].

In addition there are the logical framework systems, Isabelle, Twelf, and MetaPRL, which are all based on a *constructive metalogic;* these support a variety of logics, such as type theories and set theories, although it is the type theories that are most developed, even in the logical frameworks. Isabelle supports STT, CTT, and ZFC set theory, and MetaPRL supports CTT and CZF set theory.

For us it is natural to pick the US sytems in each category, Nuprl and PVS, and to pick the most modern and versatile logical framework, the only one that can do *inter-theory mappings* — MetaPRL. By using the US-based systems of each type, we can more directly support DoD verification efforts, and we can be sure that US systems are included in any EC-based effort similar to ours.

Our principal new content production was in reference algorithms and protocols, with a significant part devoted to distributed systems along very similar lines to NRL work. Most of the new content we collected was created in other, separately-funded projects (by DARPA, NSF, and AFOSR) concerned with reliable adaptive distributed systems and with security protocols.

We look briefly at our *technical approach* to each area. Much more detail is provided in the chapters we reference below.

(1) Theory supporting multi-logic content

The key issues here are how to relate the theories of different content providers. What is the logical basis for relating and combining results from different theories? Part of our theoretical work went into understanding key concepts supporting classes, objects, reflection, and distributed system formalization [100, 45, 17, 18]. What we learned is that type theory is especially good at expressing the structural concepts that lie behind modern programming methods, in particular elements of object-oriented programming. It is also good at expressing system composition operators and at expressing the key ideas in aspect oriented programming. In addition, we discovered that several key concepts from modern computational mathematics could not be expressed in classical set theory. How could set theory handle these concepts? Would constructive set theory bridge the gap?

We made a number of discoveries that justify translating results from the simple type theories used in theorem provers, such as HOL and PVS, into the constructive type theories used in Alf, Coq, MetaPRL and Nuprl. One of these results, Moran's Theorem, opens a new area of theoretical investigation in logic that will be of significance in support of deep connections between set theories and type theories [133, 45]. He shows that the Howe map from types into sets when composed with the Aczel map from sets into types produces an isomorphism, and this allows many results about sets, including independence results and large cardinal axioms to be proved in type theory, where they have a totally different meaning.

What systems and formalisms are most important for formalizing distributed computing and for expressing security specifications? There is good evidence that IO automata are an appropriate formalism [178, 68, 113, 84, 175, 4] can they be made more useful if formalized in a very rich type theory?

What kind of certificate can we supply for hybrid proofs in the sense of the same logic but different systems, e.g., simple type theory in HOL and in Isabelle HOL? What kind of certificate is needed if the logics are different, as in HOL and PVS versions of simple type theory?

Is it possible to translate results between all the major theories by using set theory or classical computational type theory with union and intersection operations as the semantic base?

(2) Experiments with content

We explored various methods of importing PVS into our FDL. We examined appropriate term structure for PVS. We explored data exchange formats for OMDoc. We created API's among the provers that are now connected to the FDL, and have designed more advanced versions. We also have explored further automation of the importation process.

We need to understand further the elements of an API for PVS that would be completely automatic; is this possible? Is it possible to use the FDL binding mechanism for certain PVS contexts? What advantage is there is doing this?

How can we connect formal and intuitive knowledge so that the connection remains unbroken under changes to the formal theory? Can the formalism help us locate the key ideas in a proof or in an algorithm or in a system? Can the formalism help identify with more precision the places where errors are most likely?

What is the right mix of formal and intuitive knowledge for maximum human readability? This is a typical question for human computer interaction studies, part of information science. The question is even more pressing when we look at system documentation. For example, certain well-known programming methods and various very common mathematical facts are best presented informally, while other steps should be formal. Part of the trade-off will be a function of cost and risk, another information science issue.

What is the size of a completely formal account of a difficult algorithm or protocol? Is formal theory size a good indicator of conceptual difficulty, of work involved in documenting, or of the likelihood of error? Can we answer such questions for a full system as well? How many hard algorithms are there in a typical system? In the systems we are formalizing?

Since we are assembling a reference distributed system, we are very keen to collect formal material related to it and to draw that community into reading our presentation. We are finding that this is not a simple task; some results are restricted, some are proprietary, some projects are too busy to cooperate, others see no value in cooperating and others we are too busy to contact. It is turning out to be an interesting challenge to collect the material we want from other groups.

Library Infrastructure

For these investigations, we needed to create a prototype formal digital library. Our proposal detailed our experience in thinking about these matters and our planned approach to building the prototype library. We followed that approach, including studying the library mechanisms of other provers, reading about digital libraries, and writing extensive design notes. The theory and experiment were intertwined. This activity occupied a large part of the first year.

Our prototype system involves 95,000 lines of code, and we continue to revise and expand it. Some of the most basic functions were obtained by modifying library code from Nuprl, however Nuprl and MetaPRL are clients of the FDL as many provers will be. (Note that at most 18% of our funding was budgeted for system construction, so our productivity here is very high indeed.) We have plans to code a second version, depending on the pressure of other matters; it would be a minimal digital library built in the most generic way we can imagine. The existing prototype was built to allow experiments as soon as possible.

(3) Foundations of a formal digital library

We had to explore data formats, reference structures, hypertext support, name space management, database transactions, accounting mechanisms. Our May 2002 review presented many of these basic ideas, and we are now publishing results.

What is the best basis for an FDL; is it a directory structure or a database? Many theorem provers use a theory directory structure. Automath pioneered a “tree of knowledge.” But a viable alternative seems to be a flat basis on which structure is overlaid, the flat structure based on a persistent object store. Which approach supports the operations that are most critical to an FDL?

We also went through many scenarios for how users would interact with the FDL, and what communities would be served by it. The FDL will become a component of a larger global mathematics resource, assembled with help from the EC, Japan, and North and South America. Our first ties will be with the EC. In this role the FDL will enhance the dissemination of knowledge and learning, and it will become the basis for an international journal of formalized mathematics. It will support education in computer science and in the ties between mathematics and the science of programming. In this way it is already being used by Helm, the Web, and soon the NSDL.

One of the major concerns of people thinking about this area is the proper format for proofs. We are conducting experiments on this topic, based on our considerable experience with different proof formats over the years. We made up proof terms for PVS that match the format we have adopted for the library; these are the only proof terms for PVS that we know about, and we have projected them on the Web.

We may need to explore variants of Natural Deduction style proofs if the HOL and Mizar formats, called Isar, prove compelling. We have experience with these from the past, and rejected them in favor of refinement style proofs, closely related to analytic tableau. We have the background needed to explore this topic to any depth and in any direction, including natural language translation of proofs [87].

These foundational ideas are directly implemented in the experimental FDL. They support a variety of services, such as logical dependency tracking, renaming, pruning, joining, Web display, dynamic formatting, content indexing, clustering, searching, annotation, semantic anchoring of text, and inter-theory translation.

The FDL will also become a resource for system verification, as we are now demonstrating. We are actively looking for other partners in this aspect of the enterprise.

(4) Experiments with library services

Ideally all the library services would be available via a Web browser, but the content providers do not yet have this capability, and we must decide which services should be made available on the Web first? Can we display logical dependency metadata on the Web?

One of the most critical and difficult services is a Web presentation of FDL content, including formal metadata. This is the first service needed to start creating Web based FDL services. We have exceptional results in this area, and they will support a whole variety of use of the FDL for knowledge dissemination and learning. It provides the basis for working with the National Science Digital Library (NSDL) [6].

During work on distributed system verification, we discovered that the persistent object store and complete dependency tracking opened the possibility of systematically modifying a significant portion of an implemented theory [28]. The key idea is that the modifications were done in a series of passes over the theory with tactics designed to automatically repair failed proofs. Can this technique be applied to larger theories? Can we make it a more generic service of the library?

We have explored other experimental services that are very advanced, including the translations between theories and the cooperative use of theorem provers to develop distributed system protocols [30]. How would a translation service be used? How can it be checked? What certificates apply given the fact that these services maybe rapidly evolving?

We have joined the National Science Digital Library project as an effort to establish a natural community for certain aspects of this work, those parts of CIP that involve education and the role of the FDL in scholarly publication. See the next section for a further discussion.

We summarize these results according to their contributions to the goals of the BAA in Chapter 2.

1.4 How People Will Use the Formal Digital Library

The previous section considers briefly the issue of how people will use the FDL. Since this is a specific question we have been asked a number of times, we pull together an overview of the ways in this section. Note that in Chapter 4 on Design Notes we have an extensive list of scenarios.

It is worth noting that other people are already using some elements of the experimental FDL, for example Helm and OMDoc. The verification effort at Cornell is currently experimenting with FDL services.

There are four major categories of use which we consider in the following subsections.

1.4.1 Knowledge dissemination and learning

We imagine that readers who are interested in how to create verified algorithms or formally documented code would turn to the FDL for examples. We have experience with this role from Nuprl, and the FDL contains over a hundred examples of various kinds.

We also imagine that readers will want to understand an entire reference system with its formal reference algorithms, formal classes, and formal documentation. We are assembling such an artifact, and it might be unique. It will be possible to examine questions such as: How many critical algorithms are there in a particular system of size n? For many systems we know, there are a surprisingly few interesting algorithms compared to the many lines coding basic tasks and basic datastructures.

These examples support "excellence in program construction" along the lines imagined in the proposal and in many computer science department curricula, especially at the universities represented in this project. Thus there is strong synergy with the educational mission in computer science.

Our particular approach to this topic illustrates both standard programming practice and the deeper ties with mathematics that are possible when code is synthesized from constructive proofs. Thus, we encourage the "deeper connections between mathematics and the science of programming."

The FDL will also have a role in the National Science Digital Library based on a newly funded NSF project.

1.4.2 Scholarly and scientific communication

The Cornell e-Print arXiv in physics and mathematics [65] is an example of how a digital library can transform the practice of scholarly publication in areas of physics. We think that the FDL might serve to enable a new area of scholarly publication, in an emerging field called *formalized mathematics.*

We are supporting the efforts of Michael Kohlhase, leader of the OMDoc group, in his proposal to the editors of the *Journal of Formalized Mathematics* that it be opened beyond the Mizar logic.

1.4.3 The theory and practice of system verification

This is the main application area for us, and we discuss it extensively in Chapter 3. We are actually now using the FDL in our work with DARPA and AFOSR on protocol verification. The DARPA work involves adaptive protocols and the AFOSR work involves security protocols. We have used the FDL to share mathematics between MetaPRL and Nuprl, and we are providing a translation service for PVS theorems that we expect will be useful in our verification work. Conversely, we are adding content to the FDL based on the mathematics created for these verifications. This content can be seen in the FDL content section of the project Web page and in the report by Constable and Bickford [29].

In addition, our work is supporting activity in two other MURI CIP projects, SPYCE and Language Based Security. In the case of SPYCE, we are working with Joe Halpern and graduate student Sabina Petride on knowledge based protocols. We have formulated two such protocols in our Logic of Events, and we have created a knowledge based protocol based on the concept of *formal algorithmic knowledge* [61].

We have also worked with a student of Greg Morrisett, Mat Fluet who works on language based security and knows how to use interactive theorem provers. We have used the very dependent type constructor from Jason Hickey's work to express properties of programs as types.

Our approach can be characterized as information-intensive infrastructure protection. The FDL will provide information based services in system design, implementation, verification, extension, documentation and maintainance.

1.4.4 Data for a new quantitative logic in the large

The FDL opens a new class of logical questions, about collections of theories and the quantitative consequences of various organizations or formulations. For example, once we include theories from Coq, we can ask about the number of essential lemmas and definitions need to prove the Fundamental Theorem of Algebra or the Fundamental Theorem of Arithmetic. It is possible to compare answers to the second question among a number of logics and provers, and compare the approach of proving the theorem generally for unique factorization domains (see the Nuprl algebra libraries under [FDL Content]).

In the previous section we cited another example of this kind, a comparison of the total size of two leader election algorithms. When we gain access to the PVS proofs of these algorithms, we will be able to make several interesting quantitative comparisons. We can include the time required to replay the entire verification from scratch as another measure of complexity. In this case we will see the advantage of being able to share proof tasks among provers since we have done an experiment using Nuprl alone versus in combination with MetaPRL and JProver [30].

1.5 Work Process

To give some idea of the work we are doing and the process we use to accomplish it, *we include the statement of work (SoW) from our proposal* in the first section and our processes in the second.

The work schedule that we proposed has not been subsequently amended except for encouraging us to find a community of potential users. We discussed the user community in the previous section and will say more in Chapter 2.

1.5.1 Statement of work

Year 1: We will improve and implement those library services that allow us to cooperate in building theories among ourselves at Cornell (office to office and system to system — Nuprl and MetaPRL). This will involve detailed work on sentinels, stable tactic code, archival operations and local web editing. We will also bring MetaPRL to a more advanced stage and expand our ability to interact with it from the Nuprl logical library. We will continue to develop the basic library of functions and data types and illustrate the inter-linking of text, hyper-text, and formal content; and we will continue to apply our LPE to the Ensemble protocol design and verification.

Year 2: We will extend our cooperation to the three teams working remotely. One goal is to be able to merge libraries developed independently at the three participating institutions. We will use both the logical library and multiple refiners remotely. We aim to extend web-based editing among the group. At this stage we will call the library a Common Logical Library. We will collectively demonstrate an impact of the Library on one of our major applications, such as Ensemble verification as well as on the basic library of functions and data types.

Year 3: We will begin to import theories from other remote sites and export theory building and editing operations. We will bring all of the capability to bear on the library of functions and data types and applications. We will also organize a wider experiment with the Common Logical Library by arranging to support other external efforts as in HOL, PVS, or ACL2.

Year 4: We will test the tools on a wider group of remote users and bring more capability on-line to support the emerging needs. We will also automate the process of creating links to text and start planning to connect the Common Logical Library to an active digital library effort at Cornell such as NCSTRL.

Year 5: We will link our tools and results to one of the digital libraries such as NCSTRL, supported at Cornell. The Library will also support the code base for the subset of ML used in our verification work along with components of application software such as Ensemble.

1.5.2 Work process

Each of the three sites, Cornell, Caltech, and Wyoming, has a specific focus, and in addition we work together on certain problems as is demonstrable from the joint papers, visits, and collaborative software development. There is close coordination between the groups through joint meetings, attendance at conferences, and joint work on papers and software.

Proposed task allocation Here is a summary of the task allocation from the proposal. Cornell University will have the leading role in the proposed activity. It will provide the core logical library implementation, the logical accounting and library operations. It will also supply formal content in the area of applications to distributed computing and in the use of reflection. It will apply and test the new concepts in significant on-going applications to building reliable software (such as the Ensemble protocol design effort, DARPA PCES applications, and AFOSR protocol verification). It will continue to provide a type theoretic semantics for the ML programming language used in verification work. It will also provide Web support for libraries and for connectivity with the libraries of other provers. It will provide a connection to the NSDL project.

Cal Tech will collaborate with Cornell on continued development of MetaPRL, especially using it to explore inter-theory cooperation and concurrent proof refinement. Together they will design and build an abstract interface between MetaPRL and the logical library. Cal Tech will also provide content in the form of formal programming language semantics linked to formally developed compiler code for ML. They will also explore the use of a simple core MetaPRL subset of ML to be used as the primitive proof checker for theories in the library. Cal Tech will also explore applications of formal classes in content production for areas such as constructive algebra.

The University of Wyoming will provide content in integrating algorithms extracted from proofs into systems. They will also explore the use of the ACL2 prover, for example as a proof checker for primitive Nuprl proofs.

Graduate students At each site, a significant part of our work is supervision of graduate students and engagement of the students appropriately in the overall work. We have attracted excellent students to the project, including one of the first information science PhD students at Cornell. It is not our practice to ask students to write code unless such a task is in direct support of their studies. There are three students who are adding formal mathematics content as part of their thesis work. One of them, A. Kopylov has worked directly on library services as well.

Theory and design The entire project has participated in design discussions for the FDL, and at Cornell, several seminars have been devoted to this effort. Two of the graduate students have been critical in this work as well. We also benefit from interaction with our colleagues at each site who find the project of interest.

System building At most 18% of our budget is for coding the formal digital library prototypes, and the three professors are unable to spend time in this activity. We have been remarkably productive in this effort because the programmers are also researchers who are involved in the design and use of the FDL. We have begun to use elements of the FDL to activities funded by other sources.

Community building Our hope and expectation is that in due course the FDL will become that natural repository for the results of all the major theorem provers. We hope to collect substantial segments of the dark matter we referred to earlier. Our experience is that attracting users and followers takes time.

It is first necessary to build a system that is very easy to use and beneficial to certain communities. In the case of the Nuprl system, it took several years after we had built a polished version before we attracted users and followers.

The system was finished in 1986, but only version 4 was widely used, in the 90's, and 1996 saw the largest number of citations.

1.6 Accomplishments and Productivity Measures

1.6.1 Major accomplishments

In summary, our major accomplishments are these:

1. We developed and wrote a detailed rationale and design for a prototype FDL.
2. We built the prototype system, a large undertaking.
3. We established procedures for acquiring content from Nuprl, MetaPRL, and PVS, and assembled extensive sample content. This was a major enabling effort which is groundbreaking in a number of ways.
4. We created sophisticated Web-publication services in the FDL that include a Dynamic Math Formatting mechanism, consequently Google finds our content easily, e.g. our PVS libraries are found as the first results returned on querries such as "PVS graphs" and "PVS number theory."
5. We developed an interface with OMDoc and exported FDL content to Helm.
6. We created novel content in MetaPRL that supports an object-oriented way of presenting reference algorithms and applied it to red/black trees.
7. Cornell and Caltech collaborated on the development of constructive algebra in Computational Type Theory (CTT), and a formalization in CSF was explored.
8. We designed mechanisms for adding dynamic hybrid formal/intuitive articles to the FDL that will form a basis for explaining proofs, algorithms, and systems. It is a basis for dynamic formal documentation. These mechanisms will be perfected and deployed in the NSF's National Science Digital Library.
9. We imported content that supports distributed algorithms and protocols of particular interest to CIP.
10. We made a significant theoretical discovery that relates the expressive

power of type theory and set theory, the two major foundational theories for expressing formalized mathematics.

11. We explored using ACL2 in checking Nuprl primitive proofs.
12. We made a significant advance toward adding a reflection service to the FDL based on its development in Nuprl.
13. We have built a prototype peer-to-peer server for connecting distributed FDL fragments.

1.6.2 Sample results, discoveries, and insights

Content – Theory What are the fundamental computational mathematics concepts? We have shown that polymorphism allows an elegant definition of records, classes, dependent records, formal classes, and inheritance. Set theory does not allow function polymorophism, but based on the work of Howe [91, 90], we know how to add it to set theory, and we discovered how to add it to the simple type theory of PVS [133]. Generally, we have made a strong case that type theory is the appropriate language for describing modern system structure, and it is well suited to describing mechanisms that support the coherent evolution of systems.

Content – Experiment We have combined two provers in the process of synthesizing a leader election protocol [30]. We are also able to compare the size of all results needed for simple leader election compared to the TIP leader election protocol studied at the NRL [13, 15, 14].

Infrastructure – Theory Which is the best organization basis for an FDL, a database or a directory structure? We have examined the mechanisms for combining theory development and made a case for a database with abstract object identifiers. It allows capabilities that a directory system does not.

Infrastructure – Experiment Is it possible to provide a Web based display of logical dependency metadata? We have created the tools that allow this.

We have also studied in depth a formalism for distributed computing that is widely shared in the verification community, namely IO automata and their variants. We have used this in other projects and are working toward supporting it well in the FDL. It is well suited to studying adaptive systems and system evolution.

1.6.3 Productivity measures

Publications Here is a list of the 23 publications (including two theses).

Stuart Allen. Abstract identifiers and textual reference. Technical Report TR2002-1885, Cornell University, Ithaca, New York, 2002.

Stuart Allen. Notes on the design and purpose of the FDL. `http://www.nuprl.org/FDLProject/FDLnotes/`, Ongoing, beginning 2002.

Stuart Allen, Mark Bickford, Robert Constable, et al. FDL: A prototype formal digital library. PostScript document on website, May 2002. `http://www.nuprl.org/html/FDLProject/02cucs-fdl.html`.

Stuart Allen, James Caldwell, and Robert Constable. Interactive Digital Libraries of Formalized Algorithmic Knowledge. MKM Workshop, 2001.

Brian Aydemir, Adam Granicz, and Jason Hickey. Formal design environments. In Carreño et al., editors, *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*, pages 12–22, National Aeronautics and Space Administration, 2002.

Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in Nuprl. In Carreño et al., editors, *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*, pages 23–32, National Aeronautics and Space Administration, 2002.

Eli Barzilay, Stuart Allen, and Robert Constable. Practical reflection in Nuprl. In Phokion Kolaitis, editor, *18th Annual IEEE Symposium on Logic in Computer Science, June 22–25, Ottawa, Canada*, 2003.

Mark Bickford and Robert L. Constable. A logic of events. Tech Report TR2003-1893, Cornell University, 2003.

Y. Bryukhov, et al. Implementing and automating basic number theory in MetaPRL Proof Assistant. Accepted to TPHOLs 2003 "Track B."

J. Caldwell and J. Cowles. Representing Nuprl proof objects in ACL2: toward a proof checker for Nuprl. ACL2 Workshop, 2002. In D. Borrione, M. Kaufmann, and J. Moore, editors, *Proceedings of Third International Workshop on the ACL2 Theorem Prover and its Applications*. TIMA Laboratory, 2002.

Robert L. Constable. Naïve computational type theory. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability, Proceedings of International Summer School Marktoberdorf, July 24 to August 5, 2001*, volume 62 of *NATO Science Series III*, pages 213–260, Amsterdam, 2002. Kluwer Academic Publishers.

Robert L. Constable. Information-intensive proof technology; lecture notes for the marktoberdorf nato summer school. Cornell University, Ithaca, NY, 2003. `nuprl.org/documents/Constable/marktoberdorf03.html`.

Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. In Wilfried Sieg, Richard Sommer, and Carolyn Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, Lecture Notes in Logic, pages 166–183. Association for Symbolic Logic, 2001.

J. Hickey. Introduction to the Objective Caml Programming Language. California Institute of Technology, 2003.

Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

Jason Hickey, Aleksey Nogin, Adam Granicz, and Brian Aydemir. Formal compiler implementation in a logical framework. In *MERλIN, Second ACM SIGPLAN Workshop on MEchanized Reasoning about Languages with varIable biNding*, 2003.

Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science*, pages 86–95, 2003. To appear.

Christoph Kreitz. The FDL navigator: Browsing and manipulating formal content. Cornell University, Ithaca, NY, 2003. `http://www.nuprl.org/documents/Kreitz/03fdl-navigator.html`.

Aleksey Nogin. *Theory and Implementation of an Efficient Tactic-Based Logical Framework*. PhD thesis, Cornell University, Ithaca, NY, August 2002.

Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver: Integrating connection-based theorem proving into interactive proof

assistants. In *International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 421–426. Springer-Verlag, 2001.

Tjark Weber and James Caldwell. Constructively characterizing fold and unfold. In *13th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003), held August 25-27 in Uppsala, Sweden*, 2003.

Xin Yu. Formalizing abstract algebra in constructive set theory. Master's thesis, California Institute of Technology, 2002.

Xin Yu and Jason J. Hickey. Formalizing abstract algebra in constructive set theory. Technical Report caltechCSTR2003.004, California Institute of Technology, Caltech, CA 91125, June 2003.

Theses One PhD thesis was finished under this project and there are four others in progress. In addition there was a Masters thesis.

Eli Barzilay. Practical Reflection in Type Theory, PhD Thesis, Cornell University, 2004.

Alexei Kopylov. Implementing Records and Objects in Type Theory, PhD Thesis, Cornell University, 2004.

Evan Moran. Adding Intersection Types to Howe's Model of Type Theory, PhD Thesis, Cornell University, 2004.

Aleksey Nogin. Theory and Implementation of an Efficient Tactic-Based Logical Framework, PhD Thesis, Cornell University, 2002.

Xin Yu. Formalizing Abstract Algebra in Constructive Set Theory, Master's Thesis, California Institute of Technology, 2002.

Work in progress

Mark Bickford and Alexei Kopylov. Verification of protocols by combining provers using the FDL, September 2003.

Mark Bickford. Experiments with theory modification in the FDL, September, 2003.

Sabina Petride. Knowledge-based specifications in the logic of events, draft paper and PRL seminar notes (with Mark Bickford, Robert Constable, and Joe Halpern), 2003.

Software

The Formal Digital Library (FDL) core (95,000 lines of code)
FDL internal service code (20,000 lines of code)
Dynamic Pure Structure (DPS) editor extensions for FDL
Web Posting code (4627 lines of code)
Dynamic Mathematics Formatting Program (6,000 functions in DPS editor)
FDL-Embedded-Latex
MetaPRL enhancements for system development

1.7 Future Work

1.7.1 New capabilities

We have explored the value of a new kind of object for the FDL, the hybrid formal/intuitive document, *hyfi document.* It is based on the observation that it is easier to share definitions, theorems, and reference algorithms than it is to share proofs. We created the kind of document that will allow people to more easily include formal definitions, theorems, and algorithms in their articles. We intend to build substantially more library services for hyfi documents.

We have produced experimental translations services that we intend to exploit and make part of an advanced service suite.

We have explored a technique for theory modification that works in multiple passes. We will see whether this can be made a service as well.

We have begun a more systematic study of clustering methods based on the hyperlink reference structure and on citations [96, 97, 98].

1.7.2 Vision

Our work on the FDL and the uncovering of its capabilities convinces us that there will eventually be a federated digital library of formal and informal algorithmic knowledge from the US, EC, and Japan nucleated by the FDL and related efforts to come. This *information network* will be a partner to the *computing grid.* A dozen or more powerful interactive theorem provers will be connected to the grid and the network, and several of those will include decision procedures and fully automated provers such as JProver, Otter, and EQP. Provers will have contributed over 100,000 formal theorems and proofs that support many detailed models of hardware, virtual machines like the JVM, and semantic accounts of programming languages. The provers will draw computing power from the grid and knowledge from the FDL and thus extraordinary verification power will be available as needed to protect the critical software infrastructure.

The FDL and the grid will be used to support several critical infrastructure software systems. Their core functionality will be hardened by formal verification and checking, and documentation will be an interactive FDL-supported mixture of formal and intuitive knowledge.

When extensions are made or when new vulnerabilities in the unhardened parts are detected, we will be able to coordinate via the formal knowledge network and the grid a dozen provers and to rapidly harden a system section in a few months, rather than the few years it would take now. Information-intensive and computer intensive verification technology will provide an advanced response capability that is not practical now.

1.7.3 Future plans

Chapter 8 provides a more detailed account of our goals for the next three years and our future plans. Here we summarize them briefly.

Goals: We want to attract more content providers so that by the end of the five year effort, several groups are submitting content including PVS users in distributed protocol verification, HOL users in program extraction, and so forth. This will be a challenge since there are political and institional barriers to be overcome. We will need to create the position of a collector as we proposed.

We intend that we will use the FDL directly in additional CIP activites, and we expect that at least one other MURI will use the FDL with us. This is likely given the current state of activity and our relations with two other MURI efforts.

We want to be a major player in the European Community effort in this area, which will surely include OMDoc and Helm. Our goal is to have at the end of five years a federated mathematics library of which our FDL is a part.

We want to attract authors of mathematics and computer science articles. We will attempt to use the FDL in the context of the FDL to do this.

We want to attract dataminers and machine learning experts to search the FDL for interesting patterns.

Activities: We intend to automate more FDL services. There will be more comprehensive API's to access them. These will include advanced services such as translations between theories, and we will provide additonal clustering and search methods.

We will use the FDL increasingly in verification work directly relevant to CIP/SW, including working with other MURI projects, other DoD research activities and possibly with the Naval Research Laboratory in particular.

We intend to move more FDL services to the Web, for example, more dynamic pure structure editing on the Web using the DPS editor that we are now using internally. We intended to provide text editing facilities that smoothly access our formal content and thus encourage authors to draw on the FDL content.

We will create and collect more sample content, including material that the corporation ATC-NY will pay to have us collect in the FDL.

We have also laid the ground work for building an experimental reference distributed system on which we can bring together results from HOL, PVS, Isabelle, and Nuprl on specific protocols. We will be adding considerable content in this category based on our verification work.

We will remain active members of the Mathematical Knowledge Management (MKM) community and its North American branch as well, and remain active participants in TpHOLs.

We need to decide on whether it is worthwhile to reimplement the FDL in a very simple generic version, however, we will continue to experiment with a distributed implementation.

Justification for Optional Funding

Here are seven strong reasons why we should be given the optional two years of funding:

1. Our proposal and results squarely meet the BAA requirements.
2. Our work contributes a missing piece of verification technology important to CIP/SW and to Navy missions (e.g. NRL work).
3. Our work is already being used to help in DoD missions, including Navy and Air Force missions, for example: AFOSR work in IAI, and support for NRL work on IO automata in PVS.
4. We are demonstrably very productive in all categories mentioned in the BAA.

5. We have been effective in opening a new fundable research area that no other agency was funding before our grant.
6. Our continued involvement will speed progress toward the DoD goals articulated in the BAA.
7. We are highly qualified for this work, and our progress and results are excellent, as will become more clear as time goes on; for now we can point to good conferences, invited lectures, and followers.

1.8 Outline of the Book

In Chapter 2, we summarize our results as progress toward the goals of the BAA. We itemize results in each of the ten research concentration areas. We have results in all of them. This elaborates Section 1.6 above on accomplishments.

In Chapter 3 we discuss in more detail the role of this work in critical infrastructure protection. This is a substantial elaboration of Section 1.2 of this chapter.

In Chapter 4 we present design notes for the FDL. This elaborates the discussion above in Sections 1.3 and 1.4.

In Chapter 5 we present highlights of the FDL manual. This chapter also provides depth to our account in Sections 1.3 and 1.4 above.

In Chapter 6 we present a part of the MetaPRL system library to illustrate the role of a logical framework in conjunction with the the FDL in critical infrastructure protection and thus elaborates Section 1.2 above.

In Chapter 7 we present sample content. This static representation only hints at the capabilities of the FDL and those of its Web Projection. This is an illustration of the points we made in Section 1.2 about our reply to the BAA.

In Chapter 8 we describe the new capabilities provided by the FDL for the task of system verification, for the creating of reference systems, and for translating among theories stored in the FDL. We also discuss the next steps we plan to take in furthering our research in all four quadrants of the chart used above to organize presentations of our work. This elaborates Section 1.7 above.

In the short conclusion in Chapter 9, we summarize our case for extending the project into the final two years. This expands Section 1.7.

Volume 2 contains the following sample papers of the project:

Dependent intersection: A new way of defining records in type theory, by Alexei Kopylov, 2000.

Formalizing abstract algebra in type theory with dependent records, by Xin Yu, Aleksey Nogin, Alexei Kopylov, and Jason Hickey, 2003.

Information-intensive proof technology; lecture notes for the Marktoberdorf NATO summer school, by Robert L. Constable, 2003.

MetaPRL — A modular logical environment, by Jason Hickey, Aleksey Nogin, Robert L. Constable, et al, 2003.

Representing Nuprl proof objects in ACL2: toward a proof checker for Nuprl, by James L. Caldwell and John Cowles, 2002.

Chapter 2

The Multidisciplinary University Research Program in Critical Infrastructure Protection and High Confidence, Adaptable Software

In this chapter we present the Broad Area Announcement (BAA) for this research program verbatim, and then we organize it according to a natural work flow. We summarize our accomplishments in each of the ten research concentration areas presented in the BAA. We demonstrate that we have made significant progress on all of them.

2.1 Digital Libraries for Constructive Mathematical Knowledge

The Topic Area eight BAA follows verbatim, italics ours.

Topic Area #8: Digital Libraries for Constructive Mathematical Knowledge

Background: Software itself is fundamentally about algorithms. *And,* *knowledge about algorithms is derived from discoveries in mathematical sciences that can be expressed constructively.* The research community accepts discoveries as fact only after a lengthy process of review, validation, and acceptance, which has been remarkably effective. Consequentially, software assurance depends upon this understanding, trust, and experience in constructive mathematics and its expression. Systematically developing an infrastructure for this knowledge of algorithms — as in a digital library — would contribute to higher quality software, greater confidence in program construction, faster dissemination of knowledge globally, and deeper connections between mathematics and a science of programming. This development begins by codifying constructive mathematical knowledge using a view of computation as deduction, and by engaging the research community in the pursuit of this national infrastructure.

Objective: To create a digital library of algorithms and constructive mathematics usable for program and software construction.

Research Concentration Areas: (1) Develop proof-checking and model checking for certifying proofs of the standard body of computationally related mathematics. (2) Catalog those principal mathematical concepts, together with their formal definitions, which are used in contemporary computing. (3) Investigate suitable base language and logic within which competing logics can be expressed and evaluated. (4) Provide automated assistance for routine aspects of developing libraries of formal theorems, proofs, algorithms, and their expressions as programs. (5) Investigate forms of assured interoperation for assembling, composing, specializing, and generalizing algorithmic knowledge. (6) Investigate reflection for coordination, interoperation, and dynamic adaptation. (7) Study issues of consistency and maintenance among libraries. (8) Address the human-computing aspects of syntaxes and concepts that are appropriate for both foundational developers and end-users. (9) Explore innovative metaphors and protocols for understanding, using, composing, searching, authenticating, and validating constructive results. (10) Examine models of applicable assurance structures and their economics. This research draws upon many disciplines including mathematics, logic, computer science, psychology, modeling and simulation, and software engineering economics.

Impact: Historically, libraries have greatly influenced society. Digital libraries of constructive algorithms and mathematics will certainly *encourage excellence in program construction* that will benefit the national infrastructures and, in particular, the information industries.

End BAA

2.2 Working summary of the BAA

Our understanding of this BAA is what informed our proposal. Here is how we interpreted it.

A formal mathematical understanding of computational tasks, algorithms to accomplish them, and programs to execute these algorithms has been indispensable to our ability to verify that algorithms accomplish their tasks and that programs correctly implement them. This understanding has deeply informed computer science teaching about programming.

A corresponding formal mathematical understanding at the level of systems will be indispensable to our ability to make them reliable and secure. A compelling methodology is to build systems from a library of verified components and designs; *it will require a digital library of formal algorithmic knowledge at the level of classes and systems as well as algorithms.* The fundamental ideas behind such a library will apply beyond computer science to the study of other complex artifacts and natural systems.

Objective: To create a digital library of formal computational mathematics and associated verified algorithms, formal classes, and reference systems that are useful for system building. By demonstration help nucleate a global resource of this kind that will be widely used.

Here are specific research questions associated with the goal of building and experimenting with such a digital library.

Research Concentration Areas

(1) Catalog those principal mathematical concepts, together with their formal definitions, that are used in *contemporary software system design* and implementation.

(2) Develop means of *certifying a combination of formal and intuitive reasoning about the concepts of (1)* in which the formal reasoning is checkable by machine and yet connected to intuitive reasoning about the concepts at the corresponding level of abstraction.

(3) Investigate languages and logics in which systems concepts from (1) can be most suitably expressed to achieve the goals of (2).

(4) Provide automated assistance for basic services to support a library of formal definitions, theorems, computational tasks, algorithms, and system modules appropriate to (1), using the languages and logics of (3) and the methods of (2).

(5) Investigate *means of formally composing algorithmic knowledge* about systems expressed in the languages of (3) that correspond to operations for

composing systems. Also assure that concepts and methods of certifying correctness, from (2), can interoperate across the various logics from (3).

(6) Investigate how *reflection* might play a role in (5), especially in regard to adaptable software.

(7) Investigate how a library of formal computational knowledge which provides the services of (4) and (5) can consistently manage *contributions from multiple logics* and verification systems of the kind determined in (2) and (3).

(8) Determine the features of syntax, language presentation, and proof presentation that facilitate good human computer interaction, especially given the goals of (2).

(9) Explore innovative metaphors and protocols for understanding algorithms and systems as presented in a library satisfying (1) through (8).

(10) Take into account in the design of the digital library issues of economic viability, intellectual property protection, and permanence.

Impact: Presentation of verified reference algorithms and reference systems will encourage excellence in programming, system design, system construction and support. Accomplishing these goals will create a new capability in *verification technology.* It will also allow a quantitative approach to understanding mathematical theories and software systems. Furthermore, sharing formal mathematical models, formal specifications and proofs of realizability from DoD projects will directly advance those projects, e.g. efforts at AFOSR, DARPA, ONR, including two other MURI projects, SPYCE and Language-Based Security.

2.3 Progress Toward Building Digital Libraries for Mathematical Knowledge

Here we outline our progress on the work called for in the BAA according to its categories. In Section 2.4 we factor the tasks as they were done in our proposal.

The Statement of Work (SOW) of that proposal remains in effect with one additional suggestion offered at the May 2002 review requesting that we look for a community of users. We address this topic last in this section.

It is important to note that *our proposal does not undertake the construction of a full-blown, industrial-strength digital library.* This would be a massive undertaking not appropriate for a research project with a large focus on graduate education. Our stated goal is to explore the issues involved in creating a large-scale system of this kind and to experiment with one or more prototypes that will help nucleate a larger world wide effort. Only 15 to 18% of our funding is for actual construction of a prototype software system (essentially one and a quarter full time equivalent person).

We present the topics in the order of our working understanding of the BAA. These are directly isomorphic to the BAA concentration areas, and we give the corresponding number as well, as BAA-*n.* We cite items on our project web page (www.nuprl.org/FDLproject) using the button names, such as [FDL Content], [Algorithms], [CIP], [Talks] and so forth.

2.3.1 (BAA-2) Catalog those principal mathematical concepts, together with their formal definitions, which are used in contemporary computing

One needs a formal system or systems in order to catalog the formal definitions. We use *three canonical representatives* of the major languages, logics, and systems used in the US and the European Community (EC): one, a standard classical logic used extensively in program specification, PVS (the EC uses HOL as well, a very similar prover); two, a constructive logic devoted to constructive mathematics, Nuprl (Coq is the corresponding EC constructive prover, it is closely related to Nuprl); three, MetaPRL, a *logical framework* having a constructive metalogic (the corresponding EC logical framework is Isabelle; it also uses a constructive metalogic, however, MetaPRL is the most modern logical framework, and it has capabilities found in no others). See FDL content.

Our collection allows experiments with cataloging, clustering, and indexing. It includes elements of a newly developed theory of distributed computing, *a logic of events,* that is critical in our effort to model actual computing systems of the kind most critical to the national infrastructure. It also includes new elements of computational graph theory, dependent records, red/black trees, constructive algebra, and functional program transformations. There are seven publications about these contributions

We catalog concepts based on standard categories, and we use automatic classification. We have compiled an index of concepts covered in the FDL libraries. There are 1128 PVS definitions, 1925 Nuprl definitions, and at last count over 1000 MetaPRL definitions.

List of papers, talks and implementation work related to this area and done as part of the project so far.

1. All 79 theories of PVS prelude imported [FDL Content]
2. Graph theory library developed in Nuprl, related to PVS and Leda [FDL Content]
3. Hybrid protocols imported (DARPA work) [FDL Content]
4. Logic of events imported (NSF, DARPA work) [FDL Content]
5. Finite sets, number theory, div, mod imported from PVS [FDL Content]
6. Nuprl list library reorganized (undergraduate project) [Algorithms]
7. Twenty two Nuprl libraries displayed with Dynamic Math Formatter [FDL Manual]
8. Abstract Algebra developed in MetaPRL jointly with Caltech
9. Constructive Set Theory in MetaPRL Master thesis at Caltech 02
10. Red/Black trees developed as chapter of PhD thesis at Cornell 03 [Algorithms]
11. Fold and Unfold article at LOPSTR 2003 [176]
12. Boolean Valuations article developed 2003 (Cornell funded)

In addition there is a draft article with Marktoberdorf student Nina Bohr from Arhus Denmark and R. Constable on using intersection types to define co-inductive types [31].

It should be noted that the use of MetaPRL and Nuprl was explicitly mentioned as a component of our proposed work. It is the best way for us to produce experimental content. These systems are DoD and NSF funded. Indeed, the project acquires about 90% of this content from DARPA- and NSF-supported efforts as well as Cornell funded contributions.

We are ahead of the schedule in our proposal in considering PVS already in the second year instead of the third. The proposal calls for including other systems such as HOL and Coq in the fourth year. However, the emergence of a Mathematical Knowledge Management community might make it very easy to do this by forming a Unified Formal Digital Library with the European Community. This could also bring into the library the results of the Polish Mizar system. So far this system has not stressed algorithmic knowledge, but that might change in the future.

2.3.2 (BAA-1) Develop proof-checking and model checking for certifying proofs of the standard body of computationally related mathematics

Cornell, Caltech, and Wyoming have been developers of proof-checking and property-checking (in the broad sense of SAT solvers) systems. This is a capability in which we have extensive experience and expertise. For instance, the Cornell PRL group and Gothenburg University built the first proof systems for general constructive mathematics. The other major constructive system, Coq, is closely related to the Cornell system (Chet Murthy of our project was a major architect of Coq).

The theory implemented by Cornell is Computational Type Theory (CTT), it is closely related to the Intuitionistic Type Theory (ITT) from Martin-Lof which is implemented in the Gothenburg provers (such as Alf). The ITT theory is also implemented in the Isabelle logical framework, and CTT is also implemented in the MetaPRL logical framework.

The Nuprl book is available on-line from the FDL project web site (it remains among the top 25 cited objects in CiteSeer).

We benefited from external efforts in improving MetaPRL reasoning and contributed to some extent from this project both at Cornell and Caltech. See TPHOLs '03 article on MetaPRL and Nogin's PhD thesis. This work included adding a new record mechanism and new rules for quotient types.

List of papers, talks and implementation work related to this area and done as part of the project so far:

1. MetaPRL article at TPHOLs by the Cornell/Caltech group 2003
2. Automating Basic Number Theory in MetaPRL TPHOLs article 2003
3. Reflecting Higher-Order Abstract Syntax in Nuprl, TPHOLs 2002
4. LICS 2003 demonstration of Nuprl reflection [Talks]
5. Representing Nuprl Proof Objects in ACL2 ACL2 Workshop 2002
6. Release of JProver IJCAR 2001
7. Aleksey Nogin's PhD thesis
8. Shared protocol verification Kopylov & Bickford 2003 [FDL Content]

2.3.3 (BAA-3) Investigate suitable base language and logic within which competing logics can be expressed and evaluated

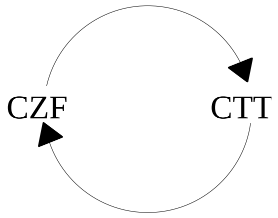
This concentration seems to call for work on *logical frameworks* as a base logic. We have in the past explored metalogical frameworks [19] and the MetaPRL logical framework [90]. We have also explored per models [7] and set models [91]. Out of this work a central concern emerged - relating classical and constructive logics.

We have done fundamental theoretical work in this area, based on Hickey's PhD thesis [81] and Howe's methods. We have also experimented with the following concepts.

We studied mappings of Sets → Types [81], and we implemented Aczel's mapping from CZF into CTT [133].

We studied mappings of Types → Sets. [90, 133], and we will be using Howe's implementation of his mapping of HOL into Nuprl.

We studied the composition of these mappings, thus closing the loop:



List of papers, talks and implementation work related to this area and done as part of the project so far.

1. MOD03 paper on work of Howe, Moran 2003 [45]
2. Nuprl-PVS connection article 2003 [Publications]
3. Translation service experiment 2003 [28]

Moran's Theorem is a major new discovery from July 2003; see MOD03, Lecture 3 included among the papers accompanying this book.

Stuart Allen's work on certificates addresses this area as well, but we listed it in Subsection 2.3.5

2.3.4 (BAA-4) Provide automated assistance for routine aspects of developing libraries of formal theorems, proofs, algorithms, and their expressions as programs

MetaPRL and Nuprl provide this automated assistance already. We expanded these methods to work in a multi-logic digital library. The FDL provides general mechanisms for these routine tasks. In the proposal we thought we might call this a Common Logical Library. For a list of the proposed operations, see Goals, Section 1.1

The group at the University of Wyoming (Caldwell/Jechlitschek) is developing a prototype peer-to-peer system which allows distributed FDLs to be queried and accessed form a single site. Currently our query language included Boolean combinations of component (opids) and named based, but we intend to extend it to include queries that require inference. This prototype system can serve as a model for FDL front-ends.

List of papers, talks and implementation work related to this area and done as part of the project so far.

1. FDL implementation's extensive services [FDL Manual], revised 2003
2. The FDL Design Notes discuss additional services 2002 [FDL]
3. The Dynamic Math Formatter is a key service [FDL Manual] improved 2003
4. The Dynamic Pure Structure (DPS) editor is being developed for the FDL 2003
5. Wyoming peer-to-peer experiments

The FDL implementation is 95,000 lines of code.

The Web project software is 6,000 functions, and the FDL itself contains 20,000 lines of service code.

We have project over 45,000 objects to the Web.

2.3.5 (BAA-5) Investigate forms of assured interoperation for assembling, composing, specializing, and generalizing algorithmic knowledge

The concentration area can be read in three ways. We have results about interoperation of system components (a goal of DARPA's PCES project) and interoperation of distinct provers. The second category is the topic of our proposal. In the FDL we have explored:

(a) Formal translation between theories (CZF to CTT); Hickey thesis (b) Hybrid proofs (c) Translation among FDL theories (d) Accounting schemes that provide assurance, the certificates of our FDL; see FDL Manual and Design Notes.

In addition, we have studied the idea that formal theories provide the basis for collaborative articles in ordinary mathematical vernacular that can interoperate because they are grounded in the same formal definitions. See the article by Allen and Constable [6].

List of papers, talks and implementation work related to this area and done as part of the project so far.

1. Abstract Identifiers and Textual Reference 2002
2. Using the FDL for semantic anchoring in the NSDL 2003

The paper by Stuart Allen on abstract object identifiers is fundamental. It among the papers that accompany this book.

2.3.6 (BAA-6) Investigate reflection for coordination, interoperation, and dynamic adaptation

We proposed the study of coordination and adaptation of a library to changes in the basic theories and the systems they support.

We conducted a major study of the difficulties caused by change - the need to change definitions, theorems, the need to revalidate. We also studied the impact of improvements to tactics. Our investigation led to several new concepts for controlling change - abstract object identifiers, closed-maps, and certificates.

Our investigation into reflection led to a more usable reflection mechanism for theories based on the FDL term structure with binding; see LICS 03 demonstration and a TPHOLs paper on higher-order abstract syntax.

List of papers, talks and implementation work related to this area and done as part of the project so far.

1. Reflecting Higher-Order Abstract Syntax in Nuprl TPHOLs 2002 (repeated item)
2. LICS demonstration of reflection 2003 (repeated item)
3. Computational complexity and induction article 2001

2.3.7 (BAA-7) Study issues of consistency and maintenance among libraries

We proposed an investigation of ways of maintaining consistency in a multi-prover library. We have studied the issue of creating dynamic formal documentation that remains current as software evolves.

List of papers, talks and implementation work related to this area and done as part of the project so far.

1. FDL Design Notes 2002 (repeated item)
2. FDL Dynamic Latex editor
3. MOD03 paper 2003 (repeated item)
4. Evan Moran thesis draft 2003 [133]

2.3.8 (BAA-8) Address the human/computing aspects of syntaxes and concepts that are appropriate for both foundational developers and end-users

MetaPRL and Nuprl use display mechanisms developed by the PRL group and brought to a stable state in Nuprl 4; see Nuprl 4 Manual.

We have provided more advanced versions of these mechanisms for the FDL as well by improving the Nuprl display mechanism. We have created a Dynamic Math Formatting feature as part of the FDL.

We illustrate the capability in the PVS Library, but the real impact comes from the dynamic redisplay mechanisms and context sensitive display; see Allen/Manion article [116].

We are also creating a Generic Mathematics Editor for the FDL which will provide structure editing features and which will benefit from continuing work on editors for mathematics.

List of papers, talks and implementation work related to this area and done as part of the project so far.

1. FDL manual on display and syntax (repeated item)

2.3.9 (BAA-9) Explore innovative metaphors and protocols for understanding, using, composing, searching, authenticating, and validating constructive results

In the past we have based our explanation on the concept of evidence, producing a *semantics of evidence.*

We are now exploring the *knowledge metaphor* with Joe Halpern. This work has led to collaboration on the SPYCE project through results of PhD student Sabina Petride [152]. We are looking at algorithmic knowledge in the sense of Fagin, Halpern, Moses and Vardi possessed by agents in a distributed system. The fact that knowledge in logics such as CTT and ITT can be recorded as types means that it is natural for a protocol to specify what it currently knows.

The knowledge metaphor has been used by Kripke and was formalized constructively by Judith Underwood. Related work led to Search Algorithms in Type Theory by Caldwell, Gent, and Underwood.

List of papers, talks and implementation work related to this area and done as part of the project so far.

1. Knowledge based protocols in the logic of events draft 2003 [152]

2.3.10 (BAA-10 and May 2002 Review)

At the May 2002 project review at ONR, the outside review panel suggested that we work to develop a community that would especially value the FDL, several suggestions were made, including the Naval Research Laboratory.

This was also a lively topic at the first North American meeting of the newly emerging Mathematics Knowledge Management community. This particular community is definitely a likely one to encourage. We are already members, having presented two papers already at these meetings. In addition, R. Constable has twice been invited to speak at the meetings.

We have also determined that the higher-order theorem provers conference, TPHOLs, is a very good community for us.

List of papers, talks and implementation work related to this area and done as part of the project so far.

1. Interactive digital libraries article at MKM workshop, 2001
2. Invited talk at North American MKM meeting 2002

3. Invited talk to be given Nov 2003 MKM meeting

In addition we have presented 4 papers at TPHOLs in the past two years.

2.4 Proposed Plans for Building Interactive Digital Libraries of Formalized Algorithmic Knowledge

In the heading of this section is the title of the Cornell, Caltech, Wyoming proposal which was the only one selected for funding in this topic area.

In this section we factor our progress report according to the categories used in the proposal and discussed in the Introduction to this collection. We quote research issues directly from the proposal. Recall that the two major categories are *content creation* and *library infrastructure.* In both categories we work on theoretical and experimental issues. The later require extensive implementation work.

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.*

Second, we contribute to formal *mechanisms for guaranteeing the reliability of large software systems*. We will discuss in Section 2.4.1 how 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 the means to connect intuitive textual documentation to formal documentation.

Third, a logical library will *complement the mechanisms of electronic publishing* and open the way to verify journals that specialize in formalized mathematics [129, 160]. 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 publicly scrutinized (this is a system that obeys the so-called *deBruijn principle*).

Fourth, there is *significant educational value* in formal reference material. We have used such material in teaching and have studied its impact [44]. In particular one can learn about a particular system in a context where the design, the specifications, the algorithms and the proofs are all linked to the relevant literature. Moreover, 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.

Fifth, 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 generated natural language proofs from parts of the Nuprl corpus [87]. These methods will now work on the entire FDL.

Sixth, public access to this global interactive digital library of algorithmic mathematics will benefit the nonexperts 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 obviously 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.

2.4.1 Content creation - theory

One of the themes of our past work was to automate the richest formal language of computational mathematics that we knew how to design, and automate in a way that is extensible. Our experience is that expressiveness greatly leverages theorem proving power.

Based on our past experience, we believe that the case for expressiveness is clear: formal concepts must match the natural mode of discourse used by people who want to read the library and interact with it. This means that ordinary first-order logical language is not sufficient, and in our view, even higher-order logic has benefitted considerably from adding dependent types, as in PVS, and it would benefit still further from subtyping relations. These points can be made well in comparing the notion of subtyping and inheritance used in the type systems of programming languages with that used in pure mathematics [49]. *We see that computational type theory can express the programming concepts, but traditional set theory can not do so directly.* The FDL will encourage connecting formal statements to their intuitive equivalents, and to the richest formalisms that are related.

We are focusing on two hard problems in formalizing computational mathematics. One is the issue of reasoning at a larger scale – about classes, theories and systems rather than about types, theorems and programs. This forces us to examine even more closely the problem of reflection and metareasoning which has been of interest to us since the 80's. Ideas for solving these problems have a strong effect on the design of the logical library.

The ability to reason about classes and systems will allow us to explore the notion that a software system can be thought of as an implemented constructive theory. We call this idea *theories-as-systems*, and it is a generalization of the concept of *proofs-as-programs* that we pioneered. We think that this approach is supported well by the class theory operations that are included in the Nuprl type theory [49, 100].

An important issue for building content is establishing links between the logical library and a variety of proofs systems. This is precisely the issue of relating theories and proving at a large scale. Our FDL now has the capability to connect to other processes, and we intend to use this for building connections to HOL [69], PVS [148], Maude [42], Isabelle [150], and ACL2 [3].

Providing a practical implementation of *reflection* for open ended computational theories will not only allow us to manage the library and prove that large scale operations are correct, but it is also a very good route to formalizing computational complexity. The fact that *computational complexity* facts are not part of the standard libraries reveals a major gap in the foundational basis. We think we can now close this gap [46, 24]. Benzinger's work also illustrates the value of connecting theorem provers to symbolic algebra systems, he uses *Mathematica* as a heuristic to estimate the computational complexity of code extracted from proofs (it is not used in proofs). The FDL will allow us to extend these ideas to all content collected in it.

2.4.2 Content creation - experiment

If we are to have a semantically grounded theory of programming “in-the-large”, then we need a semantics of classes or modules [86, 1, 54, 26], and it must be related to an account of types. Programming languages such as Java support a notion of class, but it is not formal and cannot be used as the basis of a theory. Various ML dialects support modules which do have a semantic basis, but ML does not support a rich notion of inheritance, and modules are not objects in the full sense.

A pleasant surprise about constructive type theory is that it supports a rich theory of classes which can account for modules, formal modules, dependent records, and objects [49, 33, 154]. A formal module is a module which contains assertions as well as types and methods. The class theory also allows us to treat theories as objects. For example we could define a computational number theory as a module built on a discrete nonempty type, D, and containing a successor operation, a zero element, a decidable equality operation and two recursion combinators, one of type $D \times A$ into B and one of type $D \times A$ into $\mathbb{P}_i$ for A and B arbitrary types in $\mathbb{U}_i$. A remarkable feature of type theory is that this little theory of numbers is also an example of a module with assertions or of a dependent record whose types are propositions. We have discovered a very simple way to define these dependent records using dependent intersection types [100].

Relations and functions on types extend to classes and thus to theories. For example, a theory morphism is a special kind of function from one theory to another. Also the subtyping relation on types, $A \sqsubseteq B$, provides an inheritance notion for classes and for formal theories. We know precisely what it means to say that one theory is a subtheory of another (and we see that a classical version of a theory T is a subtheory of the constructive theory T).

We have found that our use of formal modules in the verification of Ensemble [111] has allowed the components to connect more precisely; it is like having Lego blocks with many pegs and holes so that they “snap together” tightly.

Our systems can automatically extract algorithms from constructive formal proofs; these algorithms are correct-by-construction. In a similar way we can synthesize a communication protocol from its specifications. The formal proofs explain why the algorithm accomplishes the task specified by the theorem. We pioneered this technology [20, 48, 36, 39] and have been using it since 1980 with a number of major successes — from digital hardware to distributed system protocols [110]. One of the suggestive slogans used to describe this work is *proofs-as-programs*.

In the context of the FDL, we can now expand this proofs-as-programs concept to the notion of "theories-as-systems" [49]. The notion is that a small theory, say of managing *histories of computational events*, becomes the description of a small system. We are applying this notion in the case of using a formal theory of transition systems to describe communication protocols.

We propose to develop this idea further to show the value of the concept of an "active" library of algorithmic mathematics. If it is feasible to maintain the code base for a module of a real system in the Library and link it to formal documentation automatically, then the Library will prove itself as a tool in building better software. We already support parts of MetaPRL code in this way, but we have not established formal links to theorems that underlie the design.

Using the above concepts we plan to improve links among different provers. In some cases the links will be made by simply importing theorems as we have done for HOL into Classical Nuprl and use them in parts of libraries where computational content is not required or where classical content is equivalent. In other cases we will use a semantical embedding of the corresponding theories as Hickey has done for constructive set theory. A still deeper level of connection can arise in cases when we can invoke other provers in building *hybrid proofs.* Here are some of the systems that we will examine as we expand connections to the Common Logical Library.

- The HOL proof system [69], using the HOL/Nuprl link described by Felty & Howe [90, 91, 63] and the link via Maude [43].
- The PVS system [148], using techniques described in Moran's forthcoming thesis [133].
- The formal meta-tool Maude [42], using the link described in [43, 126], and MetaPRL [77, 127], developed at Cornell.
- The generic theorem prover Isabelle [150] using insights gained in [137].
- The ACL2 system [3].

2.4.3 Library infrastructure - theory

The main purpose of a logical library is to provide formal mechanisms for guaranteeing the reliability of stored information and for connecting textual documentation to formal documentation. They will support the creation and manipulation of various kinds of objects with logically significant relations between them. The correctness of our claims and explanations depends on these relations which will be accounted for within the library.

Logical Accounting Accounting in the logical library is intended to support arguments for claims of the following form:

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

We will design and implement an *accounting system* able to determine how to execute an inference as specified by a tactic. The tactic source code together with the goal of the inference step refer, directly and indirectly, to part of the library. This part should suffice for determining how to perform the inference. The accounting methods are to be defined largely in terms of such source code and expressions. For example, we define criteria which guarantee that we know how to deterministically execute tactic code given that we can find all the library objects referred to. This is what it is for the code to specify an inference rather than simply being a record to the effect that somehow, sometime in the past, there was some interpretation of the code that produced the inference recorded in the proof. Ultimately, knowing how to execute a piece of tactic code is knowing how to construct an inference engine for executing it; this is part of what must be accounted for.

2.4.4 Library infrastructure - experiment

The major technical challenges in trying to provide library services arise from features special to formal logical libraries, mainly:

- *Theories contain code* in the form of tactics. Arguments about validity of tactic proofs are partly about the result of executing tactic code, therefore the meaning of such code should be stable. We also sometimes want to reliably rerun tactic code in order to recheck objects.
- *Dependencies* arise from logical connections among objects. Some dependencies are created by executing tactic code which makes it hard to track them.
- Theories can be very large and might take days to recheck, so there must be a way to *perform local checks and rebuilds*. Also, there is a pressing need for making the tools faster.

- The theorems in the library can be created by various tools in different logical theories with different foundations. We need to be able to express relations between logical theories in order to be able to translate theorems between different theories.
- It is essential to formalize both the object level theories and aspects of the metalevel, this leads to questions about *reflection* and *metalevel reasoning* which are some of the deepest aspects of formal systems.

Library Structure The logical library is organized as a *persistent object store* for formal mathematical knowledge such as theorems, proofs, definitions, algorithms, formal theories, informal items such as documentation or explanations, and also proof tactics and search heuristics. Ordinarily older versions of objects will be retained in the library store so they may be recovered or even combined with newer versions. These version control methods are also applied temporarily or locally in order to enable users to undo changes or recover from session failures. Garbage collection methods are employed.

Chapter 3

Critical Infrastructure Protection and High Confidence Software

At first sight it might seem like a far-fetched idea that a formal digital library of computational mathematics is important to the defense of the nation's critical software infrastructure. Nevertheless, that is true, as this chapter explains.

Understanding this connection starts with knowing that we do not yet have an adequate methodology or technology for building highly reliable and secure software systems, although we can build reliable, functional programs if they are small enough. However, a technology is emerging for building reliable systems. We will call it a *verification technology.* One way to understand it is to compare it to the so-called stack — the layered collection of hardware and software of which a complete system is composed. Parallel to this stack is an *emerging verification stack.* Its components can be integrated into the programming tools used at various levels of the stack to help make the resulting layers much more reliable. The DoD has invested in several components of the verification stack and has funded some of the research that has brought it into being.

Our formal digital library (FDL) is an experimental new component starting at the top of this stack and providing a *semantic backbone* for the whole enterprise. The semantic backbone is critical to reliability in itself, and it also improves other elements of the verification stack. It also facilitates a programming methodology for large software systems. It will play a role in the *design* phase, in design *refinement,* in *specification,* in code and documentation *production,* and in the coherent *evolution* of the system. The FDL holds vital knowledge about a particular system, and it provides an interface to general knowledge needed to understand, modify it, and relate it to other systems. The FDL holds knowledge about the algorithms and protocols in the system, about what they do and how they connect. This knowledge is formally linked to critical components of the system, and it is linked to general knowledge about algorithms, data structures, and other systems.

Just as lower levels of the stack provide networking links that connect a system to other systems, the FDL connects system knowledge to the network of knowledge that is essential to understand it and support it. It is also a basis for quantitative data about the system's design.

The FDL is the interface between knowledge that is checked and generated by machines — *formal knowledge* — and the kind that can only be checked and generated by humans — some call it *intuitive knowledge*, or by contrast, informal knowledge when it is in the context of formal knowledge.

3.1 The Issue of Software Reliability and Security

3.1.1 Importance of software security and reliability

The NIST sponsored study in 2002 found that errors in software cost the US economy $59.5 billion annually. We think that is only a lower bound on the cost. Annual federal government sponsored research on the problem is less than 0.1% of this cost. We also know that software systems are a significant part of the nation's critical infrastructure. Software is used to control air traffic, the power grid, the Internet, the Web, the stock exchange, banking transactions and the operations of most large institutions, private, public and governmental — including the research laboratories and medical facilities on which we depend in times of crisis.

Vulnerabilities in the software infrastructure propagate into the physical infrastructure, and the weakest link can bring down many other vital elements.

Popular books such as *Fatal Defect* [151] and other scholarly studies by MacKenzie [115] have studied the problems and funding initiatives proposed to correct them. Modest federal and industrial funds have been spent for the past thirty years to improve programming technology, and there has been significant improvement. Likewise programming methodology has become more mathematical and rigorous and programming practice is more informed by good education.

inputs in advance of execution. Connecting proof technology to programming technology has allowed us to make guarantees at the level of formal mathematical certainty, the highest level of confidence known. When hardware failures are taken into account, other guarantees can be given to arbitrarily high level probability of correctness.

In principle we can even create a ”verified stack” consisting of formal models and verified processes that correctly transform a verified high level program into machine code on a verified chip [92, 132, 131, 35, 74]. Research is slowly expanding this paradigm, and both the hardware industry and software industry are investing in it. It is a paradigm created by fundamental computer science research over a thirty year period.

In spite of these successes on programs, the solutions do not scale to software systems. An observer of technology transition might expect that improvements in the technology would have led to a complete solution to the ”software crisis”, within the typical 20 year time frame for laboratory results to mature in industry [32]. *However, none of the major components of modern software have been stable over this period.* The hardware platforms on which the software executes have become increasingly complex, thus the compilers and operating systems change. The applications software continues to increase in size and complexity, pushing the envelope of our understanding in areas such as distributed real-time embedded systems.

Thus, fundamental research has been directed at a rapidly moving target of steadily increasing scale and complexity. At the same time, these systems have become pervasive, making the problem of reliability more and more critical to society. This situation has led government and industry alike to invest more in the problems of software reliability and security – presumably leading to the BAA on Critical Infrastructure Protection and High Confidence, Adaptable Software Research, and Topic 8, Digital Libraries for Constructive Mathematical Knowledge (see Chapter 2, and also our FDL web page).

We will describe the elements of the emerging technology below. We have been players in creating the standard paradigm, and are known for our work on the foundations, on tools, and on applications. It is a technology that has already significantly mitigated the problem (especially in hardware), and with sufficient funding for further innovation and improvement and with further engagement by industry it can "catch up" and become a basis for solving the problem of scale. However, even at the small scale, the emerging technology is incomplete and progress on the various elements has been unbalanced. Researchers have spent 30 years building powerful theorem provers and sharing the symbolic algorithms used in them, and yet they have spent no time making large collections of formal knowledge that can be shared. Government and industrial funding have helped create the powerful provers *but there has been essentially no funding for the knowledge base.* ONR/OSD is a leader in this regard.

Our OSD MURI project provides a new element to expand the standard technology and move it toward entire systems; that element is a formal digital library (FDL) of formal proofs about algorithms and protocols, and the mathematics needed to understand and extend them.

3.1.2 What is the technology for building reliable systems?

The standard approach to providing a programming technology that will produce significantly more reliable and secure software has these elements.

The basis is a mathematical account of software. It was a major accomplishment of the field of formal semantics in the 70's and 80's that we are able to view software as *both an artifact of the physical world and an abstract mathematical object.* This fundamental achievement, established by a great deal of theoretical and empirical work has organized a world-wide common research agenda of considerable power. Several Turing Awards have been given for contributions to this agenda, e.g. McCarthy, Hoare, Floyd, Dijkstra, Cook, Scott, and Milner. Some of the most respected mathematicians have also contributed to this approach, e.g. de Bruijn, Bishop, Martin-Löf, Girard, Barendregt, Aczel. We participated in this foundation phase in the 70's [59, 47, 51], including writing a book on the topic [50].

The mathematical foundations are fundamental to this MURI effort because we are collecting the formalized theories distilled from this work. We also embody the two major strands of work that this agenda calls for. One is a mathematical methodology of the kind advocated by Hoare and his followers and presented at the world famous NATO summer school that he and Dijkstra founded and Hoare still directs, *Marktoberdorf.* The other is the verification system approach made feasible by Milner and a host of computer scientists who build formal tools.

The mathematics-based methodology relies on education and access to well-explained reference algorithms. This approach succeeds for small algorithms, and it has become part of the standard curriculum in European Community and North American universities. Our FDL supports this approach by showing how to collect reference algorithms and interconnect formal and informal explanations of them and relate them to the code. One difficulty with this approach is that as formulated so far, it does not scale well to systems, because they are more than sets of algorithms. They have structure.

Parts of software engineering are attempts to scale this approach to the level of systems, but it is difficult to present ”reference systems” as a useful educational tool. It is not good enough to simply present the code with informal documentation. There should be tools for tracking the dependencies of modules on the design invariants and on the global requirements. We think that the FDL contributes to solving this difficulty, but that is not the main point of the BAA nor our proposal. The DL topic of the BAA mentions the mathematical methodology in the prelude and in two of the research concentration areas (items 2, 9, and possibly 10), *but its main thrust is toward the formal tools and proof systems* mentioned in six of the concentration areas (items 1,3,4,5,6, and 7).

A formal semantics for programs allowed us to build tools that would support the process of translating specifications and high level demonstrations of implementability into code. Such tools are able to guarantee properties of the resulting code base. Some of them can be easily integrated into standard heavy machinery such as compilers and version control systems.

The first level of integration is into the type systems because the idea of a type can be made richer based on the mathematical semantics. For instance, the type systems of programming languages have become progressively richer as our mathematical understanding of types has deepened. Now there are tools that infer types as well as check them – Algol 68, Simula, Ada, ML, OCaml, Java, C#, CCured, Cyclone form a lineage of programming languages with ever increasing type richness.

Program analyzers, such as PREfix, CodeSurfer, LCLint, and ASPECT, are used to analyze memory use and find a certain class of simple programming errors. Microsoft has done controlled studies which show the efficacy of these tools in finding common bugs.

The formal semantics allows us to extend type systems to include formal expressions that describe important properties of the code. Among these properties are freedom from deadlock, array reference within bounds, sensible pointer reference, etc. Tools are made that check for these properties. This process creates supplemental formal languages that are called *specification languages* because they help specify properties of a problem solution and of code. The semantics for these specification languages is the formal semantics discussed above.

A formal mathematical semantics for computing tasks allows us to use computers to ascertain whether design level reasoning is correct. Specification languages and formal models provide such descriptions of computing tasks, and they raise the level of abstraction at which computations and computing problems can be precisely stated. They allow designers, programmers and managers to talk about solutions and constraints in a precise way. Because the languages are formal it is possible to translate them into programs. It is interesting to note that the most primitive kinds of specifications are test suites, these are finite. Specifications can be seen as substantial generalizations of test suites.

Specification languages also help create more precise interfaces between subsystems and between classes (or objects). This is illustrated well in protocols and classes that also contain assertions. It is quite remarkable that specification languages can simultaneously raise the level of abstraction and the degree of precision in a system.

Specification languages also allowed a class of tools called *model checkers,* which have been used to locate errors in individual programs and in systems. They work by attempting to find a computation path in the system that violates a property required of the system. These tools have been very successful in hardware, and they have opened a class of checking methods that can be applied in other tools as well.

Another class of tool made possible by having a precise mathematical meaning for programs are the *theorem provers.* These can also be used to find errors, a different class of errors than the model checkers can find. The

PVS system has been designed to work especially well in this error-finding mode.

Theorem provers are used to demonstrate the absence of errors and to prove that programs and systems actually solve computing problems that can be stated in a precise way. By a precise description, we mean one describable by modern mathematics. When operated in this mode the provers are called *verifiers.* There are about a dozen of them being used and studied. They are advanced tools that are expensive to use, but they are the only tools that can deal with certain critical properties. These tools are explicitly mentioned in the BAA.

The theorem provers can also be used to synthesize programs that are known to be *correct-by-construction.* This is a remarkable capability pioneered in the US at Cornell and Kestrel and now also used extensively in the European Community (EC) as well.

They are tools that play a central role in our proposal and in all of the EC efforts to build formal digital libraries of computationally related mathematics. The provers introduce an entire subtechnology that we call *proof technology.* It is central to the BAA and to our proposal.

One of the salient features of provers is that they are inflexible. They must enforce every detail of an argument, from exact syntax to exact semantics and perfect proof rules. They are like space technology in requiring extreme precision. This makes them rigid and difficult to change. The current way to determine whether an improvement in a prover is acceptable is to rerun every proof ever performed by the system. This is not feasible in practice, so the result is that even the community of researchers using a single prover becomes disconnected as it evolves. Some people use version n while others use version n+1 or n+2, etc.

The technology we have described so far is advanced both in its goals and in the science required to achieve them. There are special cases of it which are immediately applicable in many of the categories we cited, such as advanced type systems, extended static checkers, code analyzers, model checkers and so forth. The versions that are immediately applicable provide good evidence that the standard technology is on the right track. We expect that more and more advanced tools will appear as results of basic research mature, so the solution will be incremental. However it is very important to have all the key components present so that they can all mature. At present, an FDL component is missing entirely. Because our work is providing a missing component, it is already generating interest.

In summary, one can summarize the technology we described by naming the algorithmic tools associated with it. Here are the key elements: type checkers, type inference algorithms, extended static checkers, program analyzers (PREfix), model checkers, decision procedures, verification condition generators, program development environments, interactive provers, automatic provers, and code extractors. **To this list we are adding the formal digital library.**

3.1.3 What is a formal digital library?

A digital library is a collection of documents in digital format along with programs that operate on them to provide user services associated with research libraries, such as storage, classification, search, retrieval, archiving, and so forth. The tools can provide additional services such as automatic summarization, transformation, annotation, hypertext linking, presentation on the Web, and so forth.

We call the collection *formal* when it contains a substantial number of documents and services whose meaning is given by a formal mathematical theory and *whose validity can thus be checked by computer programs.* Such libraries provide services based on the meaning as well as on metadata.

Formalized mathematics is an example of content that is completely formal. The axioms and inference rules provide the meaning. Proofs can be checked for correctness, and they can be assembled with machine assistance. So a digital library (DL) that stored formalized mathematics is a formal digital library (FDL). We can see from the above account of modern programming technology that mathematics is what enables the precise semantics of programming languages, the basis for the entire research agenda in reliable and secure programming. Formal semantics is the basis of the automated tools that create the technology of trust.

In the case of formalized constructive mathematics, its definitions and proofs have computational content, that is, there are implicit algorithms in the proofs that can be mechanically extracted. We say that the mathematics is computational if it includes algorithms and proofs about algorithms. The BAA calls for libraries of constructive mathematical knowledge. We are one of the world leaders in producing this knowledge using tools such as MetaPRL and Nuprl.

3.1.4 Role of an FDL in critical software infrastructure protection

One reason for our project is that critical software infrastructure protection (CIP/SW) is important to the DoD, and this is technology is one of the most advanced ways to attack the problem. Because this is an area in which the United States must be unsurpassed, since we have more at stake than anyone, *it is essential that we understand and command the most advanced technology available.* The FDL will not only add a new element to the set of tools, but it will significantly enhance and amplify existing tools, and it will offer the means to scale both the programming methodology and the technology for building more reliable systems.

The new capability offered by the FDL is a common repository of definitions, theorems, and proofs in formal computational mathematics. It collects the formal content from multiple interactive theorem provers such as Coq, HOL, Nuprl, MetaPRL, PVS and others. A common set of library services will apply to this collection, which is already quite large (over 15K theorems) and growing. There is currently no repository of this nature except for our FDL, although the EC is supporting similar efforts, such as OMDoc/MBase and Helm.

The FDL will enhance the standard technology of trust in several ways. It will capture the precise knowledge that informed design decisions. It will capture the knowledge used in *design refinement.* It will allow presentation of entire software systems and their formal documentation as reference systems; and it will provide the logical dependency tracking among the modules needed to make such a reference system useful. We are demonstrating this capability for distributed systems. The FDL will also make proof technology more efficient as we illustrate later. Most critically, it will make this technology more flexible. It is a lack of flexibility that is one of the limitations in deploying these tools more generally and at a larger scale.

By achieving flexibility, we also allow provers to share collections of formal mathematics. This makes them more efficient as well, and makes specification languages more flexible. Since exact formal specifications of algorithms and protocols are developed by each prover, these can be shared. Sharing will shorten the time to develop new specifications that build on previous ones. In addition the mathematical models used to justify specifications can be commonly referenced.

The kind of sharing we are talking about is not the ”cut and paste” kind, but each system will be able to incorporate archival elements into its own developments and even automatically translate specifications from one logic into another. The amount of savings will depend on the amount of material that can be shared. In the case of mathematical tasks, where there is a lot of commonality, the savings are immense — proportional to the conceptual difficulty, and perhaps on the order of weeks per definition.

The task of interactive theorem proving will benefit significantly from the FDL. Current provers do not have access to large amounts of knowledge, and there is a strong tendency to just re-prove many small lemmas, rather than finding them in another prover's library. Discoveries in AI have established the need for large knowledge bases in systems that attempted to exhibit intelligent behavior [107].

The history of research in theorem provers shows that a great deal of effort has gone into providing better logical algorithms, such as pattern matchers, rewrite algorithms, decision procedures, and so forth. But it did not make sense until recently to fund the creation of massive amounts of content. Although this is not the point of the MURI BAA, it would now make sense to fund a seven-year program to accumulate a *massive amount* of ordinary mathematical knowledge to be shared by the provers. A system such as our FDL would make this possible.

Now, after almost two decades of steady accumulation of results, there are over fifty thousand — 50K — theorems available world-wide in digital form. This is a very large amount, but it has not been collected. We are building and exploring prototype systems for collecting this material, and providing library services for it.

- The task of creating correct-by-construction programs and protocols will be sped up to match the speed of ordinary programming; this will be a factor of 5 to 7 speed up.
- The process of design, coding, and verification will be supported by the library so that these activities are automatically interconnected and cross checked; this will be a new capability.
- A new kind of documentation will be available which will be logically connected to the code, we call it formal documentation; it will improve our ability to modify the code without breaking it.
- A Formal Digital Library of computational mathematics has significant intrinsic value as part of an emerging global knowledge resource. It will

accelerate scientific discovery, it will enrich technical education, and it will enhance our capability to create better software. In particular, it will be directly applicable in protecting software infrastructure.

3.2 Role of an FDL in the Programming Process

3.2.1 Sharing formal mathematics

In principle the idea of sharing formalized mathematical knowledge is sensible and obvious, yet many people who know what is involved think it is impossibly difficult. Here are some of the challenges that must be overcome.

The problem of different logics

Sharing formal mathematics is not as easy as it might at first seem. Among eleven significant interactive theorem provers, there are nine different logics for the four major theories. Five of these use a classical core and four a constructive one. Thus a statement in one logic might mean something very different in another. For example, consider the simple statement

1. For all numbers n, Prime(n) or not Prime(n).

   This says that every number n is prime or not. In the classical logics this is a trivial assertion. It is true because by the LEM, law of excluded middle, every statement P is true or not, that is

2. For any assertion P, P or not P.

   In constructive logic, the statement (P or not P) means something very strong, it means that we can decide whether P is true or not P is true, and we have a proof of the one that is true.

See the article *Information-intensive Proof Technology* [45] for a discussion of these issues. It is included in the appendix to this book.

3.2.2 Accounting for validity under change

Although it seems like an oxymoron to say it, *change is the main invariant of our business.* Software systems must change because the world in which they are designed to function changes. It is not that the mathematical truths which justify software change, but the computational justifications of truth change as we learn better and better ways to capture the reasoning of programmers, mathematicians and computer scientists. As the reasoning systems improve, the justifications that computers "understand" becomes more compact. This section illustrates this problem.

The problem of code and state

Some provers use programs as part of their proofs, especially decision procedures for arithmetic. Many use the SupInf procedure. Some use Arith. How do we know that these programs are correct?

One method is to prove the decision procedure correct. Another method is to build a primitive proof. MetaPRL supplies these.

Provers are executed in a context that provides theorems that can be automatically used. In addition, most provers build *state* as they construct a proof; this state must be preserved and inspected to track logical dependencies. For example, the prover might build a temporary list of lemmas that provide typing for subterms.

Many of the provers allow program code as part of the proof. This is typical of the class of tactic-based provers. For these provers, the validity of a proof may depend on properties of the programming language for tactics.

If this program code is modified, say, by improving a tactic, then all previous proofs that used the tactic might change, say become smaller, or they might fail. Thus when new tactics are added, there is a problem of knowing that previous results can be recovered.

The problem of name spaces

One problem with collecting formal material from independent authors is that they use names inconsistently. For example, there might be several definitions of prime number, or several variants of the fundamental theorem of arithmetic. When we are working in separate provers or in hierarchical theory spaces, it is easy to create unique identifiers to disambiguate the names, such as pvs-peano-arith-prime. But different users of the same theory should not be burdened this way. So we want mechanisms for managing naming conflicts. We discuss these matters in Chapter 4.

Name space problems do not only arise from inconsistent usages by different users. The problem is that, in current systems, references to objects (lemmas, definitions, tactics, etc.) are made via symbolic names. But this means that if, as mentioned in the previous section, a tactic is improved, it may no longer produce identical results in contexts in which it has been previously used; if so, proofs may fail when re-run. To avoid this problem it would be possible to give the new improved tactic a new name and to leave the name of the older weaker version unchanged. But names are typically mnemonic – and there may be no appropriately suggestive synonym for the new tactic. The other option is to go back and repair all the older proofs so they work with the improved tactic, but this can be a huge effort and it is not logically necessary. This seemingly trivial problem applies to any formal artifact, software included, and can become serious in any system that is around long enough. It does not arise at all if symbolic names are not the principal means of referring to objects. In the FDL, user selected names are maps from the symbolic name to the object it refers to. This level of indirection solves many name space problems that exist in a wide range of applications today. In the example of the improved tactic, if proofs refer to tactics by a reference to the abstract identifier of the tactic object itself, old proofs will still run even after a user has remapped the name to refer to the new tactic. Persistent, abstract, object identifiers are the basis of the underlying FDL representation of formalized mathematics.

Need for working space, sublibraries (working maps)

A common problem with all documents is that we distinguish between drafts and final copies. A draft might be in an inconsistent state. There might be several different proofs because we are trying out alternative approaches, looking for the best one.

Another situation that arises with large collections is that we want to use the same results in many different documents, yet we don't want to duplicate them.

The simple act of combining definitions and theorems into a common collection requires that we solve several of these problems. In order to actually share results in a meaningful way requires solving all of them. We have developed an approach to all of these issues, and are experimenting with our proposed solutions. These experiments have led to deeper insights and improvements in our solutions.

We discuss the above topics in Chapter 4.

3.2.3 Formal documentation of systems

Good systems are developed around a clear set of principles and design invariants. Before code is written, a set of computational tasks is derived from the design; the tasks result from analysis and problem solving. They are then refined into further subtasks as the result of further problem solving. In the end, a collection of data structures and algorithms are written and assembled into a system. However, it is frequently the case that the task specifications, key problem solving insights, and the reasoning that connects them to the code are not written and preserved with the code. Even if they are preserved as separate electronic documents, they are not connected to the code in any systematic and guaranteed way. It is essentially impossible to recover these key elements of a system from the code itself.

For example, the code might need to keep a balanced tree. So a version of red/black trees is implemented. The documentation might mention a standard algorithm plus a trick used to make it suitable. See our account of red/black trees.

As code evolves, there is a large danger that it becomes disconnected from the ideas and reasoning steps that justified the original.

Formal specifications are intended to preserve at least the task requirements. They also provide an anchor point for some of the reasoning steps in the code refinement process. They provide points at which computer checked reasoning can be directed when the reasoning steps are discovered to be subtle or critical.

3.2.4 Verification and synthesis at the speed of design

Interactive theorem provers have become extensible. Tactic style provers are clear examples; those systems can easily incorporate new algorithmic advances as additional tactics. Thus new rewriting techniques and new decision procedures are rapidly incorporated into these provers. Also tactics naturally evolve into clusters that become ”supertactics.” As machine speeds increase and even the sequential provers can afford to execute more and more comprehensive tactics. The concurrent provers such as Nuprl and MetaPRL can afford to run very expensive supertactics in parallel with small ones.

The tactic style provers have provided a path for ever increasing theorem proving power and a route for advances in other communities, such as SAT solvers, model checkers, and fully automatic provers, to become components of large tactic-based provers. These advances have made it possible for talented users of interactive provers to imagine formalizing the reasoning of a programmer or designer at speeds close to the time it takes them to carefully write down their ideas as part of a documentation package.

However, the theorem provers cannot keep up with programmers at the level of basic factual knowledge about mathematics and data structures. Thus the prover becomes mired in having to prove many hundreds of facts that are immediately obvious to programmers. These facts are both of the very basic kind, say about numbers, lists, trees, and graphs. They are also about the particular problem domain, such as message structure, network connectivity, virtual synchrony, fault tolerance, etc.

We have experienced the situation where we could keep up with designers and programmers. It happened in the area of protocol verification after we had been working with the Cornell systems group for seven years building a common vocabulary and a common set of concepts. We eventually reached the stage where we had enough formalized knowledge that we could verify and idea in the same time it took to code it. We found mistakes both before and with the programmers. More significantly, we were able to generalized the solutions and provide cleaner code in the case where we participated from the beginning "at the speed of the designers."

If each prover had access to the total collection of basic facts used by others, then formal theorem proving would proceed at a pace very close to that of informal reasoning. In this case, it would be possible to prove all the critical reasoning steps in design refinement and store the proofs as part of the documentation. These proofs would make some of the documentation formal and connect it to the code. In Chapter 6 we examine how these reasoning steps are connected to the code in a compiler implementation. In the appendix, the paper by Constable [45] illustrates how comments are integrated into correct-by-construction code synthesis.

3.2.5 Fostering a richer culture of correctness

The people who build and use interactive theorem provers are very interested in correctness. They are trying to build systems that eliminate all errors in logical and mathematical reasoning. They are concerned that their systems are built correctly and tend not to trust other systems.

This creates a certain culture in which people want to "double check" results that they are interested in by proving them in their own systems. Such an attitude tends to create islands of isolated but very pristine formal mathematics. It also leads to practices that do not scale, the practice of "doing it all over again in system X." The extreme end of this is reproving in each system that 1+1 = 2. The FDL is a component that can change the culture to make it more like the well proven culture of rigorous but not completely formal mathematics.

What we really care about is a large body of mathematical results in which we have great confidence and which can be used in software construction. The top level then are the theorems, the facts, stated in the context of a particular theory, such a type theory or set theory. Below that is the level of the rules and logics used to establish these facts. These logics might differ a great deal, but the details are generally not interesting to the practitioners of software construction; the facts and the theory suffice. Below the level of the logics is the level of the deductive systems (or proof engines) that implement the rules and proofs. This is a very low level of abstraction that is not of interest even to most mathematicians, but it is the level at which the theorem prover communities are interacting. The FDL aims to raise this level to that of the theorems.

3.3 Integrating an FDL into Verification Technology

We have built a prototype FDL; it was designed to provide the capabilities needed for critical infrastructure protection according the role we outlined in the previous sections. It already includes formalized mathematics from four systems, MetaPRL, Nuprl, JProver, and PVS, and we plan to add more as we learn how to do that well. We have already learned a great deal from the year of experiments conducted with the experimental FDL.

A manual for the FDL is available at the project web site, both under Publications and under FDL Prototype. In addition there are extensive Design Notes available at the web site which include use scenarios. In this section we relate the FDL capabilities to the discussion of the previous sections.

The FDL can be accessed through VNC (see the FDL Prototype link). In addition we provide a service that posts mathematical content to the Web and posts key formal metadata that is harvested from the FDL. This service is discussed below.

3.3.1 Basic architecture

One of the most critical roles of the FDL in software infrastructure protection arises from its capability of connecting formal knowledge produced with theorem provers to intuitive knowledge produced and needed by computer scientists, mathematicians, system architects, and programmers.

Data formats

The FDL must contain formal material from a number of provers. This requires a formal notation that provides a uniform data format for storing definitions, theorems and proofs internally and that it accepts data in a variety of external formats. We accept data in XML and in ASCII. The internal format we call terms.

The FDL manual provides a description of the data types that support content and metadata. We only list some of the features to convey the basic concepts.

1. term structure

2. abstract object identifiers

3. binding structure

4. connection API's

Accounting mechanisms

Another essential feature of the FDL is that it supports mechanisms that account for the soundness of results. The results might be from a single version of a system, from multiple versions of a system, from several systems implementing the same logic, from two logics, from a logic and a programming language, from a logic and a test suite of examples, from a logic and corresponding informal language and so forth. We have described the use of certificates and sentinels to provide broad and robust accounting mechanisms.

Library services

Our proposal listed many services that an FDL should support. Many of them are routine, such as

1. Store a result
2. Retrieve a result
3. Display a result
4. Extract code from a theorem
5. Display logical dependencies
6. Combine closed maps
7. Display use links
8. Cluster
9. Search

We present some of the advanced services below.

Presentation services

The FDL should provide extensive links into the web since it will be hold the broader network of knowledge that provides the context for many items in the library. Ideally many FDL services will be available from the Web using large compute servers to drive the computationally intensive tasks.

The work required to obtain a flexible and readable presentation of mathematics in HTML and MathML turned out to be exceptional. One of the technical difficulties is reported in a short note by Allen and Hickey on how to indent mathematical text in a way usable by the standard browsers.

Here are specimens of text from Nuprl and PVS that illustrate what we have been able to accomplish. Appendix A contains more.

Advanced library services

The proposal lists a number of advanced services that we will explore, some of them are listed in the Goals section of the Introduction. Some are listed in the FDL manual as well. Here we simply offer the flavor of advanced operations.

Google provides a translation service that will produce a rough translation of articles in foreign languages into English. We are experimenting with a service that translates statements among the logics supported in the FDL.

It is possible to attempt to move a theorem from on theory into another by attempted to "replay it" in a new context. This is much like translating a bush or plant from one garden to another. It must be possible to collect up all the main roots (definitions, axioms, rules, lemmas) and replant them in a new context. We are experimenting with this service.

3.3.2 Integration with verification technology

Here are some of the elements of verification technology that are supported by the existing experimental FDL.

Collecting formal semantic theories and mathematical models

We provide sample documents on formal semantics among the booklets stored in the Nuprl content section. We have included a complete account of the formal semantics for distributed systems; this is an ongoing effort. We also have the capability of including a formal semantics for a large subset of the OCaml programming language, but that work was deemed to be too Nuprl centric for this effort at this point.

Our distributed system semantics includes formal models of IO Automata [113] and Message Automata [29]. We are likely to include Security Automata [166, 60] among these models. When we import more Isabelle HOL, we will also have a JVM model.

Formal semantics of types and specification languages

The FDL provides the semantics of an extremely rich type system, Computational Type Theory (CTT). We connect this formal knowledge to an extensive intuitive semantics of the primitive concepts.

We also provide a semantics for the Simple Theory of Types which is a very important logic used by HOL and PVS. Our semantics is complete for HOL, and in principle it might be made complete for PVS, but that is not in the scope of this work.

Verified programming tools

The MetaPRL library includes components of a formal compiler which illustrates a means for producing a verified and formally documented compiler.

Formal classes

We have experience with classes that also use logical expressions as part of their interface. We call these *formal classes,* and we learned in our distributed verification work that they help classes "snap together" more tightly. We are storing formal classes in sections of the library and will make them part of our reference distributed system.

Verified reference algorithms and protocols

Systems consist of many algorithms; typically only a few are critical and even fewer are novel. Nevertheless, we think it is valuable to include examples of verified algorithms in the library. We have provided access to over a hundred algorithms so far which are common in systems. We have also included a few algorithms whose verification is challenging, such as red/black trees. We will add more algorithms of this class, such as Dijkstra's shortest path algorithm and an algorithm for Higman's lemma from HOL.

Examples of shared mathematics and translation services

We have experimented with verifications that use results from more than one prover. One method of sharing involves translating from one theory into another. We have experimented with this method and are encouraged enough to continue this line of investigation.

3.3.3 Basis for advanced technologies

The FDL provides the basis for several advanced capabilities. It is an example of a database that contains several computational symbolic theories. These can be used to support investigations in several areas of science in which the premises and assumptions can be expressed mathematically.

The FDL is very concerned about the links between formal and informal mathematical texts. Having a large amount of content will allow people to explore automated ways of producing the links and will even suggest ways to help people formalize mathematics.

The FDL will also provide a repository of material that can be the target of methods of translating formal mathematics into natural language. We have experimented with this task and know that it can be done well.

Once the FDL has collected a very large number of theorems, perhaps 30K or so, it will become a magnet for data mining efforts. This may become a boon to mathematics education and research in unforeseen ways.

The use of formal theories based on abstract object identifiers will provide an attractive alternative to file system management.

An FDL will also be a good basis for a multisystem version control system. We discuss future work on the FDL in Chapter 8.

Chapter 4

Working notes on Formal Digital Library (FDL) Design

These are working notes[1] explaining the purposes and design of Formal Digital Libraries. The paper Abstract Identifiers and Textual Reference[2] [8], based on these notes, elaborates on our basic concepts and methods for managing inter-referential digitally stored texts with the special aim of accounting for logical dependencies upon which meaningfulness and correctness of these "formal" texts depends. We are also especially concerned with safe sharing of formal texts among diverse clients who may not fully agree on what constitutes meaningful and correct material. These working notes are considerably more detailed than the paper mentioned above, and cover concepts not discussed in the paper.
[**Boldfaced** terms outside of headings indicate terms to be found in the Glossary of FDL Terminology (Chapter 10).]

4.1 Content vs Infrastructure.

The Cornell effort toward developing digital libraries of algorithmic mathematics is explicitly bifurcated. We aim at generation of constructive math by means we have long developed and employed. But, and this is the topic of these notes, we also aim to design a repository system, a "Formal Digital Library" (FDL) that supports content that is radically different from ours,

[1] the notes are maintained at http://www.nuprl.org/FDLProject/FDLnotes
[2] http://www.nuprl.org/documents/Allen/abids_n_ref.html

and perhaps even incompatible. Our intention is that our contributions toward content and content creation are but some of many possibilities, and we strive to avoid bias towards our content and methods. Our established status as providers of mathematics in a minority (though widely known) genre has made us sensitive to the possibilities of exclusion. Our suspicion that our content would be excluded from libraries of mathematics designed by those whose experience and loyalties lie with more widely employed genres has caused us to appreciate the need for a more radical inclusiveness. It is not our desire to use the general system design as a means to unfairly promote our mathematical preferences. The FDL design is intended to be neutral, and our content must vie for consideration on its own merits.

This means that we are not obliged to create methods for content creation generally, but must flexibly anticipate likely methods, including extant ones, and publish adequate methods to access and contribute to FDL collections. Because of the variety of semantical intentions and epistemic assumptions, sometimes incompatible, underlying different developments, flexible and stringent accounting methods must be introduced if these mixtures of content are to coexist.

What we offer are to-some-extent common syntactic structures, storage, and accounting services, especially accounting for formal facts contributing to arguments for validity of proofs. See the glossary entries for **Text**, **Certificate** and **Proof**.

Further, we shall *not* require a particular logic (even a particular form of assertion), type theory, tactics, or the use of the ML programming language. However, it should be noted that our experience with these has led us to reject a number of presumptions about methods of proof and expression and left us with an expectation of rather liberal practices. In particular, we have become accustomed to using a language in which, as in *informal* mathematical practice, the sensibleness of expressions is not always obvious, and must sometimes be demonstrated, and in which some other simplifying assumptions about notational adequacy are avoided (such as assuming there is a single domain of all values). Our use of tactics to formalize the notion of effectively explaining how to make inferences, rather than restricting our expression of inference to schematic forms, has forced us to deal with issues pertaining to accounting for reliability of programs. Briefly, we have already committed ourselves to, and are experienced with, sophisticated methods of applied logic, and are therefore more likely than we would otherwise be to anticipate a great variety of formal methods.

A major purpose for development of our newest system is to analyze many concerns about how to simultaneously support independent developments. Many of these concerns were driven by centrifugal forces within our own project, whose members represent a variety of different opinions about how to formulate and develop formal mathematics, sometimes differing even as to what is legitimate. Our desire despite this to share what we can in various ways has driven us to a flexible design.

4.2 Formal Artifacts and Informal Understanding.

By the "formal" we mean that which has precise meaning or objective, ideally computer verifiable, criteria of correctness, based upon the "syntactic" form of the content. By the informal we simply mean everything else. An informal practice or artifact may have formal components.

We consider it essential to embed formal **Texts** (proofs, programs, definitions, propositions) in a body of informal **Texts**, recognizing that the use of the formal material is in any case embedded in informal practices.

If the formal artifacts are completely separated one will not understand how to apply them. Often these artifacts are formalized counterparts to informal ones. Some benefits of formality are that analysis can be pursued to great detail, and that combinations and transformations of the formal artifacts can often be performed mechanically without need for human intervention in order to make sure the particular intermediate stages of manipulation are meaningful and correct; systematic and abstract arguments about formal expression can be made reliably (sometimes such argument can itself be made formally by reflective devices).

Formalization should be introduced where it is economical to do so. A smooth flow and intricate interweaving between the formal and the informal is key to correctly deploying formal artifacts.

Complementary to introducing formal artifacts in place of informal material is attaching informal material to formal artifacts in order to better expose them to human understanding, much of which is not in fact formalized. It should be further noted that informal material can be formalized in various ways, and that the same formal material can be variously organized informally. Thus many interwoven organizations are natural. See Words vs

Formality and see Readings (section 4.2.3).

4.2.1 Words and Images vs Formality

We believe that the choice of which names we attach to concepts or objects, and how expressions are presented to the senses are not normally pertinent to the **Formal** significance or correctness of formal artifacts (but see Bates's Point (section 4.2.2)). We expect uniform renaming or alteration of notation to leave formal correctness intact.

On the other hand, our own abilities to understand formal material are often deeply tied to our habits of notation and nomenclature. Indeed they are so important that we must neither ignore these matters nor presume to be able to formalize them. Such habits may vary from person to person, from circumstance to circumstance, and from time to time. Often there is dispute over such choices of naming and notation, which need not impinge on the formalization process.

The use of abstract syntactic structure is widely understood to isolate structural features of notation pertinent to "formal" correctness. Of course there is a sense in which to write "3-4" to mean the result of subtracting three from four (instead of four from three), or to mean the sum instead of the difference, is wrong, but that is not a matter of formal correctness as intended here. When we employ formal notation in an informal context, such errors can be ruinous, and so we must have ways of avoiding them, but we take this to be a different issue from formal correctness.

Our project conceives formal structure, therefore, to be abstract. In particular we take the expressions or **Texts** to be essentially like parse trees of a very simple grammar, and deem it to be a separate matter how they are viewed or created or edited by humans.

Similarly, though written natural language is essential and should be included among, and intimately interwoven with, formal expression, we consider the meaning and syntax of natural language as a whole beyond the purview of the FDL project *per se*. As researchers succeed at formalizing parts of natural language, those parts may come to occupy more of the formal domain as those formal methods are introduced, but we must not depend upon that eventuality or put off the incorporation of informal language as mere uninterpreted text.

The independence of formal structure from notational appearance, epitomized by the use of abstract syntax trees instead of strings, eliminates much notational dispute from the formulation of formal expressions. But there remain the problems of resolving disputes and informal errors involving the names appearing as atomic components of formal structures. See Formal vs Informal (section 4.2), Readings (section 4.2.3), Naming Problems (section 4.4.6), and Abstract Ids & Closed Maps (section 4.4.1).

4.2.2 Bates's Point: Words could matter to programs too, though.

Even if appearance of expressions and naming are not normally considered essential to formal correctness, this, as pointed out to us by Joe Bates, does *not* mean they could matter only to humans or be purely informal. It could well be that some heuristics for performing inference could indeed make use of how expressions look and how things are named. Still, such data can be supplied to these programs without building them into the formal material directly. Indeed, we intend for specifications of how to display expressions and how to denominate entities to be themselves expressed as texts to be included as objects in the FDL, and they would therefore be made available to such proof heuristics as needed.

4.2.3 Readings

The fundamental relations between **Formal** objects that are accounted for in the FDL are "internal" ones. For example: the fact that one proof cites another as a lemma; that a certain definition for an operator is employed in a proof or procedure; that a certain inference rule is used by a certain inference engine; that a certain object contains the source code for a given procedure; that a certain object's content is used as data in, or is the result of, a certain computation; that one object refers to another.

But persons and programs normally need much more information about the organization of formal data (see Formal vs Informal (section 4.2)), "external" relations between objects, and connections to informal concepts. By a "reading" we mean a collection of objects and procedures that provides a guide to a collection of formal objects. Naturally, a reading is itself stored in the FDL. Readings provide alternatives to simply following internal links among formal objects, and would typically include: pages of text mentioning formal data, hyperlinks to formal objects, to other objects of the reading, and to material beyond the FDL collection; search utilities for focusing on the subject material of the reading; objects specifying how to display expressions to the reader.

Multiple readings may be given for the same material. When content is developed as a formalization of a conventional body of knowledge, one should provide a reading of the material using conventional organization. Other organization is possible as well. For example, presentation and organization for the purposes of programmers looking for graph algorithms may require an emphasis that differs somewhat from presentation for graph theorists. Formalization itself can reveal aspects of material that suggest other organizations as well; persons that formalize material may have something significant to contribute to the explanation and organization of conventional subjects. One might also tailor presentation to the sophistication of the audience.

As an adjunct to readings, one may provide data and utilities for the use of programs employed by those at whom the readings are directed, including as part of the reading an explanation of how they are to be used.

An example of an external organization would be presenting a collection of definitions, proofs and programs as being about abstract algebra, say, based upon a given body of assumptions, and perhaps grouping parts in "modules" sharing common assumptions and methods. This grouping need not be built in to the formal objects themselves since, after all, their correctness need not depend on such grouping. One may well find it efficient or informative to take a few pieces from variously organized collections of material and recombine them into a new reading.

4.2.4 Concise Informal Annotations - titles, paraphrases, domains, roles

Consider the related problems of finding **Formal** content from an informal starting point, and giving informal paraphrases of formal content. These are problems of building bridges between the formal and the informal with the right endpoints.

If the only utility for search were based on formal content, one utility which is indeed essential to using a formal library or archives, then searching for formal content from an informal starting point would be constrained to judging (often guessing) what formal expressions were likely used for informal concepts. As valuable as this utility would be, it could not be considered adequate.

The inclusion of "Readings" (section 4.2.3) in the FDL will provide explanatory **Texts** that consist largely of words that have embedded in them references to formal objects they are about. So by employing a content search on words and phrases we could find these explanatory texts, and by reading them ascertain the relevant formal objects. This too is a valuable capacity, but there are three notable inefficiencies in this method: reading a relatively large text that may discuss and relate several formal objects in order to discern which are relevant to the search is not automated; if one simply assumes that all the formal objects may be relevant in order to apply automation instead of reading, this is inefficient in a different way because one must often expect the assumption to be wrong; and lastly, the production of explanatory material of the sort we imagine as part of readings is itself rather expensive, and is not likely always to "reach" all the formal content reliably.

These problems would be mitigated by concise informal annotations of formal objects. Each "concise" annotation would be about one object, and would consist of informal words or phrases. This addresses each of the three inefficiencies mentioned above. Concise annotation focuses the search and "reaches" more formal material since it's cheaper and less distracting to create than discourse.

Two typical kinds of concise annotations are titles and paraphrases. Often theorems, concepts, or programs have conventional titles which should therefore be attached to their formalizations. For example "Ramsey's Theorem," "Dedekind Infinite," and "Dijkstra's Shortest Path Algorithm." The paraphrases used as concise annotations would not normally be "read off" the formal expression, but rather would complement the formal detail. For example, one might paraphrase

$$\forall k{:}\mathbb{N}^{+},\ r_1,r_2{:}\mathbb{N}_k,\ q_1,q_2{:}\mathbb{Z}.\ q_1{\cdot}k{+}r_1 = q_2{\cdot}k{+}r_2 \Rightarrow q_1 = q_2\ \&\ r_1 = r_2$$

as "Integer division is unique," or paraphrase

$$\forall as,bs,cs{:}T\ \text{List}.\ ((as\ @\ bs)\ @\ cs) = (as\ @\ (bs\ @\ cs))$$

as "List catenation is associative" or "Appending lists is associative." Some theorems may be unworthy of paraphrase such as

$$\forall n{:}\mathbb{Z},\ as{:}A\ \text{List}(n).\ \|as\| = n \in \mathbb{Z}$$

whose obvious paraphrase, “Lists of a given length have that length,” is silly. If you want to find a lemma like this, it would be because you were looking for theorems relating the formal expressions “$\cdot$ List($\cdot$)” and “$\|\cdot\|$” and you would already be in the formal domain.

Amanda Holland-Minkley[3] advises us that these annotations, and others, useful to search are also useful for proof paraphrasing. The reason is that while one may develop methods for paraphrasing proof structures principally by analyzing the structure of the proof and recognizing forms that can be reorganized into conventional verbal forms, one gets down to a level where the proper paraphrase is simply not derivable from the internal structure. When a proof is paraphrased one may want to cite a lemma by conventional title or to paraphrase its content in a way that is not easily determined by its formal structure. Further annotations that are concise and useful for search and paraphrasing are indications of intended domain, and the role in the larger body of material, such as whether a theorem is intended simply as a lemma for citation by a particular proof rather than of broader interest, or whether it is simply a technical device for facilitating formal proof rather than having a more general cognitive significance.

4.3 Formal Digital Libraries

A Formal Digital Library (FDL) is intended to serve as a repository with evidential aspects, a repository for knowledge as opposed to mere information. It is also essential to recognize that FDLs will be expected to cooperate in ways that do not ruin their epistemic value.

4.3.1 Logical Libraries

Our conception for **Formal** Digital Libraries involves some functions of ordinary libraries, but is extended to accounting methods for logical relations between documents. Ordinary objects, of varying degrees of structure from informal **Text** to **Formal Proof**, are the main content, but the collection also includes documents that serve as certificates of facts established by the processes maintaining the collection. Among the kinds of facts certified by the FDL process are that the ordinary objects it stores were acquired from

[3] http://www.cs.cornell.edu/Info/People/hollandm

specific sources or built by specific means (archival functions). These certifications are represented as certificate objects in the FDL. See Certificates (section 4.5.1).

Our project is especially interested in the certification of formal proofs according to specific (user suppliable) criteria, and when stressing this function, when thinking of the FDL services as principally directed at supporting this function, we call it a "logical library." A particularly esteemed feature a certificate may exhibit is its making an objective and independently verifiable claim; this is the ideal for "logical" claims, and is the alternative to authority. Of course, one might view this as the essence of scientific claims generally, but the procedures for verifying "logical" claims are largely reducible to automatic methods which are in principle directly verifiable by an automated text management process since it can itself in principle perform, and so certify, the computations.

One aspect of ordinary libraries we emulate is theoretical neutrality. Thus, we do not require that ordinary documents must be "correct," indeed there is no privileged criterion for correctness of ordinary texts. Criteria for correctness (via proof engines, eg) are supplied by the users of the FDL as contributions to the text collection.

Certificates in the FDL, however, are guaranteed to be correct according to published criteria; the criteria for certificate correctness must therefore be understandable to users of the "logical library." The correctness of some certificates will be independently verifiable, the ideal for logical claims, but other certificates may not be practically verifiable because they attest to the existence of something that is not made practically accessible (an example being a certificate meaning that there once was a proof of a given formula in the collection).

When a process, say with a user behind it, accesses a collection, the "view" is a sub-collection of objects we call the **Current Closed Map** served by the FDL and subjected to transformation, perhaps with further copying from the FDL, and is usually stored back to the FDL (see Abstract Ids & Closed Maps (section 4.4.1)). A subtlety is that since **Certificates** are created only by the FDL process itself, when certificates are to be created in the current closed map, they must be created by the FDL, then added to the current closed map. And since certificates usually refer to objects they are about, those objects must also be in the collection. So one is pretty tightly coupled with an FDL when developing certified content. However, as long as one is only reading from the collection, the coupling can be loose.

This leads us to the matter of Multiple FDLs. Also see FDL Functions (section 4.3.3) for further discussion.

4.3.2 Multiple FDLs

As was pointed out in Logical Libraries, development of certified content (as opposed to mere use of extant content), requires the use of the FDL process since the **Certificates** are "owned" by the FDL.

It would not long be tolerable, however, for multiple developers to have to yoke themselves together to a single process. After all, different parties will want to work at least temporarily in isolation, or may want to maintain independent FDLs. Here is one scenario. Periodically a user connects to a large shared FDL and downloads parts to be developed onto a laptop computer. Later they want to contribute the developed content back to the large FDL. Here is another. For purposes of reliability or speed, an entire FDL is copied, then both the original and the copy are developed, and eventually both parties want to combine their content. A combination of these two: a user wants to draw material from two divergent FDLs that have not (yet) been combined, and do some local development, eventually making it available for general use.

In all these scenarios the situation is one in which there is a distributed repository, namely all the FDL repositories one can connect to, which is relative to time and circumstance. But these repositories are still distinct because each repository is responsible for its own certificate objects and cannot be responsible for the others'. This is rather like a real library system (ie, one with real books) made of individual libraries that control their own premises, but cooperate as providers of content. The libraries cooperate but are independent; this is fortunate because sometimes a library burns down or loses books or mismanages its records. It is the mismanagement of certificates that is of particular concern to our ambitions for "logical" libraries.

The bottom line is that the certificates in the source FDL cannot be simply copied as certificates into another FDL. What *is* possible is for an FDL to create its own certificate containing the foreign certificate's content, and indicating the foreign source of that content. Let us call this kind of certificate a "borrowed" certificate. Note that the new "borrowed" certificate will have different meaning from the original, although the content of the foreign original is extractable from it. Parties that trust the source of the certificate can use the borrowed certificates in lieu of original local certificates, and parties that don't can ignore them or attempt to have fresh local certificates built with the same content as the foreign certificate. To summarize, a borrowed certificate is a certificate attesting to the fact that the content was taken from another specific supposed FDL process, and as such, whatever certification policies appear to be indicated by the content, and are *purported* to have been enforced by the foreign FDL process, are *not* adopted by the borrowing FDL.

The local re-establishment of borrowed certificates, ie the attempt to create equivalent native certificates, might be a good use of spare cycles in the background; here's a place where the "objective" certificates mentioned in Logical Libraries (section 4.3.1) pay off. On the other hand, managers of several large FDLs may aspire to collective trustworthiness, going to great pains to assure users of reliability of communication with them and of their faithful implementation of certificates, and simply "borrow" each other's claims without locally recertifying them except upon explicit demand. See Borrowed Certificates (section 4.5.7) for elaboration.

4.3.3 Library, Archival, and Workspace Functions

Continuing the discussion of Logical Libraries (section 4.3.1), we may regard services or functions of the FDL we envision as roughly falling into three overlapping categories: (1) Library functions focussed on maintenance of collections of items containing information typically thought to be of some general value; (2) Archival functions directed at maintaining integrity of interrelated collections of items, many of which are records expressing facts verified by the FDL process; (3) Workspace functions providing utilities for the preparation and development of items suitable for collection in the FDL.

The intention that the FDL be oriented towards the maintenance of **Formal** materials, particularly charged with maintaining verified relations between formal documents such as logical relations between digital documents and artifacts, compels us to maintain records in the manner of archives (in the sense of archival science). The point of formal entities is that there are explicit verifiable criteria for claims about them and relating them, and much of their value depends on establishing and recording the validation of such claims; these are essentially archival functions.

A library of **Formal** documents maintained in isolation without regard to their relations between each other or without regard to whether the verifiable relations claimed for them actually obtain, is considerably less useful than one which maintains records of these verifications. Thus, we would consider the archival functions to be essential to any Formal Digital Library. And complementarily, we consider the inclusion of informal material essential to any large body of formal material to make it widely and repeatedly useful (see Formal vs Informal (section 4.2)).

Workspace is a different matter, though. Functions supporting the development of material for eventual submission to an FDL are not essential for the use of the FDL as a repository of information and knowledge. Indeed, the needs of a person or organization or community trying to develop material may be significantly greater than those who simply need to access it, and implementation of a workspace facilitating experimental development and collaboration may go significantly beyond effective implementation of a digital repository. Thus, we would not expect every FDL to effectively support development.

However, because of the record-keeping functions of the FDL, effective development of material that will meet the certification requirements implicit in the useful submission to an FDL, effective development of formal material to be submitted requires that the material be developed incorporating the same record-keeping devices as are needed in the target repository. Archival scientists have articulated essentially the same principle by holding that the original record creating institution needs to adopt record creation methods that anticipate archiving.

This suggests that although not every FDL needs to efficiently implement workspace functions, it is important that appropriate workspace processes be implemented. Our FDL design includes workspace functionality, since the same basic accounting devices should be used for the repository and development. Different FDLs may be maintained and differently implemented with varying emphases on facilitating workspace functions.

One institution might implement an FDL principally for "publication" of formal material with little support for development. Perhaps this FDL provides utilities to expedite search and establishes policies for long retention, but does not allow modification of submitted material, or provide development oriented facilities such as multiplexing **Inference Engines** for heuristic proof methods. Another organization may implement relatively low storage capacity FDLs aimed more at development of material to be submitted later to other FDLs, and may emphasize version control and flexible methods for collaboration between developers.

Finally, it should be noted that the design of "finding aids" is not an explicit part of the FDL design, and yet without such facilities for finding content in a repository one can hardly consider the collection to be a library or an archives. Finding aids are essential, but we consider them to be content themselves; methods of organizing and finding content are themselves contributions that can be accessed, innovated and improved upon.

4.4 Repository Data

Identifiers treated abstractly, embedded in texts and used as pointers to repository objects, form the basic data upon which reference and accounting methods are based.

4.4.1 Abstract Identifiers and Closed Maps

The abstract use of identifiers for FDL object "names" and as components of expressions within the text collection is basic to our methods for managing the collection, especially as regards correctness. The intention is that the user should treat names or identifiers abstractly, simply taking them to be discrete and atomic. They fill formal roles normally filled by simple names (identifiers), but have no informal value, in particular no mnemonic value or value as conventional nomenclature. See Abstract Identifiers (how) (section 4.4.4). This is our technique for avoiding disputes or errors concerning names in the formal content, and how we facilitate essentially arbitrary combination of objects, avoiding name collision; abstractness of names entails their uniform replaceability.

We will discuss a variety of issues concerning abstract identifiers, but here we give an overview of relevant facts about our intended system without elaborate motivation.

By a "closed map" we mean a function of type $D \rightarrow \text{Text}(D)$ where D is a finite discrete type of values. The type $\text{Text}(D)$ is the class of expressions where the values of D are "abid." Here we must digress. Abstractly, for many purposes of reference and accounting, $\text{Text}(D)$ could be any kind of data for which it is understood what counts as "occurrences" of D-values within the text. However, the **Text** structure we adopt is a simple recursive type of iterated operators on subtexts because the subexpression relation is dominant in typical computations on expressions in precise notations. In addition to its subtexts, a text contains a sequence of labeled values presupposed by the construction of texts, which we call here "pro-textual" values. When we define the class of texts, the kind of pro-textual values that can occur with a given label is determined by the label, and is stipulated when that class of pro-textual values is introduced. Text(D) is the type of texts in which the possible pro-textual value constituents of form "x:abid" are those such that $x \in D$. We identify objects in a closed map with indices. See Pro-textual Constituents (section 4.4.5).

In practice the class D of object indices 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. If the restriction of a closed map $f \in A \rightarrow \text{Text}(A)$ to a subclass $X \subseteq A$ is in $X \rightarrow \text{Text}(X)$, and so is itself a closed map, then we call the restriction a "submap" of f.

Two closed maps $f \in A \rightarrow \text{Text}(A)$ and $g \in B \rightarrow \text{Text}(B)$ are "equivalent" when they are simply "renamings" of each other, i.e., when there is a one-one correspondence between A and B such that for corresponding $a \in A$ and $b \in B$, $f(a)$ and $g(b)$ are identically structured modulo matching abid occurrences that correspond. See Closed Map Operations (section 4.4.2). The abstract treatment of object indices entails that whatever criteria of correctness hold of one closed map hold also of equivalent closed maps.

Dependency management between objects in a closed map $f \in D \rightarrow \text{Text}(D)$ is based upon an explicit criterion of object reference: an expression $t \in \text{Text}(D)$ refers "directly" to object (index) $x \in D$ just when x:abid is a pro-textual constituent of the text t (this includes any subtexts of t). An expression refers, perhaps indirectly, (wrt f) to $x \in D$ just when either it refers directly to x or else it refers to some object $y \in D$ where $f(y)$ refers to x. The basic model of working with **Closed Maps** is to maintain a "current closed map" (section 4.5.2) as a part of state that is updated repeatedly as one works.

The **FDL** 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. One engages the **FDL** in a **Session** to help build and manipulate closed maps and also to store them for later retrieval (modulo closed map equivalence).

While there are useful notions of dependency between objects that may arise during a session as part of state outside the current closed map, the dependencies of enduring value shall be formulated purely in terms of closed maps (and treat indices abstractly).

Relatedly, while there are useful notions of correctness that can be defined with respect to state, the enduring ones shall be formulated in terms of closed maps alone. Further, while one can make good use of criteria of correctness of a proper submap of a closed map that depends upon the full closed map (say involving a search of the full closed map), we shall attach greater enduring value to those criteria for correctness of a closed map that depend only on that submap alone. Such criteria are preserved by monotonic extension of the closed map to a supermap.

Sometimes our criteria for correctness will depend on how programs (for example, **Tactics**) execute. Observe that when identifiers are treated abstractly by the computation system, they cannot be "secretly" computed. They must be provided as data to the computation either directly or via the current closed map. Further, since object indices are simple rather than complex, computation of object indices cannot be hidden by runtime combination.

Had we defined closed maps concretely as functions of type $\mathbb{N} \rightarrow \text{Text}(\mathbb{N})$ or $String \rightarrow \text{Text}(String)$ then program execution could create references to objects without its being apparent from the program code. Similarly, had we defined the closed maps as (A List)$\rightarrow$Text(A), even leaving the basic identifiers abstract but indexing objects by complexes, again one could "hide" the references to objects in the execution.

4.4.2 Operations on Closed Maps

Here we describe some basic operations on closed maps. Recall that a closed map is a function of type $D \rightarrow \text{Text}(D)$ for some finite discrete type D. See Conservation and Destruction (section 4.4.3) for connotations of + and − prefixes used below.

+ Uniform Renaming. Let the function $r* \in \text{Text}(D) \rightarrow \text{Text}(X)$, for $r \in D \rightarrow X$, replace each abstract id constituent i:abid (for $i \in D$) by $r(i)$:abid throughout the text. A renaming of a whole closed map $f \in D \rightarrow \text{Text}(D)$ is a closed map $((r*) \circ f \circ r^-) \in X \rightarrow \text{Text}(X)$ for inverse functions $r \in D \rightarrow X$ and $r^- \in X \rightarrow D$. Two closed maps $f \in D \rightarrow \text{Text}(D)$ and $g \in X \rightarrow \text{Text}(X)$ are "equivalent" when they are renamings of each other.

+ Contracting. If $X \subseteq D$ then "contracting" a closed map $f \in D \rightarrow \text{Text}(D)$ around X is restricting f to X together with objects referred to by objects in X (ie, the smallest $A \subseteq D$ such that $X \subseteq A$ and $f(i) \in \text{Text}(A)$ for all $i \in A$). Hence the contraction is the smallest submap including X. So, contracting a closed map around some of its objects (indices) X is discarding all objects except those among or referred to by the objects of X.

+ **Focusing.** If $X \subseteq D$ then "focusing" a closed map $f \in D {\to} \text{Text}(D)$ on X is "contracting" f around X together with objects that refer to members of X. So focusing a closed map on some of its objects X is restricting it to the objects relevant to X.

+ **Deleting.** If $X \subseteq D$ then "deleting X" from a closed map $f \in D {\to} \text{Text}(D)$ is finding the largest submap of f excluding X from its indices. That is, the deletion of some objects (indices) X from a closed map is gotten by removing X along with all objects whose contents refer to any of X.

– **Reassigning.** A reassignment of a closed map is a closed map with the same indices but perhaps different contents for some or all the indices.

+ **Zipping Merge.** This is a way of merging two closed maps with a stipulation that certain objects be identified. Zipping can fail.

Assume $f \in A {\to} \text{Text}(A)$, $g \in B {\to} \text{Text}(B)$, that A and B are disjoint, and Z is a collection of pairs in $A{\times}B$, and that Z is one-to-one. If $t \in \text{Text}(A)$ and $s \in \text{Text}(B)$ then let us say they "match" just when they have identical structure except for abstract id occurrences within them; wherever an abstract id occurs in one, then a possibly different id must occur in the other. If $t \in \text{Text}(A)$ and $s \in \text{Text}(B)$ match then let IdPairs(t,s) be the collection of pairs $\langle x, y\rangle \in A{\times}B$ such that x occurs in t where y occurs in s.

Given a collection Z of pairs in $A{\times}B$ let Z' be the smallest one-to-one extension of Z such that if $\langle x, y\rangle$ is in Z' then $f(x)$ and $g(y)$ match and IdPairs($f(x)$,$g(y)$) is a subcollection of Z'. A procedure for generating Z' from Z is to iteratively compare contents of corresponding indices, adding the IdPairs(?,?) of the texts if they do match, or *failing* when either the texts don't match or adding them ruins the one-to-oneness.

To zip g into f along Z, determine Z' as above, then rename g to g' by replacing each $y \in B$ by $x \in A$ when $\langle x, y\rangle$ is in Z', then take the union of closed maps f and g' (recall that A and B are disjoint). We call the Z the zipper-start. Then f is a submap of the zipping, and g is a renaming of a submap of the zipping. The bias towards f may be useful when f is the current closed map of a **Session** involving some state beyond the current closed map that mentions objects in f that would impractical to rename in that state. Then zipping is a way of merging a new closed map into f. Of course, the merge of f into g (along the *inverse* of Z) is simply a renaming of the merge of g into f.

A variation on zipping merge would allow A and B to overlap as long as f and g agree on the intersection of A and B, which would have the same result as renaming g by some function r to avoid A, then adding pairs $\langle a, r(a)\rangle$ to the zipper-start. This corresponds to merging a partial variant into ones current closed map.

– **Folding.** Using a method for zipping much like in the Zipping Merge, we define a way of "folding" a closed map into itself by identifying (with each other) certain objects within it.

Assume $f \in A \rightarrow \text{Text}(A)$ and Z is a 2-place relation on A.

See Merging for definition of "matching" texts and IdPairs(t,s). Let Z' be the smallest symmetric-transitive extension of Z such that if $x(Z')y$ then $f(x)$ and $f(y)$ match and each pair of IdPairs($f(x)$,$g(y)$) satisfies Z'. A procedure for generating Z' from Z is to iteratively compare contents of already identified pairs of objects, failing when the texts don't match, or identifying the IdPairs(?,?) if they do match, then (at each step) taking the symmetric-transitive closure.

To fold f along Z, determine Z' as above, then choose a function $r \in A \rightarrow A$ such that $r(x) = r(y)$ if $x(Z')y$, and $r(x) = x$ if x is related to nothing by Z'. (Thus, r picks a canonical representative for each partial-equivalence class.) The folding of $f \in A \rightarrow \text{Text}(A)$ is $((r*) \circ f) \in X \rightarrow \text{Text}(X)$, where $X \subseteq A$ is the range of r over inputs from A.

Observe that folding a closed map need not result in a submap (modulo equivalence).

+ **Cloning.** Cloning a collection of objects is replicating them and replacing the references to originals within the clones by references to their clones. This is probably done for the purpose of subsequently modifying some of them. If A and B are originals with clones A' and B', and if B references A, then B' references A'.

Assume $f \in A \rightarrow \text{Text}(A)$ and $X \subseteq A$. To clone X, first choose a class B disjoint with A, and a bijection $r \in X \rightarrow \text{B}$ with inverse $r^- \in \text{B} \rightarrow X$; let $r' \in A \rightarrow \text{B}$ agree with r on X and be identity outside X. Then extend f from A to $r'*(f(r^-(b)))$ for $b \in B$.

+ **Splitting.** Splitting a closed map is cloning some objects along with all objects that refer to them, causing the closed map to branch into two equivalent closed maps, probably for the purpose of subsequently modifying a branch into a variant.

Not only is the original closed map a submap of the split, but deleting the cloned objects (and leaving the clones) would leave a closed map equivalent to the original.

4.4.3 Conservative and Destructive Operations

In the descriptions of Closed Map Operations, some are marked with +, some with -. The "+" indicates an operation that is considered conservative with respect to Certification (section 4.5.1) in that performing these operations on the current closed map leaves certificates intact. The "-" indicates that the operations are not likely to succeed on the current closed map without some more complex and destructive modifications, especially entailing deletion or modification of system generated "certification objects."

+ Uniform Renaming
+ Contracting
+ Focusing
+ Deleting
+ Zipping Merge
+ Cloning
+ Splitting
- Reassigning
- Folding

See Certificate Bias (section 4.5.3) and Current Closed Maps (section 4.5.2).

4.4.4 Abstract Identifiers - What are they really?

The notion of abstract identifier was introduced in Abstract Ids & Closed Maps (section 4.4.1). There it was explained that we mean them to be uniformly renamable, "unowned," atomic, and discrete.

Obviously the intent is to tightly restrict what operations may be performed upon them, but it has not been suggested how this can be accomplished or what values might be used for them.

If we were using a single programming language, we might borrow some method from it. For example, some programming languages admit the introduction of "abstract types" for which one can specify methods of creating new values and restrict the methods of access. Some programming languages have a concept of "pointer value" which is extremely similar to what we have in mind for abstract identifiers as object indices. Even in programming languages with no built-in methods for creating such values, programming practices are adopted for treating some values as abstract – different executions of the program may generate different particular values, but the result may be the "same" (modulo which values get picked for these "abstract" types).

Using these programming methods one does not so much implement abstract values as one abstractly implements values. When one gets down to execution, one uses concrete values, but they are used with a discipline that makes the choice of values irrelevant to the desired effect.

But we do not intend to embed the FDL in a particular programming language environment. When an external process connects to the FDL, some concrete, externally understandable, values must be communicated instead. We cannot directly enforce a programming discipline upon external processes to have them treat identifiers abstractly; but we can stipulate that the choice of concrete values communicated by the FDL during a **Session** is undetermined prior to that session. In particular, if one stores a closed map, see Abstract Ids & Closed Maps (section 4.4.1), in one session and then attempts to retrieve it in a later session, there is no guarantee that the same closed map will be retrieved. The only guarantee is that it is an equivalent closed map, i.e. same modulo uniform change of values chosen for the "abstract" identifiers.

The freedom to uniformly rename identifiers permits various useful operations on closed maps, especially as regards combining independently or partially independently developed closed maps, maintenance of multiple versions, and pursuing alternative lines of development off pre-existing closed maps. The threat of uniform renaming encourages users to employ well known methods for treating these identifiers from the FDL abstractly, a discipline which cannot nowadays be considered onerous. If source code is stored in the FDL, then tests can be specified by **Clients** and applied to the source code to certify that it does treat the FDL identifiers abstractly.

4.4.5 Pro-textual Constituent Values

As explained in Abstract Ids & Closed Maps (section 4.4.1) the method of inserting primitive or external values into our expression syntax is by means of components of texts of the form "<value>:<kind>". Examples of pro-textual values we inject are natural numbers, character strings, and boolean values.

We leave it open-ended which values are considered pro-textual constituents; adding a new class of such values involves implementation. A class of values is introduced when we are not normally interested in what expression-like structure they might be given, because their significance is determined by external uses. Future extensions may well require the introduction of new pro-textual values. We are also motivated to introduce "bit files" or "blobs" (binary large objects) as values to manage the relation between, say, program source code and it compilation (related ideas have been implemented by the Vesta[4] project). We would "outsource" the compilation process, but record the fact that a certain file is the result of compiling the given source code. The <kind> place is treated further in Initial Closed Map (section 4.6).

4.4.6 Problems with Naming

Individuals and traditions can disagree over the proper use of names informally, as well as whether a particular formalization of an informal named concept is appropriate. On such occasions, it may be important to change names. With appropriate technology, it would not be necessary to resolve such disputes; the parties could select the names they prefer and operate independently with regard to naming. Similarly, a person may come to regret their own name choices and desire to change them. This is most acute when a name belongs to an informal concept that does not match a formal use of the name, probably as a result of a mistake. And yet the material may be formally correct and worth preserving and using further, just with a better nomenclature.

Another form of dispute over names might be thought of as a property dispute. When two persons have independently developed formal material using the same name for different formal concepts, they may well agree that both are correct, but cannot formally combine their work because of the

[4] http://research.compaq.com/SRC/vesta/

unfortunate coincidence of naming. The parties must agree to renaming if they are to unite their work formally.

The name collision problem becomes particularly intense when programs operate on names as data and come to depend on them, and here it is not the human-mnemonic value of names but their role as identifiers that is our concern. While we may tend to think of renaming mathematical concepts or identifiers in programs as paradigmatic, when names are runtime data the problem becomes more difficult. If we attempt to combine programs by resolving name conflicts in their code, we may still be left with name collisions in their execution because they may generate names as runtime data that we cannot avoid by static renaming. See Abstract Ids & Closed Maps (section 4.4.1).

The above remarks treat the names as essentially interchangeable because they focus on the problems of maintaining formal correctness under change of names, and assume the desirability of methods for freely changing names. But there is also the problem of informal correctness. The names must be assumed often to matter to the human ability to understand and apply the formal material. This means that name changes must not be imposed on individuals without their control. Each user must be empowered to manage their own assignment of names, albeit with advice provided for naming objects and concepts they did not themselves create.

As suggested in Words vs Formality (section 4.2.1), by design we separate the nomenclature from formal content, making it changeable without effect on the formal content, and use objects in the FDL to specify for any given **Session** how to attach informal names to objects and concepts. This exposes naming (and renaming) to management by those that depend on it. Just as users may rely on stability of formal content, so they may rely on stability of nomenclature as they see it, and yet cannot impose on others their nomenclature for formal content. (Note that even informal nomenclature within informal documents may be made variable by picking an abstract name for use within the document, and binding it to a concrete name by user choice using the same mechanisms as for naming formal objects. This would be like using TEX macros in a document source, but applying systematic methods for managing nomenclature choices throughout the system.)

4.4.7 Abstract Id Allocation

Keep in mind that assignment of concrete mnemonics to Abstract Identifiers (section 4.4.1) is extrensic to the existence of abstract ids. Concrete names, when desired, are assigned for whatever purpose by specifying the connection in an object whose content, a table perhaps, stipulates the assignment. Such assignment can be altered without altering occurrences of the abstract ids.

We have decided to identify object indices and abstract identifiers. We shall discuss this below. But first let us consider some scenarios in which abstract identifiers are allocated for various purposes.

Suppose while developing a certain closed map (a function mapping object indices to object contents) one decides to add a new definition for some mathematical operator, say for GCD. If names were concrete, and users chose them, then allocation of a new name for this operator would comprise the user's coming up with a name not already used in the current closed map, and which one hopes will be acceptable in the future. (Complications involving errors and disputes could arise as mentioned in Naming Problems (section 4.4.6).)

But in our abstract identifier scenario, this name is selected by the system being employed to maintain the current closed map, and is not a subject of dispute. The user says "give me a new abstract id to work with," then proceeds to employ it, say by giving a definition intended to describe the GCD function, and proving some theorems about it. The user might also create a new object explaining the intended meaning of the new operator informally, as well as stipulating how to show instances of the new operator consonant with its intended meaning.

It is then feasible to later retain the definition and proofs involving this operator while rejecting or altering the informal explanations and advice that refer to it.

Names are needed for things besides mathematical operators, of course. Programming operators, macros, procedure names; command interface components; and, especially, objects in the FDL, where most of the content is expressed. Examples of FDL objects are: definitions of constants, operators and macros, proofs, source code for programs, specifications for how to concretely display abstractly structured expressions, data used as arguments to user-defined programs, informal documents, data stipulating external relations between formal objects, Readings (section 4.2.3).

We have chosen to identify abstract identifiers with object indices in

"Closed Maps" (section 4.4.1). In doing so we identify object allocation with identifier allocation. To add a new object to a closed map is to add a new object index and extend the closed map with a new object content assigned to that index. To get a new abstract identifier is to add a new object, which will be indexed by this new identifier. See Adequacy of Single Id Space for arguments that this collapse of all abstract identifier spaces to the space of object indices is without loss of generality.

4.4.8 Adequacy of a single space of Abstract Identifiers, as object indices.

Here we consider a couple of reservations about the identification of abstract identifiers and object indices. First, is it okay to have a single class of abstract identifiers rather than several independent spaces? Second, if we do have a single space of abstract ids, is there something wrong with assigning an object to every one?

The possibility that one might want multiple independent spaces of abstract ids is solvable by reduction. To introduce a new class of abstract ids, pick a new abstract id from the basic system to represent the new abstract id space, and implement abstract ids of this space as *pairs* of basic abstract ids, one being the space identifier, the other identifying the member of the space. Thus allocation of ids in separate spaces is easily implemented on top of the allocation of basic ids.

As for the assignment of object content to each allocated abstract id, the worst case is that an "empty" content must be distinguished as a way of coercing unattached ids to object ids, i.e. persons who wish they could have ids without assigned objects could for their own purposes interpret object indices with empty content as such. But we think it is actually quite natural to associate content proper with abstract ids. For example, when a new operator is introduced (as either defined or primitive) it seems most natural to associate with its identifier the formal object that gives its definition or that declares it primitive. In such cases, each occurrence of this new operator would directly refer to its definition (or declaration). When one introduces an abstract id for a procedure name, it would be quite natural to have that name refer to the object containing the source code that defines it. If one introduced an abstract id for a special purpose variable, it would be natural to associate an informal explanation of how that variable is intended to be used. In all these cases, it is good to associate with an abstract id some data explaining, either formally or informally, its use. Of course, the most common reason for introducing abstract ids will be in order simply to refer to objects in the first place.

4.5 Keeping Records in the Repository

We elaborate in some detail how facts are established and recorded by the FDL.

4.5.1 Certificates

Among the objects included in closed maps are "certificates." Certificates belong to the system, ordinary objects belong to the **Client**. Be warned that the notion of certificate discussed here is generic, although explicitly formulated, and provides merely a *form* for accounting for facts. The substance lies in the design and implementation of particular kinds of certificates.

Clients can create and modify ordinary object content pretty much free form. But they cannot force a certificate with a certain content to be created or altered or, when objects referenced by a certificate are altered, to be preserved; certificates are a kind of derived content. The client can delete them, with consequences, and request certification services such as attempting to create certificates of specified kinds. The client can also create content that refers to certificates, just like to any other objects. See Current Closed Maps (section 4.5.2).

Certification services and kinds of certificates are open-ended, subject to extension by implementation. The basic significance of each kind of certificate must be explained when it is introduced, and provides a basis for promises and correctness claims on the part of the system implementors. See Certificate Significance (section 4.5.4) and Certificate Structure (section 4.5.6).

The dominant kinds of certificates we have been anticipating are for formal proof certification (section 4.5.11.1), recording claims that certain inferences have been validated via specified methods. The basic intention we anticipate is that a kind of certificate is designed with the aim of supporting claims about closed maps based upon the existence and content of those certificates. Here is a strong paradigm case: the client may infer from a certificate about a formal proof that it conforms to specific methods of inference, and could test this claim.

Consider also an example of a form of certificate which has no value, its being simply an extreme possibility: suppose that a certain kind of certificate can be created willy-nilly by the system and added to the client's current closed map, and that it has no content except the indication of its kind; there is nothing of interest that one may infer from the existence of such a certificate.

Another kind of certificate that would be useful: the client may ask that a certain compiler be run on some source code object in the current closed map; the system then runs the compiler according to the specification, saves the output as a new object in the current closed map, and adds a certificate of this kind that points to the source-code object as well as to the new load-module object (say); if one later wanted to execute the code, one might use the existence of this compilation certificate to justify later claims about execution of this program involving the load-module.

Another: a certificate kind could be designed to mean that, by some specified method, a date and client identity for a **Session** have been ascertained when a certain object has been updated or created.

In each of these scenarios it is possible for multiple certificates of various content and import to be created by the system. Thus, a **Proof** or **Inference Step** might be validated by several independent (in various ways) **Inference Engines**, each certified; different compilers might be applied to the same source code, the certificates indicating them and their outputs; different means of establishing date and identity might be employed, with perhaps with varying degrees of credibility.

We give preferred treatment, described in Certificate Bias (section 4.5.3), to certifications of facts about closed maps that are abstract, monotonic, and "localized."

4.5.2 Current Closed Maps

As was mentioned in Abstract Ids & Closed Maps (section 4.4.1), in a **Session** a **Client** develops a current closed map, normally by initializing it from the FDL, transforming it through a sequence of operations to generate new closed maps, then storing it back into the FDL in such a way that it can be later retrieved easily. Our focus here is on sequences of closed maps that may constitute the current closed map's history, especially with regard to

Certificates, rather than on interaction with the FDL.

Some operations on closed maps which will be used to modify the current closed map are listed in Conservation and Destruction (section 4.4.3). Basically they involve "conservative" operations that leave certificates intact, and operations that may force modification of certificates. Whether an object is a certificate is conceived here as a distinctive property of the content.

It should be kept in mind that during a **Session** the current closed map is part of the session's state, and that the map will typically include programs that can operate on the state.

When a kind of certificate is implemented, policies for creation and alteration are stipulated, and the "internal" significance of the existence of a certificate object is that the current closed map has been developed from an Initial Closed Map (section 4.6) (inherent to the FDL) by a sequence of transformations enforcing policies implicit in the rules for introducing or altering certificates in the FDL. Actually, we must weaken the claim slightly to say that the current closed map *could* have been so developed, since it may have been initialized from an FDL, and so part of being an FDL of closed maps is that any maps extracted from it could themselves have been built from the initial closed map by a sequence of transformations.

Design of certificate policy is nontrivial, especially as regards incremental reconsideration of doubtful certificates, and proposals follow (start with Altering Certificates (section 4.5.9.1)). But one part is simple, namely some generic operations that cannot be constrained by certificate policies:

- One can always delete a collection of objects as long as every object referring to an object in the collection is itself in the collection.
- One can always uniformly rename objects, so long as no distinct objects become identified (references in all objects are altered as part of the renaming).
- One can always add a new object with any non-certificate content you please.
- One can always "clone" any collection of objects. Cloning some objects means making a new collection of objects with identical contents except that the clone references replace the original references.

Let us call these operations "conservative." Conservative operations are simple, whereas other operations can force change of certificates - see Altering

Certificates (section 4.5.9.1).

4.5.3 Certificate Bias

As was mentioned at the end of Certificates (section 4.5.1), we give preferred treatment to certifications of facts about Closed Maps that are abstract, monotonic, and "localized." Elsewhere we shall address the questions of what kinds of certificates are likely to be preferred, and which disfavored, and why we give this preferential treatment. But, here we address: What constitutes this preferential treatment?

When the user's current closed map is modified, the FDL interface system is responsible for deleting or altering certificates, and advising the user about problems, especially to avoid surprising de-certification which could be expensive to recover from by recertification.

Setting aside the issue of how to forestall undesired damage to a current closed map, what happens to certificates normally? We shall use program execution in some scenarios because the problems should be familiar, and because some methods of proof (eg, **Tactics**) involve execution of programs from a general purpose programming language.

Here are the features of certificates about closed maps mentioned above, and examples of *dis*favored kinds of certificates. Following each explanation of a feature is a description of undesired consequences of using certificates lacking the feature; the fact that such undesired consequences arise constitutes the system's bias towards violated features, ie, the system treats certificates having those features more favorably.

Abstractness: This means abstractness with respect to object identifiers (see Abstract Ids & Closed Maps (section 4.4.1) and Abstract Identifiers (how) (section 4.4.4)). When a closed map is uniformly renamed or is retrieved from the FDL, which is only guaranteed modulo renaming, there is no rechecking of certificates; they are treated like any other objects.

Suppose, for example, that during a session, a person goes out or their way, and against advice, to store a program whose execution depends on the concrete values that happened to be used in that session for object ids, and has the system execute that program and certify a result. Then if that current closed map is saved and reloaded (modulo name change) in a later session, that certification object will cease to entail that the old connection between the program and result has been preserved.

Monotonicity: Similarly, suppose one wrote a program whose execution involves a heuristic search of the whole current closed map, whatever it may be at execution time. Then executing the same program on a extension of the same closed map may well have different results, thus a certification of the result of executing in one closed map will not certify the result in an enlarged closed map.

But, again, the system will not modify a certification object when the current closed map is enlarged.

Locality: Locality of a claim based on a certificate is dependence of that claim only upon the certificate and objects referred to by the certificate. A localized claim will also be monotonic since extending the map doesn't change what a localized claim about a certificate depends on.

Suppose that one cuts down the current closed map to a submap by Contracting or Focusing (see Closed Map Operations (section 4.4.2)). For example, suppose one picks a certification object and Focuses on it, deleting every object that is irrelevant to it (ie, no reference path between them). The certificate will not be modified or checked. The problems are the same as for monotonicity above.

See Conservation and Destruction (section 4.4.3) for a list of some closed map operations that leave certificates intact.

4.5.4 The Significance of Certificates

The "internal" significance of Certificates (section 4.5.1) in a closed map is determined by the policies for how they are created, deleted and altered. Some policies for the effects upon certificates of various closed map operations are suggested in Certificate Bias and in Conservation and Destruction (section 4.4.3). Thus, the existence of a certificate in a current closed map directly indicates merely that at some time it was created in a current closed map according to a creation-policy implemented for that kind of certificate, and has not been deleted by various subsequent alterations of the current closed map, and that any alterations to the certificate's content has been according to the policy implemented for certificates of its kind. As usual with formal and computational data, further significance is attached externally based upon understanding the "internal" significance.

The only general policy for changing certificate content that we currently expect to adopt was alluded to in Certificate Bias (section 4.5.3); the system can change the content of a certificate by marking it as "stale" and possibly deleting object references within it to no-longer extant objects. Such a vestigial certificate thus signifies merely the past existence of a certificate having certain content. These vestigial certificates will, therefore, tend to have little formal value, and are expected to serve as hints about previous situations, which may have some heuristic value. See Certificate Structure (section 4.5.6) and Stale Certificates (section 4.5.9.2) for detail on certification policy.

A simple content alteration policy, one might deem it the default, is extreme sensitivity to the content and number of objects referred to by the certificate. It is this: if the content of any object the certificate refers to is changed, then the certificate must be deleted (or its content must be altered to mark it vestigial); its fate will be the same should multiple references within the certificate get "identified," by Folding (section 4.4.2), say.

For example, if a proof "goes bad" because of some alterations to underlying content, say a change of definition or deletion of an inference rule upon which it depended, or through forced identification of two "primitive" notions, then the objects certifying it as a well-formed proof (according to whatever standards) will be deleted or marked as stale. If they are altered to these vestigial forms rather than deleted, they may be useful in further attempts to attach new certificates to the proof.

A more permanently useful vestigial certification would be a certificate of object creation. An object creation certificate might be implemented to indicate that an object it refers to was created at a certain time in a certain **Session**. Then, as we can infer from the general policies mentioned above, a vestigial certificate of this kind still signifies that the object was created at the specified time but may have undergone a change of content. So some vestigial certificates do retain their significance.

One might also implement kinds of certificates that are insensitive to some alteration of content in referenced objects. For example, one might adopt a criterion for mere annotation of programs or formal data that does not affect correctness, and leave "annotation insensitive" certificates intact if objects they refer to are altered merely in their annotations.

Returning to external significance of certificates, it is also possible for us to be mistaken or in disagreement about the external significance we have attributed to a kind of certificate. This lies beyond the responsibility of the closed map management system. For a scenario exploring this situation see Conflicts of Significance (section 4.5.5).

The most important promise implementors can make with regard to certification is that the policy for creation and update of each kind of certificate is permanent, once implemented. If a new policy is needed or desired, a new kind of certificate must be implemented.

4.5.5 Conflicts of Certificate Significance - a Scenario

Continuing the discussion of internal and external significance of certificates begun in Certificate Significance, it is possible for us to be mistaken or in disagreement about the external significance we have attributed to a kind of certificate. For example, suppose we have employed a kind of certificate for generating or verifying formal inference steps; the certificate policy might go something like this: invoke a specified inference engine, building it first (perhaps compiling its code) if need be, and apply it to an inference step object in the current closed map; if the inference engine says the step is "good," create the certificate, whose content will refer to the inference step. This is the "internal" significance. External significance to some persons might comprise the conformance of the inference step to a certain independently understood class of inference rules. For a person who judges all these rules to be correct, the external significance may further comprise the validity of the certified inference step. Of course, maybe it will be later discovered or suspected that the inference engine is flawed and doesn't simply implement the inference rules it was thought to implement. Then that external significance is lost.

Let us continue with this example of the flawed proof engine, and consider a recovery scenario. Suppose that further study and improvement of the inference engine leads to the judgement that there is an easy bug fix, and that there is a simple test that detects at least the bad inferences that the older version of the engine erroneously okayed. Now we have fallen into doubt about already certified proofs. One simple fix is to employ a similar form of certificate, the difference being that it invokes the new engine rather than the old one, and simply try to certify all the old inferences with the new form of certificate. The old certificates need not be deleted, although perhaps they may be eventually.

Consider now a more sophisticated scenario. Stipulate a form of certificate that is created this way: given an inference step, first look and see if there is a certification for it (pointing to it) of the old sort that uses the defective engine; if there is no old-engine certificate then use the new engine; if there is an old-engine certificate then apply the test for possibly-bad inferences postulated above for this scenario; if the inference is possibly-bad then use the new engine, but otherwise, simply point to the old-engine certificate. If there were many proofs with the old engine but the bulk of inferences were okay according to the new test, and if running the engine is often expensive, then this method could represent a significant efficiency as a corrective.

Let's do a quick survey of when these various certificates are appropriate. Let's also make the more interesting assumption that whether the engine is flawed is not agreed upon by all parties. Persons who think the old engine was correct will still accept the old certificates with their original significance for correctness. Persons who think the new engine is correct and that the test for possibly-bad cases is an accurate assessment will accept both new-engine certificates and also the hybrid certificates that depend on the old engine. Persons who think the new engine is correct but don't trust the test for possibly-bad cases will insist on the simple new-engine certificates. All these certifications can coexist, even if their external significances are in dispute.

4.5.6 Certificate Structure and Internal Significance

Continuing from the discussion in Certificate Significance (section 4.5.4) we elaborate upon what counts as a certificate, and how certification procedures are determined by certificate content.

Before getting specific, let us sketch the design. The principal, dominant form of certificate indicates two **Native Language** programs, the first being the procedure invoked to create the certificate (and others like it), and the second being a procedure to be applied when "reconsidering" (section 4.5.9.1) the certificate later. These native language programs are pointed to by the certificate, and to understand them along with understanding the general method for reconsidering Stale Certificates (section 4.5.9.2) is to understand the "internal" or direct significance of the certificate. In order to facilitate Borrowing Certificates (section 4.5.2), we admit a special kind of certificate that has the same content as a normal "native" certificate, but which cannot be updated and can be created only by copying from a foreign FDL. A third kind of certificate-like object is a "certificate identifier" which may exist in the FDL *ab initio*, but is a distinguished object created as part of the FDL in order to identify other certificates by their content.

The unifying characteristic of these certificates is that their content is strictly regulated by the FDL, unlike ordinary objects whose content is whatever the **Clients** make it. We defer discussion of Certificate Identifiers (section 4.5.8), stipulating here that the FDL classifies each certificate identifier as either "native" or "borrowed."

Both native and borrowed certificates of the ordinary kind, ie not certificate ids, have as content a **Text** whose operator consists of two object references, the first being a certificate identifier and the second being a reference to an appropriate code specifying object. The intention is that this second object contain the **Native Language** programs for creating and updating certificates referring to it in this way. This pair $\langle C, K\rangle$ of references constituting the certificate contents' operator may the considered the certificate "kind," and it can never be altered in an object once created. The content of K cannot be altered as long as K is part of the certificate kind for any existing certificate; this policy is part of what makes the FDL trustworthy. If you want better code and the old code still matters to someone, you must make a new object K' and employ a new certificate kind $\langle C, K'\rangle$.

A certificate with kind $\langle C, K\rangle$, where C is a native-certificate identifier, can only be created by interpreting the first subtext of object K, applied to some specified arguments. If this execution returns a **Text** T, then a new object is created whose operator is $\langle C, K\rangle$ and whose first subtext is T. There may be other subtexts as well indicating generally useful information as determined by the FDL implementors, such as dates or other transaction related data. How the native language program is interpreted is determined by the C in a way inherent to the FDL process. Thus, within an FDL, there may be multiple native language interpreters indexed by the certificate id. If one needs to add a native language to an FDL or improve a native language of an FDL, then a new certificate identifier should be introduced for the new interpreter. Once an interpreter is employed it must be unaltered if clients are to be able to trust the FDL. Similarly, different FDLs may have different implementations of a native language, perhaps with different execution results either intentionally or accidentally, or may support different native languages. Thus, native certificates must never be transferred as such between FDLs if client trust is to be maintained.

A certificate with kind $\langle C, K\rangle$, again where C indicates native, can be updated only under circumstances explained in Stale Certificates (section 4.5.9.2). Then the second subtext of the content of object K is executed as a native language program according to C, applied to arguments as stipulated in Stale Certificates (section 4.5.9.2). If the procedure returns a text, then that becomes the new first subtext of the extant certificate. Again, any other subtexts are updated as the FDL implementors choose. We shall consider

Borrowed Certificates in more detail.

4.5.7 Borrowed Certificates

In Certificate Structure it is explained how creation and update code are determined for "native" certificate identifiers. Here we explain how certificates are borrowed from other purported FDLs.

A certificate with kind $\langle C, K \rangle$, where C is classified by the FDL as indicating a borrowed rather than a native certificate, cannot be updated – reconsideration of borrowed certificates always fails, which means that if they get in the way they should be replaced by native certificates. When one wishes to locally "re-establish" a borrowed certificate, one requests the creation of a native certificate stipulating a native certificate id C' which one thinks will execute the native code of K similarly to the way the foreign FDL is supposed to have interpreted it. FDLs that hope to cooperate and share **Clients** would do well to provide a standard collection of similar native languages and interpreters; multiple instances of a common implementation will, of course, minimize coding in this regard.

A certificate with kind $\langle C, K \rangle$, where C is a borrowed-certificate identifier, is created by Merging (section 4.4.2) a **Closed Map** from a foreign FDL into the local FDL. When a merge is attempted one specifies a correspondence between objects that should be identified. If one stipulates that certificate id C in the local FDL is to be identified with C' of the foreign FDL, then the merge fails if any *new* certificates of kind C would have gotten imported; ie, foreign certificate ids can only be identified with native certificates ids if the certificates depending on them were already in the local FDL.

How can a cross-FDL correspondence between identifiers, certificate identifiers in this case, be established when each space of identifiers is "internal" to its own FDL? In general, a **Client** of an FDL might establish an association between abstract identifiers of the FDL and other values (which might themselves be either concrete values or abstract identifiers), by storing in the FDL a text serving as a table pairing the values with the abstract identifiers. One FDL, say A, could borrow from another FDL, say B, by becoming its **Client** and creating two tables, one in A and another in B. One concrete value for each cross-FDL abstract id pair would be generated, and these values would mediate the two tables in A and B.

Establishing such cross-FDL abstract identifier correspondences, to facilitate certificate borrowing or other content sharing more generally, should be so common that it should be "internalized" as an FDL service and obviate the need for exposing the intermediate concrete values that coordinate cross-FDL pairings. Further, significant efficiencies are likely in explicitly recognizing a mutual client relationship between FDLs. Hence there is a special role for FDLs as clients of other FDLs, essentially supporting federation, that is worth making efficient.

4.5.8 Certificate Identifiers

The content of a certificate identifier is a **Text** whose operator is simply a reference to itself, and the subtexts, if there are any, may contain whatever is convenient or necessary to execution of **Native Language** programs as explained in Certificate Structure (section 4.5.6). The content of certificate identifier cannot be changed as long as any certificates mention it, hence the certificate identifier serves as an authentic indicator of how the certificates referring to it were established.

The classification of an abstract identifier as a certificate identifier must be inherent to the FDL, as is its further classification as indicating normal certificates as opposed to indicating Borrowed Certificates (section 4.5.7). Certificate identifiers for borrowed certificates must be bound by the FDL to a Process Identity Certificate (section 4.7) for a foreign FDL, and to a cross-FDL abstract id correspondence as described in Borrowed Certificates (section 4.5.7).

4.5.9 Updating Certificates

It is possible to manage the modification of certificate objects by methods specific to the kind of certificate. The point is that when large numbers of inter-related objects are created by means of automated procedures, such as proof generators, it will often be desirable to create slight variants of large data that should have rather localized effects. An economy can be achieved when this localization can be effectively detected and managed automatically. Of course, one is always free to stipulate kinds of certificate that are unmodifiable once created.

4.5.9.1 Altering Certificates

In Current Closed Maps (section 4.5.2) we specified some operations on the current closed map considered “conservative.” Conservative operations are simple, whereas other operations force reconsideration of some Certificates (section 4.5.1) which may have become “stale” as a result. The assumption here is that certificates are normally intended to attest to some facts about the contents of, or identity between, objects and so may fall into doubt when those change. Rather than simply deleting or discounting certificates that become doubtful, we anticipate a more incremental processing of doubtful certificates that might rehabilitate some certificates or even leave them intact. The presence of stale certificates in the current closed map corresponds to an “inconsistent” state in a database, and part of completing an operation on the current closed map is to eliminate staleness. Reconsidering a certificate can result in its modification or deletion, which can force reconsideration of further certificates that depend on it, and so a protocol is needed for resolving these cascades of certificates going stale. See Stale Certificates (section 4.5.9.2).

As explained in Certificate Structure (section 4.5.6), to implement a kind of certificate one adopts a procedure for creating new certification objects of that kind, and a procedure for reconsidering a certificate. When a reconsideration procedure for a certificate is executed to (successful) completion, the certificate object is either left intact, updated, or deleted. These certification procedures, in addition to creating certificates or modifying contents of “reconsidered” certificates, may create or delete other objects, and may alter the contents of non-certificates. But some basic operations may have cascading consequences on the FDL, beyond the control of any specific certification procedure:

- Execution of a certificate creation procedure is a basic operation on the current closed map, and the content of the certificate will include an indication of its kind. Creation procedures can take arguments supplied at execution. The indication of certificate kind is beyond the reach of the certificate creation procedure stipulated for the kind itself, and is controlled by the system.

- Execution of a reconsideration procedure for a kind of certificate is a basic operation that can be applied to any extant certificate of that kind. Again, the content cannot be altered to omit the fact that it is a certificate of its kind.
- The content of any non-certificate can be changed to any content except that it cannot have the form characterizing certificates.
- Object indices can be identified with each other (see Folding (section 4.4.2)).
- When any object's content is altered other than by conservative operations (see Conservation and Destruction (section 4.4.3)), be it a certificate or not, each certificate object referring to it in certain limited ways (section 4.5.9.3) 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. If reconsidering a certificate leaves it intact, then it engenders no further marking for reconsideration.
- Similarly, when multiple objects are identified with each other (see Folding (section 4.4.2)), any certificate that contains references to more than one of them gets marked for reconsideration.

See also Assimilation to Certificates (section 4.5.10).

Of course, there are practical issues of making these current closed map transformations convenient. For example, users must be able to abort transformations and get advice about consequences of proposed transformations.

4.5.9.2 Stale Certificates

In Altering Certificates the notion of "stale" Certificate (section 4.5.1) was introduced as a sort of database "inconsistency" induced by various operations and resolved by certificate "reconsideration" procedures. Here we elaborate on the methods for inducing and resolving staleness.

We do not consider the staleness of certificates in the current closed map to be part of the map, but rather part of the state of the **Session** to which the current closed map belongs. As will be explained, other data pertinent to reconsidering a stale certificate are also maintained along with the mark of staleness.

Some quick orientation: At the level of interaction via a **Session** with the FDL process, current closed maps have no stale certificates; every operation on the current closed map at this level is automatically followed by reconsideration of all certificates marked stale. If an operation is attempted which induces staleness in any certificates, and those certificates are not recertified, rehabilitated or deleted, then the operation fails leaving the current closed map as it was.

There is a procedure stipulated as part of a certificate's kind (section 4.5.1) for reconsidering a stale certificate of that kind, and it is the execution of this procedure that effects the recertification, rehabilitation, or deletion or the certificate.

The operations on the current closed map that can induce staleness in pertinent certificates are updating the content of an object, identifying two or more objects that were previously distinct (see Folding (section 4.4.2)), stipulation of a set of objects to be "shunned" in anticipation of deletion, and direct stipulation that a certificate is to be considered stale. When one of these occurs, data which may help to rehabilitate the certificate is saved as well, such as the old content of an object that has been changed.

Not all certificates that may eventually become stale as the result of a change to the current closed map are necessarily marked stale at first; this makes it possible for some certificates to serve as buffers against various changes deemed "irrelevant" by other, more remote, certificates. When reconsidering a certificate object results in its alteration, pertinent certificates which depended on it in turn will are marked stale, thus a cascade of staleness marking is possible.

This presents the following issues:

- What are the "pertinent" certificates for reconsideration when an object is altered, multiple objects become identified, or a set of objects is to be shunned?
- When a staleness inducing operations occurs, what data are saved for pertinent certificates?
- How is a certificate's reconsideration procedure applied in order to reconsider the certificate?

See Pertinence, Extra State (section 4.5.9.4), and Resolving Staleness (section 4.5.9.6).

4.5.9.3 Certificates' Pertinence to Objects (stipulation)

Here is a key concept used in propagation of Stale Certificates. It is based on the reference relation (section 4.4.1) between objects:

> An object X refers "simply" to an object Y just when X refers (perhaps indirectly) to Y via a reference path with no interior certificate references. That is, either X refers *directly* to Y or X refers directly to a non-certificate object Z that refers "simply" to Y.

As will be seen in Staleness (extra state) (section 4.5.9.4), the certificates pertinent to operations involving some change to a set of objects are those that simply refer to objects in that set. Certificates are intended normally to be directly "about" the content and identity of the objects to which they simply refer, and they get marked for reconsideration as a result of changes to such content or identity.

This strategy is adopted rather than one of two more obvious ones for the following reasons. A simpler strategy would be to force reconsideration for all certificates that refer at all to the affected objects because they are, after all, about those objects and so could go wrong. However, we expect many certificate kinds to be *designed* to be independent of various features of the objects they refer to, for example such as what "comments" may be adorning their contents. When the reconsideration procedure stipulated for that kind of certificate is executed, it may ascertain that the referenced objects have not changed in ways it considers relevant and simply resolve the certificate by leaving it intact; then our staging strategy avoids forcing reconsideration of other certificates on account of changes to this one.

This staging strategy enables the use of certificates as buffers against certain changes to their subjects. We shall use the term "buffering certificate" to mean certificates that remain intact when reconsidered due to certain changes to the objects they simply refer to. In order to make a certificate unaffected by certain changes to objects, rather than having it simply refer to those objects, one can have it refer to such "buffering" certificates that refer to those objects; then such changes will not even induce reconsideration beyond the buffering certificates.

That expensive strategy of reconsidering all (indirectly) referring certificates would at least have been useful, and indeed is equivalent to our staging strategy when buffering certificates are not employed, since if no certificates remain intact after reconsideration then all the (indirectly) referring certificates will eventually get reconsidered (unless there is a failure earlier).

Consider another extreme strategy we have not adopted. Suppose that rather than forcing reconsideration of all *simply* referring certificates, we were to force reconsideration of only *directly* referring certificates. Under such a regime, certificates must be designed to carefully restrict their attention to those objects they directly refer to, which would fail to exploit a major convenience of our closed map design, namely allowing free and easy access to all the content one can reach from a small collection of object identifiers, i.e. allowing small expressions to refer (indirectly) to large numbers of objects.

4.5.9.4 Certificate Staleness Processing State

The management of Stale Certificates (section 4.5.9.2) involves some extra state beyond which certificates of the current closed map are deemed stale. Recall that the ordinary state of the current closed map is that there are no stale certificates, and that staleness is intended as a temporary state; failure to eliminate staleness results in failure of the original operation and restores the current closed map to its original “prestale” state.

In addition to the finite collection of stale certificates, there is for each stale certificate a collection of those objects that it simply (section 4.5.9.3) refers to whose contents have changed since the prestale current closed map; and each of those entries in the “changed object” collection is paired with the prestale value, which the certificate was presumably originally about. The purpose of this, as explained in Staleness (pertinence) (section 4.5.9.3), is to anticipate reconsideration procedures that leave the certificate intact based upon some relation between the old and new values that is deemed irrelevant to the the correctness of the certification. It is also possible that even if reconsideration does change the certificate content, thus forcing propagation of staleness to further pertinent certificates, knowing the difference between the old and new contents of simply (section 4.5.9.3) referenced objects may permit a more efficient incremental recertification than simply rerunning the certificate’s origination procedure.

When the content of any object is updated, each certificate simply (section 4.5.9.3) referring to it is marked stale if it is not already so marked. Further, for each of these certificates the object is added to its “changed object” collection along with the prior content, unless it is already in the collection, in which case it is left as is.

Another operation on the current closed map that changes the staleness state is Folding (section 4.4.2), which “identifies” some distinct object identifiers with each other. For each stale certificate there is a "folded object collection" of the objects that it simply (section 4.5.9.3) refers to that resulted from such an identification of originally distinct identifiers. Similarly to the content change case above, the purpose is to admit the possibility of leaving the certificate intact or enable a more efficient incremental recertification.

Upon Folding (section 4.4.2) the current closed map, any certificates that simply (section 4.5.9.3) refer to any of the newly "identified" objects are marked stale if not already so marked, and the identified objects get inserted in this folded object collection. It should also be noted that the extant "changed object collections," "folded object collections" and the stale object collection must themselves be collapsed to reflect the new identifications.

Another operation is simply to mark a certificate as stale even if nothing it simply (section 4.5.9.3) refers to changes in order to express doubt for external reasons. In order that the nature of this doubt may be communicated to the procedure for reconsideration, this operation takes a **Text** as argument to be saved with the stale certificate. With each stale certificate, therefore, is associated a (finite) collection of these "staleness fiats;" this operation then consists of marking a certificate as stale, if it's not already, and adding the given "staleness fiat" to the collection for that certificate.

Next we consider certificate deletion.

4.5.9.5 Staleness Processing State Continued - shunning objects and deleting certificates

We continue the discussion of State for processing stale certificates. There we introduced state and operations for incrementally adapting certificates to change of object content and identity. Here we introduce state and operations for facilitating incremental anticipation of object deletions.

In addition to the operations described in Staleness State (section 4.5.9.4), another operation is to indicate that some objects should be "shunned," which is simply a device for passing another set of objects to the certificate's reconsideration procedure, but the intention is that the reconsideration procedure should try to update the certificate content in such a way that it avoids reference to the "shunned" objects. The purpose of this operation is its use prior to deleting (section 4.4.2) a collection of objects from the current closed map. When objects are deleted so are all objects that depend on them; by "shunning" the objects beforehand, the reconsideration procedure has a chance to rebuild the certificate to avoid referring to the objects whose deletion is to come, which would otherwise simply delete the certificate as well. Shunning an object thus makes any certificate referring simply (section 4.5.9.3) to it stale, if it's not already, and adds itself to the "shunned object collection" for each of those certificates.

Finally, because reconsideration of a certificate may result in deletion of the certificate, and because we want to forestall deletion of such certificates until other certificates depending on them have had a chance to shun them, another part of the staleness processing state is a set of certificates marked for later deletion. When the reconsideration of a certificate requires its deletion, it is marked for deletion then shunned rather than being immediately deleted. When no stale certificates remain, certificates marked for deletion are deleted along with every object that refers to them. Of course, if any reconsideration procedures fail, then this point is never reached, and indeed this device could be exploited to protect an object from accidental deletion by referring to it with a certificate whose reconsideration procedure always fails.

See Staleness State (section 4.5.9.4) and Resolving Staleness.

4.5.9.6 Resolving Stale Certificates

As indicated in Altering Certificates (section 4.5.9.1), rather than simply adopting the expensive policy of simply deleting, or permanently marking as obsolete, Certificates (section 4.5.1) that fall into doubt, we permit the stipulation of a "reconsideration" procedure as part of defining a kind of certificate.

The arguments to the reconsideration procedure are the (object id of the) certificate to be reconsidered and various values attached to that certificate object described in Staleness State (section 4.5.9.4) and Staleness and Deletion (section 4.5.9.5), namely: the "changed object collection," along with the "prestale" contents associated with each member of the collection; the "folded object collection" of objects; the collection of "staleness fiats;" and the "shunned object collection."

When a stale certificate is reconsidered, this procedure is applied and the disposition of the certificate is determined by its result. If the procedure fails then the certificate remains stale, and presumably control passed to wherever the failure is caught. If the procedure completes, it returns one of three values: "intact," "delete," or a "new content" text. In any case the certificate is no longer stale; if the result of reconsideration is “intact” then that’s it for that certificate and no further staleness is induced by this; if the result is “delete” then the certificate is marked for deletion (section 4.5.9.5) and and the certificate is then shunned (section 4.5.9.5); if the result is a “new content” text then that text becomes the new content (except for the part indicating the certificate kind) of the certificate and pertinent certificates depending on it go stale.

Let us emphasize a few points:

1. Although the **Client** can force deletion of a certificate (along with everything that refers to it) from the **Current Closed Map** of the **Session**, the only way to modify its content is by reconsideration.

2. Part of a certificate is an indication of its kind, which determines its creation and reconsideration procedures, and that part is beyond the reach of the reconsideration procedure (or the creation procedure for that matter).

3. Certificates will not be deleted until no stale certificates remain; they can only be marked for deletion until then.

4. Changing a certificate’s content or marking it for deletion will make any pertinent certificates referring to it stale, thus it is possible for reconsideration to cascade.

5. The difference between leaving a certificate “intact” upon reconsideration andusing its old content as its new content is that the latter will induce staleness in pertinent certificates referring to it.

4.5.10 Assimilating Ordinary Objects to Certificates

We have presupposed a distinction between ordinary FDL objects and certificate objects. It would be possible to treat non-certificate objects as a specific kind degenerate of certificate object. To do this we would have to allow reconsideration procedures for certificates to take optional arguments; whether there would be some benefit other than contriving this assimilation remains to be seen.

Let the creation protocol take as argument a possible content, and let executing it simply be using that argument as the new certificate’s content.

The "reconsideration" procedure would take an optional argument; if the optional argument is not supplied, then leave the object intact, otherwise update the content to the specified argument.

Of course, if we were to do this assimilation to a single kind of object, we should drop the terminology of "certificate" in our general exposition, and simply say each object kind has creation and reconsideration procedures. The fact that simple reconsideration of our "degenerate" kind of object leaves its instances intact means that their being marked for reconsideration would have no effect.

4.5.11 Proofs

We consider how to deploy certification systems to represent proofs.

4.5.11.1 Proof Organization and Certification - independence of inference steps

Different organizations of **Inference Steps** into proof structures and certifications are possible. We shall assume that the inference steps of a proof are organized into a dag. For convenience of discussion we assume the dag has a single root, i.e., a node having no parents, and that the proof purports to justify the conclusion of the root from the premises of the leaf nodes, i.e. the conclusion of the proof is the conclusion of the root inference and the premises of the proof are the premises of the leaf inferences; more generally any dag could also be construed as the collection its rooted subdags. An extreme form of proof organization would be to represent a proof as a rooted dag of certificates each certificate comprising the conclusion of the inference, references to the root certificates of the proofs of the premises of the inference, and an indication of a justification (such as an inference schema or **Tactic** code) of the conclusion from the premises. The principle drawback of this simplest organization of proofs would be that it does not anticipate the potential costs of justifying an inference.

In a realistic proof system one can expect the cost of inference to dominate, whereas the organization of inferences into a rooted dag is cheap. There are reasons to certify individual inferences independently of other inferences in the larger proof, so let us assume that each inference will be represented as the content of an **Inference Step** object (a non-certificate), comprising the conclusion and premise **Propositions**, and a **Justification**, which is simply an expression used by an **Inference Engine** to help recognize the inference.

By making the inference step content external to certification content it becomes possible to have multiple certifications by different engines for the same inference. By making the proof structure external to the inference step content, i.e. by *not* having the inference step point to proofs of its premises, we make it possible for the same inference to be used in multiple proofs without recertification. While one might imagine having two otherwise unrelated proofs sharing an inference step, the main sharing would be across time among intermediate partial proofs being developed toward a single large proof, or across versions of a proof. Observe that if certifying an inference step in a proof required certifying the proofs of the premises as well, building a proof top down, i.e. by progressive arguments for premises, one would either require repeated certification of inferences as the subproofs were built or one would have to forestall certification of each inference until the whole proof was finished, both of which would be absurd. Another interesting potential of this proof organization is the development of hybrid proofs, in which different kinds of inferences can be performed by distinct inference engines on the various steps in a certified proof. A procedure for "passively" certifying a proof would not certify inferences, but rather would simply involve collecting extant certifications of inference steps if possible. Passive certification of a proof would take as argument a rooted digraph of inference steps and a test for acceptability of inference certificates. It proceeds by checking to see that the children of the root inference have as their conclusions the premises of the root; also it checks to see that there is a certificate object verifying the root inference and which passes the test for acceptability; then the procedure is applied recursively to each of the subgraphs for the children of the root; if all this succeeds then build a proof certificate comprising the specification for acceptable inference certificates, references to an acceptable root certificate and to each of the recursively created proof certificates for the children of the root. Note that if this procedure terminates, then it describes a dag of inference certificates for inference steps with the right correspondence between parent premises and child conclusions. Obviously, rather than recursing perhaps forever, we would use a loop detecting variant that would fail on loops in the digraph.

A procedure for "actively" certifying a proof would be similar, taking as a further argument a procedure for certifying inferences for which no acceptable certificate exists. Instead of simply failing when it cannot find an appropriate inference certificate, it invokes on the uncertified inference the creation procedure specified for active proof certification. Indeed, the passive proof certification would simply be this active certification procedure where the procedure for certifying unacceptable inferences simply fails. At the other extreme, if the active procedure is employed with the stipulation that the only acceptable inference certificates are those it creates itself during that run, then its success would entail certifying anew every inference reachable from the root of the digraph.

The above method depends on little about how inference engines work, but this account is deficient with regard to inferences that cite lemmas, and we cannot expect a uniform solution independent of the inference engines that might be supplied by FDL clients because they may have been based on differing choices about how they treat lemmas.

4.5.11.2 Inferences Citing Lemmas

When one cites a lemma as justification for an inference, correctness may depend not just upon "internal" syntactic relations between the conclusion and premises (if any), but also upon the correctness of the lemma. And of course, lemma citation between proofs must be well-founded (hence acyclic).

There is a subtlety here. In Proof Organization (section 4.5.11.1) we have given reasons of efficiency to make inference objects independent of the proofs they are part of, based on the assumption that inference checking dominates cost over proof organization, and we assumed inference step certification to be independent of proof certification. But when an inference cites one or more lemmas, then justifying the inference depends on justifying the entire proofs for those lemmas, so in order to maintain the economy of independent inference step certification we must allow inference step certification to fall short of full justification to the extent that lemmas are cited. We modify proof certification to require certification of lemmas cited by an inference just as it requires certification for proofs of children of an inference.

But how is the FDL process supposed to determine which lemmas are cited by an inference? Different **Inference Engines** supplied by FDL clients might treat lemmas rather differently. A fine grained solution is possible if the inference engine reports which lemmas it depends on for a step; in that case the certification procedure invoking the engine can store in the certificate the list of lemmas as reported by the engine. If, on the other hand, the possible lemmas were to be supplied by the FDL process to the inference engine as part of certification, then again these can be listed as part of the certificate. A coarser solution would be to assume that any proof referred to by the justification part (section 4.5.11.1) of the inference content might be cited as a lemma and so must be certified in the course of certifying the citing proof.

A second subtlety is implicit in the proof certification procedure extended to lemma citation as described above. Although with "normal" proof methods, one expects the cited lemmas to be proved by the same methods, some systems of proof may not be "uniform," i.e. they may require that certain lemmas cited in certain inferences be proved by a more restrictive method. This means that in order for the FDL process to certify a proof citing lemmas, one must know how to certify the lemma, which in turn means that along with each lemma cited in an inference step certificate, one may need to indicate the certification criteria to be used for the lemma. Of course, the default is just to use the same criterion as for the citing proof.

This same mechanism can be adapted for combining proof methods that are not compatible at the level of individual inference, and so could not participate together in the sort of hybrid proof mentioned in Proof Organization (section 4.5.11.1).

Another interesting distribution of verification involves shifting some lemma certification dependencies into the records of how the inference engine was built. It could be practical to create certain inference engines containing a specification of a large ground set of lemmas that were certified by the time the engine was created; this would obviate the need, when the engine is applied, to check those lemmas explicitly (or to mention them in the inference certificate) because: the certification of the inference refers to a process identity certificate (section 4.7) for the engine, which in turn refers to the certifications of its base set of lemmas; consequently the whole chain of certificates from that inference down to each base lemma can be assumed to remain intact.

4.5.11.3 Proof Sentinels

If one were building inferences according to a single uniform logic processed by a single inference engine, then assembling proofs would amount to collecting inferences into a dag and aligning conclusions of some inferences with premises of their parent inferences; all inferences may enter into proofs. But in a collection of inferences that may employ a variety of logics and inference engines that cannot be simply combined, collection into proofs requires something else, and recognizing when proofs or inferences are acceptable for a given purpose or by a given person requires further accounting for acceptability of inferences according to how they were validated. These are the purposes of "proof sentinels;" the connotation of "sentinel" is that it guards against the intrusion of untrusted entities into an inference. A logic can be represented by sentinel expressions that certify its inferences.

We shall temporarily make one oversimplifying assumption while we discuss the content and use of sentinels, namely, that all the inference steps in a proof dag must have certificates containing the same sentinel. Thus, a sentinel is used as the a standard of coherency between different inferences.

A sentinel is an expression, and we sometimes emphasize this by calling it a sentinel expression. The sentinel is intended to represent a class of basic logical resources and methods as opposed to derived ones. For example, one might build an inference engine which takes as a parameter a primitive rule set; then a sentinel expression appropriate to inference certificates invoking this engine would indicate the kind of inference engine invoked along with the primitive rule set. The presumed external meaning of these inference certificates is that the engine restricts its dependency upon primitive rules to those mentioned in the sentinel (plus, perhaps, some "built-in" rules that need not be mentioned explicitly in the sentinel expression).

The sentinel is *not* a general indication of resource restriction. Inferences citing lemmas, or definitions, or derived rules or, in the case of tactic provers, sources for tactic code, do not require extension of the sentinel. Resources can be developed and employed based upon a sentinel without altering the sentinel. Thus, while one may wish for some reason to restrict the resources actually used by a particular inference certification, and while one may want to make a record of resources actually used in a particular inference certification, neither of these should be effected by choice of sentinel. The purpose of the sentinel is to express a common basis for acceptable inference (which may differ on different occasions).

Another part of the sentinel expression is an indication of when an inference engine itself is acceptable. While one could build a sentinel that depends upon a particular running inference engine process, this would not normally be practical since one normally builds many inference engine processes, over the course of time according to each method for doing so. Therefore, it is more practical to build into the sentinel the method for building (or finding) individual inference engines of the appropriate kind. Indeed, the reason that an individual inference engine process is trusted in the first place is really that it was built according to certain methods, so there would be little point in stipulating a specific running process (rather than the method for creating one) as the engine for a proof sentinel.

The sentinel expression becomes the basic epistemic focus for users. The hard work on the part of a person or community of persons is coming to trust the basic inferences endorsed by a sentinel expression, and the FDL process should support this connection by retaining the user recognizable sentinel expressions in certificates for inferences. The intention is that persons become familiar with particular sentinels which they come to trust. Thus, it must be possible for a sentinel expression to be expressed briefly and be recognizable; and yet since in fact there will often be some complex structure to the meaning of the sentinel expression, this means that abbreviation of expressions is key.

Use and Structure of Proof Sentinels A sentinel expression is used to direct certification of inferences and assembly of proofs from inference steps, and occurs in records identifying inferences and proofs as having been certified accordingly. In its role as identifying a way to certify an inference, a sentinel expression must identify a method for finding or creating a suitable **Inference Engine** and it must provide any data (such as pointers to primitive inference rules) to be supplied to the inference engine in addition to the content of the **Inference Step**. These are specified as programs in a **Native Language** of the **FDL**.

In Proof Sentinels we supposed for the sake of explanation that all inference steps of a proof dag were certified according to some given sentinel; we now drop that supposition. In anticipation of the practice of extending a logic, we stipulate that a sentinel expression X determines which other sentinel expressions shall be accepted in assembling proofs according X, i.e. X inherits all the inferences passed by those other sentinels. A search for certificates according to a sentinel X should normally also find those certificates whose sentinels are inherited by X. Again, this can be specified by a program of the FDL's **Native Language**.

Typical motives for stipulating sentinel inheritance would be: implementation of an inference engine which is believed to be equivalent to an extant implementation; supplying an inference engine with more primitive resources (such as inference rules); the desire to explicitly unite distinct inference methods, simply inheriting them both, and specifying an order to try them when trying to create new certifications according to the union.

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 (section 4.5.11.1), either passively or actively, the sentinel expression determines whether the certificate for a 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. By allowing liberal use of packaging complex material into objects then referred to by object ids, and by allowing liberal use of **Native Language** macros, a suitable degree of abbreviation should be easily achieved.

4.6 The Initial Closed Map

The current closed map (section 4.5.2) is supposed always to be derivable from a simple generic closed map, the "initial closed map." The natural choice would be the empty map, but it may be convenient to build in certain objects distinguished by the FDL process for its operation. For example, the "certificate identifiers" introduced in Certificate Structure (section 4.5.6) serve to distinguish certificates from ordinary objects and to distinguish ways of executing **Native Language** programs, should there be more than one. And if such **Native Language** programs are expressed using particular operators identified by abstract identifiers, then those identifiers will also be included in the initial closed map.

Another obvious opportunity for deploying abstract identifiers is in the Pro-textual Constituents (section 4.4.5), which constitute the non-term constituents of texts – the <kind> part of the pairs <value>:<kind> that mediate the injection of these values into the **Text** syntax would be naturally filled by an abstract identifier since its role is simply to stipulate the kind of value that fills the <value> place and has no structure. The argument for using an abstract identifier for the <kind> is in the domain of multiple FDL implementors rather than multiple clients. From time to time one may expect extensions to be made to the kinds of pro-textual values incorporated into texts, and different FDL implementors might independently extend their systems either incompatibly, leading to the usual expensive name collision that clients might not care to pay for when porting from one FDL to the other, or it might lead to a merely tedious divergence of kind names that would be more happily identified.

Plainly the use of abstract identifiers for pro-textual value kinds would easily avoid these problems. But this means that these kind name identifiers must be in the current map in order to form texts that use them, and so must be in the initial closed map since they cannot be created as part of the ordinary FDL operations on closed maps.

The initial closed map, modulo identifier renaming, is part of what must be explained to an FDL **Client**. Or more precisely *some* initial closed map adequate for that client must be explained, since it is conceivable that a given client might be adequately served by a proper submap of the initial closed map. That this is possible is implicit in the possibility of extending an FDL implementation to include some new distinguished objects, such as new kinds of pro-textual values as described above; assuming the client was doing just fine before the new elements were implemented they would have done just as well had these elements already been implemented and simply not been exposed to the client.

4.7 Processes as Owners, Process Certificates, and Engine Stables

We use the term "process" here informally. Individual processes are identified by various criteria, and proceed through states, with some sort of integrity attributed to the series of states constituting the process. We shall take processes as the entities that one might trust or distrust. At one extreme we may regard persons as processes. More pertinently we may regard as a process the activities involving persons, equipment and institutions directed at maintaining an FDL. Other processes, well-defined and of shorter duration, are those implemented as computer processes in the usual sense. In all these cases, one process may create and track the identities of other processes, and may intermittently communicate with such processes.

Process Certificates and Process Identity Certificates Certificates (section 4.5.1) generally are owned by an FDL, i.e., by the process maintaining the FDL repository. This means that the creation and alteration of certificates is done exclusively by that process. The FDL (process) promises to create and alter certificates according to specific procedures associated with certificate kinds. One policy that could be adopted by an FDL for some kinds of certificates is to assign exclusive modification rights to a subprocess in perpetuity. Let us call such certificates "process certificates." Once a process terminates, the contents of process certificates assigned to it are frozen, and they become faithful records of past subprocesses.

To track and remember process identity, when the FDL builds a subprocess it could create a "process identity" certificate containing information about its construction, and assign it as a process certificate to the subprocess; the subprocess might further create its own process certificates. In any records that one cared to develop referring to the subprocess, this reference could be effected by object reference to its process identity certificate.

A major kind of subprocess of an FDL is the establishment of a "**Session**" for developing a current closed map (section 4.5.2). Another major kind of subprocess involved in the development of a logical library (section 4.3.1) is an "inference engine" for certifying inferences according to some specific criterion. Managing this process becomes interesting in light of the practical fact that establishing a process capable of performing such inferences can be expensive and we shall typically want to establish an inference engine then use it repeatedly for many inferences. If one were simply to build in a single inference engine into the FDL as a permanent vehicle, then there would be no particular complexity in using it and accounting for its use - certifying an inference would simply be calling the built-in inference procedure (which is *not* to say that accounting for its *correctness* is simple). But that simple scenario has little relation to the intended deployment of the FDL. In fact, we expect to manage multiple inference engines, each with complex state, as a matter of course, and indeed we expect this to involve building external processes as needed, and communicating with them repeatedly while they endure.

Stables of Engines We imagine the FDL process maintaining a "stable" of these workhorses. Let us suppose that the FDL creates a process identity certificate for each of these engines indicating how it was established (the external significance (section 4.5.4) typically being that an engine built in a certain way recognizes an understood class of inferences). Then when a request for verification of an inference arises, the desired sort of engine is determined and either an appropriate member of the stable is put to work, or the stable is further developed by building a new engine of the required sort. It should be expected that an engine would normally be built by constructing an external process, and then communicated with by appropriate protocols; the internal FDL subprocess would remember how the external process was established and how to communicate with it in order to effect inferences which the internal subprocess, as a subprocess of the FDL, then certifies.

4.8 Sharing Formal Mathematics

Here we discuss how two systems implementing formal logics might share theorems, proofs, or definitions. These are general methodological rather than technical considerations.

The Form of the Problem The form of the problem we consider is how to produce a candidate for a proof in a "recipient" system that borrows results from a "donor" system, and how one would argue for the result's correctness in the recipient's logic. We use "logic" in a sense that two systems can share a common logic but with some different primitives, including axioms. A logic in this sense characterizes not a specific class of theorems or inferences but rather proof structure. When multiple systems serve as mutual donors we would have a hybrid system proper.

We approach the problem from two directions: the Forms of Sharing Math and What Math can be Shared (section 4.8.2).

4.8.1 Basic Forms of Borrowing Formal Mathematics

The forms we propose as basic are the following:

1. Direct incorporation of results when the donor logic is also the recipient logic (modulo name collision and accounting for base differences such as axioms)

2. Informal inference from one system to another – Here one selects entities of one system, or translations of them, and takes them as new primitives in the recipient system, their occurrences in the old system standing as mere informal justification of their plausible adoption. This uses the same mechanism as simply stipulating new primitives, indicating informal reasons as part of the stipulation. This is the special case of making some informal appeal to another independent formal development (it could prevent cycles of such informal justification, though).

3. Direct or translated formal inference across systems (perhaps some offerings are rejected by the recipient)

4. Formal appropriation – Rather than selecting entities to be accepted informally, one directly justifies them or their translations, using the donor's proof not as evidence but as advice on how to establish the results by the recipient's own methods.

Discussion of Basic Forms of Borrowing We presuppose adequate accounting methods for possible differences in logical bases such as axioms and logics. Forms (1) and (2) are both simpler conceptions than the others, and we take them as more basic.

Forms (1), (2) and (4) are logically light-weight methods, whereas (3) can require substantial semantic or proof theoretic work.

Forms (1), (3) and (4) are all directed at extending the formal methods formally, systematically preserving validity of results in the recipient logic. One argues generally that the result is correct based upon the correctness of the recipient logic and the method for borrowing (which in (3) requires argument for correctness of the donor logic).

Form (2) on the other hand, is an *ad hoc* extension which essentially amounts to adopting new primitives. The cross system dependency is purely informal, but explicit. The argument for validity depends upon informal argument for the acceptance of new primitives, the references to the donor sources serving merely as elements of informal arguments for plausibility.

Form (1) is the most obvious method, amounting to accounting for different primitives that the donor and recipient might use, but otherwise introducing no significant commitment to theory or computation beyond the logic. Note that there is no automatic guarantee that combining primitives is consistent, hence the need for accounting.

Forms (3) and (4) can both be seen as methods for rehabilitating method (2) to eliminate the *ad hoc* arguments for plausibility of the borrowed entities. Form (3) is likely to be theoretically heavy but computationally light; (4) is likely to be computationally heavy but theoretically light.

While (3) produces proofs in the recipient system whose justification is based upon the logic (semantics or proof theory) of the donor system, (2) and (4) are not deductively based upon the donor logic at all, but simply treat the occurrences in the donor logic as plausible suggestions, which in the method of (4) are filtered automatically down to valid borrowings, and in (2) are filtered provisionally admitting possibly invalid borrowings.

Forms (3) and (4) may give rise to what might be considered hybrid "proof systems," the former comprising a hybrid "logic" and the latter a hybrid "system" for combining proofs without any inter-logic negotiations.

These considerations lead us to consider What Math can be Shared.

4.8.2 What Formal Mathematics can be Borrowed

Another dimension of borrowing besides the Forms of Sharing Math is what kind of data gets borrowed.

Propositions (Inference-free) This is the least intimate form of borrowing. Essentially what one borrows is whole statements of theorems. The "inference-free" qualifier is meant to distinguish those forms of **Proposition** (assertoric content) that "precede" formal inference system design, as opposed to those forms, such as sequents, designed as aspects of formal systems. (Presumably these latter forms supplement or elaborate those antecedent forms in order to facilitate argument pertinent to them.)

The most natural sort of propositional borrowing would be simply to take proved propositions (or their translations) from the donor system as true in the recipient system. This is pretty straightforward when the donor and recipient have the same logic or are logics of different kinds but justified by commensurable semantics of propositions.

Inferences and inference-system-specific propositional forms It may be possible to borrow the more logic-specific inference-laden forms of proposition, such as sequents. Of course, it is also possible that two logics attach distinct semantics to these kinds of propositions, and they might require translation or even be incommensurable.

If one can borrow the inference-system-specific forms of **Proposition**, one may also be able to borrow **Inference Steps**, consisting of conclusion/premise complexes of these propositions. Or one may not, since there may be some insurmountable distinction about the semantics of these inferences with premises, since the meaning of an inference with premises may require a special relation between the conclusion and premises beyond the mere fact of logical consequence. (For example, the inference may involve "metavariables" whose semantics involves instantiation across premises and conclusions.)

If one can borrow inferences then one can borrow whole **Proofs** or derivations as inference trees (or dags).

Justifications of inferences Even if two systems may accept each others inferences, they might be unable to make sense of the "justificatory" data supplementing inferences which help make them them recognizable as instances of a kind. Significant sharing of **Tactic** code, for example, can be expected to be quite rare among systems not arising from the same origin. A more likely sharable form of justification would be instances of low-level inference rules that can be independently interpreted by other systems. Thus a donor tactic system ought to be able to provide the "primitive proofs" instead of its tactic code on demand for justifications.

Of course, even if the justificatory data are not literally meaningful to the recipient system, they may yet serve as heuristic data for the method of formal appropriation described in Forms of Sharing Math (section 4.8.1) labeled (4).

Can definitions be borrowed? Certainly among systems with the same conception of **Definition**, sharing might be tractable, but it might be less likely than one supposes. The characteristic features of a(n eliminable) symbolic definition are that (1) it stipulates how some syntactic element can be eliminated from expressions (within the definition's "scope") without change of "denotation" of those expressions, and (2) the definition does *not* count as an extension of the epistemic basis for claims whose meanings depend on that definition. To use such a definition requires only that one understands it as such, and does not require persuading oneself of the meaningfulness or truth of a new primitive; it has more the derived character of a theorem than the primitive character of an axiom.

It is possible that two logics' notion of definition disagree enough on some aspects that some problematic definitions, rather than being translated into definitions in the recipient logic, would have to be employed as part of a translation from the donor to eliminate the defined syntactic elements from propositions.

4.9 Scenarios of FDL Use.

These scenarios are intended to prime the pump of imagination about what we think can be achieved (for a start), as well as to suggest the FDL design that supports them.

These scenarios were generated by a combination of recalling some that have been discussed among our group before, as well as reading through these notes and producing scenarios they rather directly suggest; they seem obvious to us because of prior discussions, a certain familiarity with the use of the FDL concepts, and our experiences in the actual production and manipulation of formal and informal content.

1 A reader wants to find what material there is in a library concerning a subject or a particular idea. They start by looking for keywords, which builds a collection of informal documents that mention the keys, and a collection of concise titles and paraphrases of formal objects that mention the keys. The reader then conducts further searches which can be based upon the formal structure of objects now discovered to be relevant.

2 A student wants to read supplemental material because the course text is unclear or obviously wrong on some point.

3 A teacher wants to ascertain the suitability of some material in the FDL as class material.

4 A teacher wants to reorganize some material for accessibility by a class, leaving the formal material alone. This person is contributing a "reading," and it can be made easily available to others.

5 A teacher wants to modify or add some formal material to an extant body, either to fill in pedagogical lacunae or to give variants of formal objects more appropriate for the students; the new formal material is as trustworthy as the old assuming it uses the same logical basis.

6a A programmer wants a precise explanation of a standard (perhaps common, perhaps obscure) algorithm to know why it works in order to implement a variant of it. To her delight, 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 she finally sees what she was missing.

6b A user (or manager or programmer) learns that there are documents in the FDL about the MediaNet distributed computing system for video. Examination reveals pertinent formal definitions and proofs about the schedule checker for quality of service, constituting conclusive documentation grounded in the formal material. A particularly helpful experience involves understanding a “stream transformer.” Because the transformer function was defined constructively it can be run on examples by an interpreter invoked by the FDL, ie, there is a “dynamic model” of the stream transformer available from the FDL based on the constructive content.

6c A programmer must modify a protocol in the event processing protocol stack of a system, due to changes in the upstream processes. The formal documentation of this protocol in the FDL system/theory entry for the stack states assumptions about the input stream and invariants of the protocol. It is not clear to the programmer that the new message stream will satisfy the input assumptions. Further investigation of the upstream process, also in the FDL, reveals that the assumption is violated in a way that can be described by simple additional clause such as “or condition P holds.” The programmer asks to reexecute the proof that the invariant holds under the new assumption. The tactic is in the FDL and it is executed, revealing the need for an additional fact about streams of messages of a certain type T. This appears to be a general fact of mathematics. The programmer searches the FDL for facts about streams and finds the very fact needed to justify the proposed code modification, but the proof uses a different logic than the one the tactic is written for. The programmer asks for the fact to be translated into the tactic’s logic and then runs the prover on this fact which is established automatically. The programmer is able to make a simple modification that is provably correct even though he or she does not understand formal logic.

7 A mathematician wants to publish the definitive verified version of some tricky or tedious proof, making every detail available to the reader on demand. This mathematician finds that there are already proofs in the right style with the right mathematical basis, and finds not only the originator of the theorem, but also a reference to the "technical" creator. The mathematician is not proficient in the actual production of formally verifiable proofs, or at least not with the system he wants used, and so contracts the discovered "formal technician" to formalize his proof and have it posted to the FDL.

8 A person embarking on a formalization wants some ideas about how to start. They search for all theorems pertaining to a similar topic, and collect them for comparison. At best, someone has already made an adequate formulation, as demonstrated by the the elegance of the proofs that use those formulations. At second best, there are examples that are unsatisfactory, and so need not be re-attempted.

9 A formal proof technician wants to experiment with an unfamiliar logic used by another group, but does not want to have to learn the interface they use. Fortunately, the preparation system she normally uses has been interfaced to the FDL, and so has that of the other group. This means that the content and inference engines of the other group are accessible from her usual interface.

10a A crank finds that he can never get his proofs to check with any of the logics everyone else seems to be using. People ignore his proofs because he introduced proof methods they have not become interested in.

10b An amateur (or crank) manages to write a couple of proofs that do adhere to standards of proof widely used in the FDL, and one day while doing a formal search for lemmas useful in a new proof a mathematician comes across them and finds they are useful. It's okay that the person who wrote the proofs is an amateur (or crank).

11 A person who has developed or read a fair amount of material finds that a handful of authors are heavily represented among the FDL documents she uses, the lemmas cited, and programs incorporated. She then decides to survey all the material posted by any of the handful and has a process regularly advise her when new material is posted by those authors.

12a Many utilities are developed over time by various parties, for searching the FDL or allowing a user to modify and develop his content. These utilities are themselves contributions to the FDL, and can be employed and discussed.

12b Data mining for patterns of inference: an interested party applies a procedure to an extant body of formal proofs in the FDL that attempts to identify repeatedly used patterns of inference that should be repackaged as lemmas, derived rules or tactics.

13a A person wants to experiment with some variation on extant material, such as changing a definition or deleting a rule of inference. They find it easy to discover what changes as a result of dependencies on those definitions or rules. They post the variant to the FDL, where it can coexist with the original.

13b Transplanting. A person wants to try some proofs in a context other than the one they were developed in; perhaps they want to "transplant" some results from constructive type theory to a subset of higher-order logic. Variants are built by replacing several expressions and lemma citations specific to the old context by purported analogs from the new context, and the proofs are rerun (we think of this as transplanting the roots and seeing what thrives). Unsurprisingly, many of the proof variants check in the new context, since they used some rather high-level forms of reasoning. Surprisingly and informatively, a few proofs depended on some unexpected detail, and are studied.

14a A programmer wants evidence that a program they intend to download works, and can be applied to the data they intend. This might be direct evidence such as a proof, or it might be indirect, namely, a certification of the existence of a proprietary proof of a program whose source is unavailable (only the i/o specifications and compiled code are available; the source and proof, though in the FDL, are unavailable to this programmer). In this case the person must understand which logic the proof employs, and be persuaded of the trustworthiness of the FDL in regard to those proof methods having been adhered to.

14b A commercial concern 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 an FDL process trusted to implement its published policies for certifying proofs. The commercial concern would maintain its own private FDL of source and object code and proofs of correctness. The impartial FDL 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 FDL checked (then deleted).

15 It is discovered that an inference engine supplied by a group is defective. All certifications of proofs that were developed with that engine can be located and discounted. If the engine is repaired the inferences that depended on the old one can be retried with the new version.

16 A person has doubts about whether a particular inference method is correct, then happily discovers that someone else had the same doubts and has rechecked all the inferences made by the suspect method with an independent inference verifier. This is possible because a single inference can have attached to it multiple certificates of verification.

17a Two (or more) large FDLs are maintained by parties that learn they can trust each other's FDL 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 FDL is part of the certificate.

17b It is discovered that one institution maintaining an FDL has not been following the protocols that all of a group FDL maintainers agreed to in order to promote federation. Because certifications passed from one FDL to another are recorded as such, they can be located, and all things that depended on them can be identified and scrutinized (and hopefully recertified).

18 A whole FDL's content is replicated into a new FDL because of inadequacies in accessing the original or some threat to its existence.

19 A person decides to introduce a concept, by definition say, but it turns out that someone else has already used the nomenclature they wanted to use. Because such nomenclature is extrensic to the formal material, and the attachment of nomenclature is performed as part of the connection of the user to the FDL, the user simply changes the old name to something she prefers, and adopts it for her newly defined concept. All the formal material using the old name survives the renaming because it was independent of the nomenclature.

20 A person finds a few formal objects of particular interest for a particular purpose, such as an algorithm and a specification and proof of correctness. They decide to collect into a single compact sublibrary just what is necessary to support those few formal objects. This is a basic FDL operation.

21 A person wants to develop some material "offline" at an independent site for a while. The procedure is to build a new temporary FDL (on their laptop, say) extracting the material of interest from a larger FDL. Later, after developing the material locally, that material can be posted back to the large FDL from which it came (or another one). This is possible because name collision is systematically avoided in FDLs by an abstract use of identifiers.

22 (a more speculative scenario, but with some basis in experience) A person wants to formalize a standard text or a chapter thereof not only to get the general benefits of formalization, but because as the original author proceeded, the proofs get progressively sketchier owing to copious allusions to the structure of earlier proofs, and the person wants to work those sketches out and save them. They are relieved to find that much of the basic generic math the text depends on is already in the FDL and can be immediately incorporated, and that using potent tools for manipulating proofs as data makes it easier than expected to realize those "proofs by allusion."

Chapter 5

Selections from the FDL Manual

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 MURI research project funded by the Office of the Secretary of Defense and managed by the Office of Naval Research entitled

Building Interactive Digital Libraries of Formal Algorithmic Knowledge.

A key purpose of the prototype library is to experiment with the type of system with 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. We have been adding services to the FDL and experimenting with them during the period from May 2002 until now.

The experimental 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 Coq, HOL, Isabelle, Minlog, 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. The FDL is accessible from our project Web site at www.nuprl.org/html/Digital_Libraries.html

5.1 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.

5.1.1 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.

5.1.2 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.

5.1.3 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 [165]. The President's commission on critical infrastructure protection and the PITAC report reached the same conclusions [155]. 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 [155] 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 [129, 160]. 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 [44]. In particular one can learn about a particular system in a context where the design, the specifications, the algorithms and 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 [143], 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 [87]. 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.

The use scenarios for the formal digital library suggest that its design be open in several dimensions. 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.

5.1.4 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.

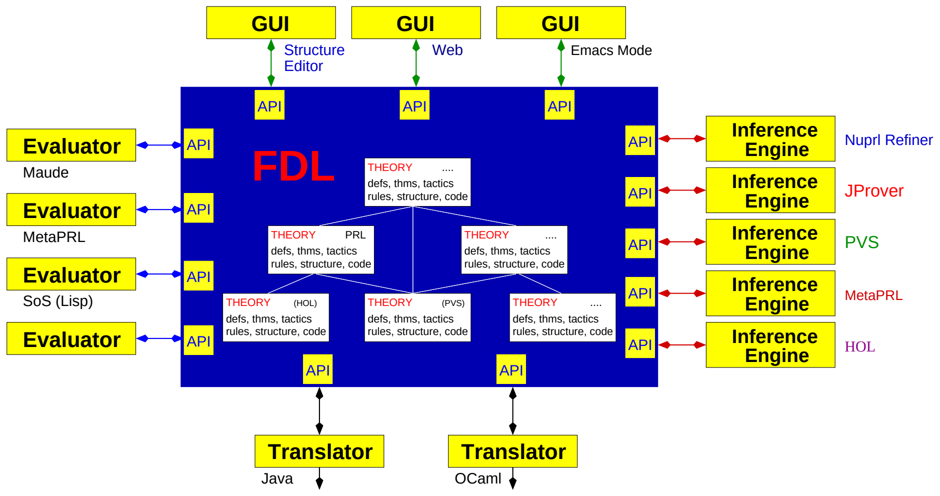


Figure 5.1: Interaction between the FDL and its clients

5.1.5 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 5.2.1). 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 5.2.2) in a similar way complex proof techniques are built from from basic inference rules and tacticals.

We build *closed maps* (Section 5.3) representing the work space for client sessions on top of the basic operations and logical *accounting and security* Section 5.4) on top of that.

5.1.6 FDL Prototype Circa 2003

Figure 5.1 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 [48, 5], MetaPRL [127], PVS [148], SPECWARE [169], HOL [69], Coq [58], Isabelle [150], or Ωmega [25], first-order provers like JProver [164], Otter [180], EQP [122], or Setheo [108], proof-based program generators like MinLog [23], rewrite engines like Maude [42], computer algebra systems [179, 117], decision procedures [140, 167, 171], and model checkers [125, 56, 88]. 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.

5.1.7 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.

5.2 Library Data and Operations

5.2.1 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.

5.2.2 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.

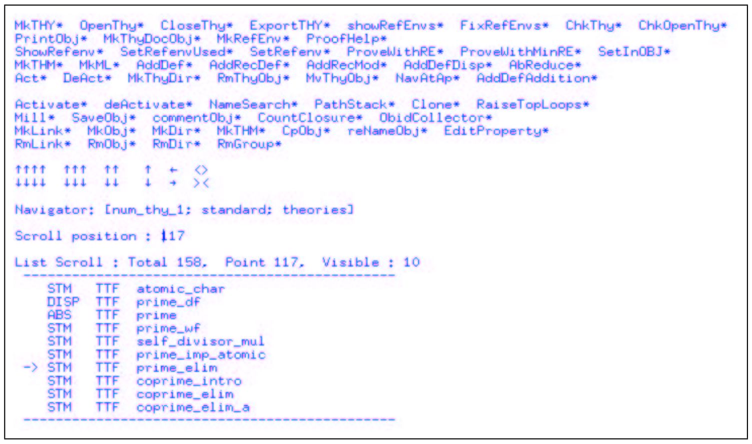
In contrast to the primitive library operations described in Section 5.1.5, basic services describe the interaction with the client through the client's current closed map, although some of the also affect the contents of the library itself.

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.



```
MkTHY*  OpenThy*  CloseThy*  ExportTHY*  showRefEnvs*  FixRefEnvs*  ChkThy*  ChkOpenThy*
PrintObj*  MkThyDocObj*  MkRefEnv*  ProofHelp*
ShowRefenv*  SetRefenvUsed*  SetRefenv*  ProveWithRE*  ProveWithMinRE*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*  NavAtAp*  AddDefAddition*

Activate*  deActivate*  NameSearch*  PathStack*  Clone*  RaiseTopLoops*
Mill*  SaveObj*  commentObj*  CountClosure*  ObidCollector*
MkLink*  MkObj*  MkDir*  MkTHM*  CpObj*  reNameObj*  EditProperty*
RmLink*  RmObj*  RmDir*  RmGroup*

↑↑↑↑  ↑↑↑  ↑↑   ↑  ←  <>
↓↓↓↓  ↓↓↓  ↓↓   ↓  →  ><

Navigator: [num_thy_1; standard; theories]

Scroll position : 117

List Scroll : Total 158,  Point 117,  Visible : 10
 ---------------------------------------------
    STM   TTF  atomic_char
    DISP  TTF  prime_df
    ABS   TTF  prime
    STM   TTF  prime_wf
    STM   TTF  self_divisor_mul
    STM   TTF  prime_imp_atomic
 -> STM   TTF  prime_elim
    STM   TTF  coprime_intro
    STM   TTF  coprime_elim
    STM   TTF  coprime_elim_a
 ---------------------------------------------
```

  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.

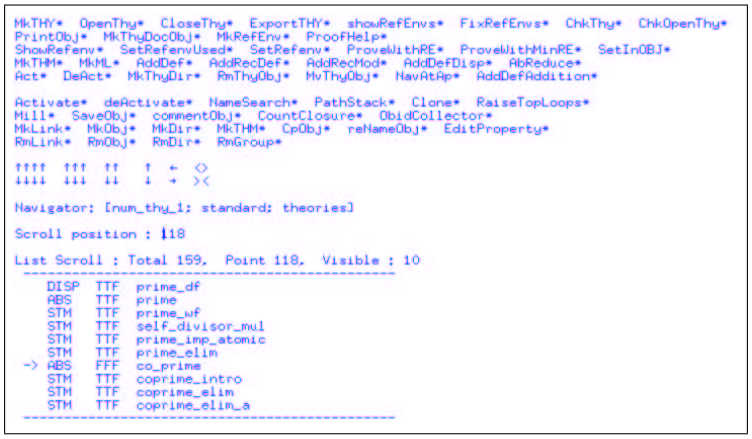
- *Naming and creating objects* is a combination of two, more fundamental operations. In the first step, a function `mk_obj` creates a default object of a given kind and adds it to the library table. The object will be bound to a new object identifier, which will be returned as the result of the function.

  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.



```
MkTHY*  OpenThy*  CloseThy*  ExportTHY*  showRefEnvs*  FixRefEnvs*  ChkThy*  ChkOpenThy*
PrintObj*  MkThyDocObj*  MkRefEnv*  ProofHelp*
ShowRefenv*  SetRefenvUsed*  SetRefenv*  ProveWithRE*  ProveWithMinRE*  SetInOBJ*
MkTHM*  MkML*  AddDef*  AddRecDef*  AddRecMod*  AddDefDisp*  AbReduce*
Act*  DeAct*  MkThyDir*  RmThyObj*  MvThyObj*  NavAtAp*  AddDefAddition*

Activate*  deActivate*  NameSearch*  PathStack*  Clone*  RaiseTopLoops*
Mill*  SaveObj*  commentObj*  CountClosure*  ObidCollector*
MkLink*  MkObj*  MkDir*  MkTHM*  CpObj*  reNameObj*  EditProperty*
RmLink*  RmObj*  RmDir*  RmGroup*

↑↑↑↑  ↑↑↑  ↑↑   ↑  ←  <>
↓↓↓↓  ↓↓↓  ↓↓   ↓  →  ><

Navigator: [num_thy_1; standard; theories]

Scroll position : 118

List Scroll : Total 159,  Point 118,  Visible : 10
 ---------------------------------------------
    DISP  TTF  prime_df
    ABS   TTF  prime
    STM   TTF  prime_wf
    STM   TTF  self_divisor_mul
    STM   TTF  prime_imp_atomic
    STM   TTF  prime_elim
 -> ABS   FFF  co_prime
    STM   TTF  coprime_intro
    STM   TTF  coprime_elim
    STM   TTF  coprime_elim_a
 ---------------------------------------------
```

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.

5.2.3 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 5.4.1). 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 5.3), 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 5.4.2). 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.

5.2.4 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 5.3.1).
- 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 5.3.3)
- 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.

5.3 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.

5.3.1 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 $m \in D \to \mathtt{Term}(D)$ to a subclass $X \subseteq D$ is itself a closed map (i.e. is in $X \to \mathtt{Term}(X)$), 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 $m \in D \to \mathtt{Term}(D)$ and $m' \in D' \to \mathtt{Term}(D')$ 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 $t \in \mathtt{Term}(D)$ *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.

5.3.2 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 5.4.2). 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.

5.3.3 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.

5.4 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.

5.4.1 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.

5.4.2 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 5.3). 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 5.6.1) 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.

5.4.3 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.

5.5 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 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 [55], i.e. to the principles of *atomicity*, *consistency*, *isolation*, and *durability* (*ACID*).

The FDL prototype is centered around the *library table* (Section 5.2.1). 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 5.5.2) 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 5.5.3) 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 5.5.4) that enable users to interact with the library and to develop new formal content for it.

5.5.1 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

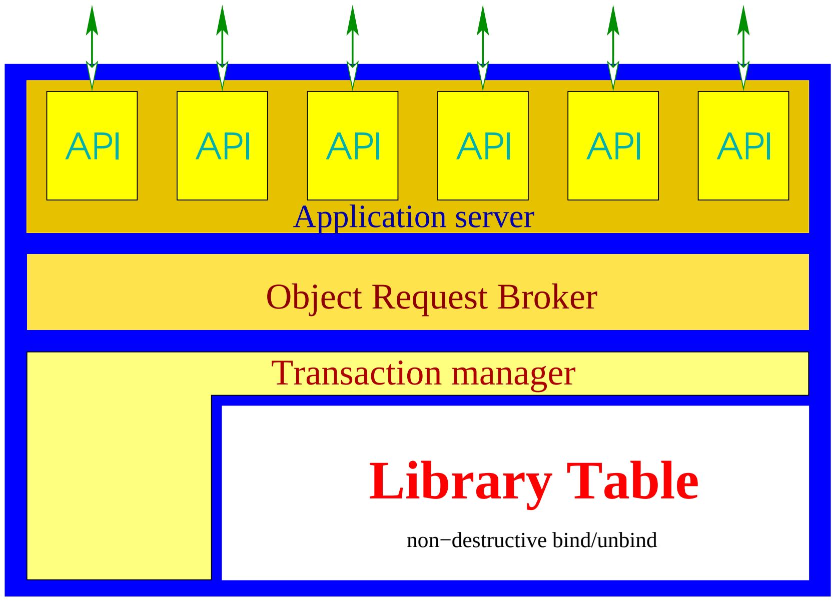


Figure 5.2: Architecture of the FDL prototype

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.

5.5.2 Transactions

Transactions are a well-established technique to ensure the ACID property of data bases [55]. 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.

5.5.3 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 5.2.1), which have to obey the grammar described in Table 5.1. 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

```
    <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 5.1: Grammar for requests, responses, and notices

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 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] [121], which currently is used in many connections between the FDL and proof engines (see Section 5.6.1). 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 [64] and HELM [76] 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 [168, 170] for an introduction into writing such modules).

5.5.4 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 5.2.2). The editor can interpret library contents and can be customized by adding display forms to the library.

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

A *web interface* processes library information for publication on the web (Section 5.7). 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 5.6.1.

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 5.6.2.

5.5.5 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 [5, 142, 120] 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 [148, 158] 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 [127] 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 5.6.2) we will integrate this framework into the FDL library.

5.6 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.

5.6.1 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 [5, 142] 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 [102, 164] is a complete theorem prover for intuitionistic and classical first-order logic. Its proof search procedure [146, 147, 101] is an extension of *matrix methods* developed by Andrews and Bibel [11, 27]. 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 [163, 103], 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 [148, 158] 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 [69], the proof-based MinLog [23] program generator, Isabelle [150], and even Larch [106] and Automath [34].

5.6.2 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 [148], Nuprl [5], MetaPRL [127], MinLog [23], HOL [69], Coq [58], and Isabelle [150]. 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 [5, 142], the structure of Nuprl 5 terms is similar to the one of basic FDL data (Section 5.2.1). Migrating Nuprl 5 theories into the FDL can therefore be based on the mechanisms for exporting and importing theories described in Section 5.2.2, which only have to rebuild certificates.

- MetaPRL [77, 127] 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 [148, 158] 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.

5.7 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 [136] 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.

Chapter 6

Logical frameworks and the FDL

6.1 Introduction

A *logical framework* is a system that allows the definition and use of *multiple* logics, supporting derivation in any of the logics that are defined. We take the term logic in its general meaning. A logic may characterize a very large domain, like arithmetic, or it may characterize a smaller domain, like compilation, abstract algebra, graph theory, or any other computational, algorithmic domain. The MetaPRL logical programming environment is an augmented logical framework supporting reductions and relations between logics, as well as the embedding of one domain within another. MetaPRL is a modular system, organizing and collecting knowledge in a hierarchy that corresponds closely to standard practice in software engineering.

The FDL is a companion in this enterprise, because it captures, stores, and indexes the knowledge from these domains regardless of the proof system which produced it. This allows a programmer to search for knowledge and algorithms from any of the contributions to the library. The FDL and MetaPRL provide an instance of systems that cooperate to provide the programmer with a rich source of algorithmic knowledge, as well as a state-of-the-art means to organize and incorporate that knowledge into software systems.

We have experience using MetaPRL to host and support a compiler. Elements of the compiler are proved correct in the CTT logic of MetaPRL, and we will discuss this process here. This compiler support is one concrete example of how the FDL and a proof system provide a context for using FDL knowledge to support a specific system. The compiler work is relatively new, but published. We have not yet used FDL knowledge outside of that produced by MetaPRL itself. However, we have used the FDL and MetaPRL in a cooperative effort to verify a protocol. The protocol work is not yet ready for publication, so we have chosen to illustrate the points with the compiler effort.

In this chapter, we describe the development of a new class of algorithms using the MetaPRL/FDL system. Our goal has two parts: 1) to contribute new algorithmic content to the FDL, and 2) demonstrate how the logical framework is used to organize and develop the knowledge during the software development process. We organize this as a case study for the development of several important, fundamental algorithms that are widely used in the programming language and compiler community. We develop this case study as far as an implementation of a complete compiler that uses our algorithms

The text for this chapter is generated from source code of this project. That is, this document is part of the formal content that is stored and generated as part of the FDL. By doing so, we connect the documentation directly to the source that provides the algorithmic knowledge in the FDL.

6.1.1 Compilers and programming languages

One of the standing challenges to the programming language community is the problem of compiler validation. The task of designing and implementing a compiler can be difficult even for a small language. There are many phases in the translation from source to machine code, and an error in any one of these phases can alter the semantics of the generated program.

There are two main reasons to validate the compiler. First, if there is an error in the compiler, then there is no assurance that the properties specified by the programmer are valid at runtime. Second, if the compiler is validated, it becomes possible to certify mobile code. That is, source-level proofs provided by the programmer can be translated automatically to runtime proofs that accompany the code and can be validated by the recipient.

In initial work (with a Caltech undergraduate) [16], we explored the feasibility of using a formal system to reason about the internal compiler representations for a program. More recently [79], we developed an alternative approach, based on the use of higher-order abstract syntax [141, 153] and term rewriting to construct an entire compiler in MetaPRL. All program transformations, from parsing to code generation, are cleanly isolated and specified as term rewrites.

There are many advantages to using a formal system. The most important is that the correctness of the compiler is dependent only on the rewriting rules that provide the formal part of the translation. The vast majority of compiler code does not have to be trusted. In our current work, the correctness of the compiler depends on less than 1% of its code. In addition, we find that in many cases it is *easier* to implement the compiler because the logical framework provides a great deal of automation.

Compilers make use of many fundamental algorithms used to transform and analyze programs. For example, CPS (continuation-passing style) transformation is a widely-used algorithm that transforms the control flow of a program. As Sabry and Wadler show [162], Plotkin's CPS translation [157], Moggi's monadic translation [130], and Girard's translation to linear logic [67] are all related; insight into any one of these translations provides insight into all three. While CPS transformation may be fundamental, it is an example of an algorithm that is difficult for humans to understand, and it is quite difficult to perform by hand. The payoff for a general formalization as part of the FDL is thus quite high.

Other widely-used algorithms include closure conversion (where functions in a program are lifted to top-level), code generation, register allocation, including spill selection. Instances of these algorithms are used widely is many areas of computer science, especialy for compilers for languages ranging from Lisp and ML, to Java, C, C#, etc. By defining them as part of the FDL, we provide a freely-available formal implementation that serves as a common reference for the many applications that use these algorithms.

Our approach is based on the use of higher-order abstract syntax [141, 153] and term rewriting in a general-purpose logical framework that uses the FDL. All program transformations, from parsing to code generation, are cleanly isolated and specified as term rewrites. In our system, term rewrites specify an equivalence between two code fragments that is valid in any context. Rewrites are bidirectional and neither imply nor presuppose any particular order of application. Rewrite application is guided by programs in the meta-language of the logical framework.

The correctness of the compiler is dependent only on the rewriting rules. Programs that guide the application of rewrites do not have to be trusted because they are required to use rewrites for all program transformations. If the rules can be validated against a program semantics, and if the compiler produces a program, that program will be correct relative to those semantics. The role of the guidance programs is to ensure that rewrites are applied in the appropriate order so that the output of the compiler contains only assembly.

The collection of rewrites needed to implement a compiler is small (hundreds of lines of formal mathematics) compared to the entire code base of a typical compiler (often more than tens of thousands of lines of code in a general-purpose programming language). Validation of the former set is clearly easier. Even if the rewrite rules are not validated, it becomes easier to assign accountability to individual rules.

The use of a MetaPRL and the FDL has another major advantage that we explore in this chapter: in many cases it is *easier* to implement the compiler, for several reasons. The terminology of rewrites corresponds closely to mathematical descriptions frequently used in the literature, decreasing time from concept to implementation.

In this chapter, we explore these problems and show that formal compiler development is feasible, perhaps easy using the MetaPRL/FDL system. This chapter is organized around a case study, where we develop a compiler that generates Intel x86 machine code for an ML-like language using the MetaPRL logical framework [78, 81, 85]. The compiler is fully implemented and online as part of the Mojave research project [83].

6.1.2 Organization

The translation from source code to assembly is usually done in three major stages. The parsing phase translates a source file (a sequence of characters) into an abstract syntax tree; the abstract syntax is translated to an intermediate representation; and the intermediate representation is translated to machine code. The reason for the intermediate representation is that many of the transformations in the compiler can be stated abstractly, independent of the source and machine representations.

The language that we are using as an example (see Section 6.2) is a small language similar to ML [173]. To keep the presentation simple, the language is untyped. However, it includes higher-order and nested functions, and one necessary step in the compilation process is closure conversion, in which the program is modified so that all functions are closed.

The high-level outline of the chapter includes the following sections: Section 6.2 Parsing, Section 6.3 Intermediate representation (IR), Section 6.4

Summary and future work, Section 6.5 Related work. Before describing each of these stages, we first introduce the terminology and syntax of the formal system in which we define the program rewrites.

6.1.3 Terminology

All logical syntax is expressed in the language of *terms*. The general syntax of all terms has three parts. Each term has 1) an operator-name (like "sum"), which is a unique name identifying the kind of term; 2) a list of parameters representing constant values; and 3) a set of subterms with possible variable bindings. We use the following syntax to describe terms:

$$\underbrace{\textit{opname}}_{\textit{operator name}}\ \underbrace{[p_1; \cdots; p_n]}_{\textit{parameters}}\underbrace{\{\vec{v}_1.t_1; \cdots; \vec{v}_m.t_m\}}_{\textit{subterms}}$$

Displayed form	Term
1	`number[1]{}`
$\lambda x.b$	`lambda[]{ x.  b }`
$f(a)$	`apply[] { f; a }`
$x + y$	`sum[]{ x; y }`

A few examples are shown in the table. Numbers have an integer parameter. The `lambda` term contains a binding occurrence: the variable x is bound in the subterm b.

Term rewrites are specified in MetaPRL using second-order variables, which explicitly define scoping and substitution [141]. A second-order variable pattern has the form $v[v_1; \cdots; v_n]$, which represents an arbitrary term that may have free variables $v_1, \ldots, v_n$. The corresponding substitution has the form $v[t_1; \cdots; t_n]$, which specifies the simultaneous, capture-avoiding substitution of terms $t_1, \ldots, t_n$ for $v_1, \ldots, v_n$ in the term matched by v. For example, the rule for β-reduction is specified with the following rewrite.

$$\text{[beta]} \qquad (\lambda x.v_1[x])\ v_2 \longleftrightarrow v_1[v_2]$$

The left-hand-side of the rewrite is a pattern called the *redex*. The $v_1[x]$ stands for an arbitrary term with free variable x, and v_2 is another arbitrary term. The right-hand-side of the rewrite is called the *contractum*. The second-order variable $v_1[v_2]$ substitutes the term matched by v_2 for x in v_1. A term rewrite specifies that any term that matches the redex can be replaced with the contractum, and vice-versa.

Rewrites that are expressed with second-order notation are strictly more expressive than those that use the traditional substitution notation. The following rewrite is valid in second-order notation.

$$[\text{const}] \quad (\lambda x.v[])\ 1 \longleftrightarrow (\lambda x.v[])\ 2$$

In the context λx, the second-order variable $v[]$ matches only those terms that do not have x as a free variable. No substitution is performed; the β-reduction of both sides of the rewrite yields $v[] \longleftrightarrow v[]$, which is valid reflexively. Normally, when a second-order variable $v[]$ has an empty free-variable set $[]$, we omit the brackets and use the simpler notation v.

MetaPRL is a tactic-based prover that uses OCaml [177] as its meta-language. When a rewrite is defined in MetaPRL, the framework creates an OCaml expression that can be used to apply the rewrite. Code to guide the application of rewrites is written in OCaml, using a rich set of primitives provided by MetaPRL. MetaPRL automates the construction of most guidance code; we describe rewrite strategies only when necessary. For clarity, we will describe syntax and rewrites using the displayed forms of terms.

The compilation process is expressed in MetaPRL as a judgment of the form $\Gamma \vdash \textbf{compilable}(e)$, which states the the program e is compilable in any logical context Γ. The meaning of the $\textbf{compilable}(e)$ judgment is defined by the target architecture. A program e' is compilable if it is a sequence of valid assembly instructions. The compilation task is a process of rewriting the source program e to an equivalent assembly program e'.

6.2 Parsing

In order to use the formal system for program transformation, source-level programs expressed as sequences of characters must first be translated into a term representation for use in the MetaPRL framework. We assume that the source language can be specified using a context-free grammar, and traditional lexing and parsing methods can be used to perform the translation.

MetaPRL provides these capabilities using the Phobos [70] lexer and parser. A Phobos language specification resembles a typical parser definition in YACC [93], except that semantic actions for productions use term rewriting. Phobos uses *informal* rewriting, which means that it can create new variable bindings and perform capturing substitution.

$$
\begin{array}{lll}
op & ::= & + \mid - \mid * \mid / \mid = \mid <> \mid < \mid \leq \mid > \mid \geq \quad \text{Binary operators}
\end{array}
$$

$$
\begin{array}{lllll}
e & ::= & \top \mid \bot & \text{Booleans} & \\
 & \mid & i & \text{Integers} & \\
 & \mid & v & \text{Variables} & \\
 & \mid & e\ op\ e & \text{Binary expressions} & \\
 & \mid & \lambda v.e & \text{Anonymous functions} & \\
 & \mid & e; e & \text{Sequencing} & \\
 & \mid & e.[e] & \text{Subscripting} & \\
 & \mid & e.[e] \leftarrow e & \text{Assignment} & \\
 & \mid & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e & \text{Conditionals} & \\
 & \mid & e(e_1, \ldots, e_n) & \text{Application} & \\
 & \mid & \mathbf{let}\ v = e\ \mathbf{in}\ e & \text{Let definitions} & \\
 & \mid & \mathbf{let\ rec}\ f_1(v_1, \ldots, v_n) = e & & \\
 & & \vdots & \text{Recursive functions} & \\
 & & \mathbf{and}\ f_n(v_1, \ldots, v_n) = e & &
\end{array}
$$

Figure 6.1: Program syntax

The lexer for a language is specified as a set of lexical rewrite rules of the form *regex* $\longleftrightarrow$ *term*, where *regex* is a special term that is created for each token with the the matched input as a string parameter. The following example demonstrates a single lexer clause, that translates a nonnegative decimal number to a term with operator name `number` and a single integer parameter.

$$\texttt{NUM} \ = \ \text{"}[\texttt{0} - \texttt{9}] + \text{"} \ \{\mathbf{token}[i]\{pos\} \longleftrightarrow number[i]\}$$

The parser is defined as a set of grammar productions. For each grammar production in the program syntax shown in Figure 6.1, we define a production in the form $S ::= S_1 \ldots S_n \longleftrightarrow term$ where the symbols S_i may be annotated with a term pattern. For instance, the production for the let-expression is defined with the following production and semantic action.

$$\texttt{exp} ::= \texttt{LET ID}\,\langle\texttt{v}\rangle\ \texttt{EQ exp}\,\langle\texttt{e}\rangle\ \texttt{IN exp}\,\langle\texttt{rest}\rangle \longleftrightarrow \mathbf{let}\, v = e\ \mathbf{in}\ rest$$

Phobos constructs an LALR(1) parser from the grammar specification, applying the appropriate rewrite rule when a production is reduced.

6.3 Intermediate representation

The intermediate representation of the program must serve two conflicting purposes. It should be a fairly low-level language so that translation to machine code is as straightforward as possible. However, it should be abstract enough that program transformations and optimizations need not be overly concerned with implementation detail. The intermediate representation we use is similar to the functional intermediate representations used by several

$$
\begin{array}{rcl}
\mathit{bop} & ::= & + \mid - \mid * \mid / \\
\mathit{rop} & ::= & \leq \mid < \mid > \\
 & \mid & \geq \mid = \mid <> \\
l & ::= & \mathit{string} \\
 & & \\
a & ::= & \top \mid \bot \\
 & \mid & i \\
 & \mid & v \\
 & \mid & a_1 \ \mathit{bop} \ a_2 \\
 & \mid & a_1 \ \mathit{rop} \ a_2 \\
 & \mid & R.l
\end{array}
\qquad
\begin{array}{rcll}
e & ::= & \mathbf{let}\, v = a \ \mathbf{in}\ e & \text{Variable definition} \\
 & \mid & \mathbf{if}\ a\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 & \text{Conditional} \\
 & \mid & \mathbf{let}\, v = \ (a_1, \ldots, a_n)\ \mathbf{in}\ e & \text{Tuple allocation} \\
 & \mid & \mathbf{let}\, v = \ a_1.[a_2]\ \mathbf{in}\ e & \text{Subscripting} \\
 & \mid & a_1.[a_2] \leftarrow a_3; e & \text{Assignment} \\
 & \mid & \mathbf{let}\, v = a(a_1, \ldots, a_n)\ \mathbf{in}\ e & \text{Function application} \\
 & \mid & \mathbf{letc}\, v = a_1(a_2)\ \mathbf{in}\ e & \text{Closure creation} \\
 & \mid & \mathbf{return}\ a & \text{Return a value} \\
 & \mid & a(a_1, \ldots, a_n) & \text{Tail-call} \\
 & \mid & \mathbf{let\ \ rec}\ R = d\ \mathbf{in}\ e & \text{Recursive functions} \\
 & & & \\
e_\lambda & ::= & \lambda v.e_\lambda \mid \lambda v.e & \text{Functions} \\
d & ::= & \mathbf{fun}\ l = e_\lambda\ \mathbf{and}\ d & \text{Function definitions} \\
 & \mid & \epsilon &
\end{array}
$$

Figure 6.2: Intermediate Representation

groups [12, 80, 172], in which the language retains a similarity to an ML-like language where all intermediate values apart from arithmetic expressions are explicitly named.

In this form, the IR is partitioned into two main parts: "atoms" define values like numbers, arithmetic, and variables; and "expressions" define all other computation. The language includes arithmetic, conditionals, tuples, functions, and function definitions, as shown in Figure 6.2.

Function definitions deserve special mention. Functions are defined using the **let rec** $R = d$ **in** e term, where d is a list of mutually recursive functions, and variable R represents a recursively defined record containing these functions. Each of the functions is labeled, and the term $R.l$ represents the function with label l in record R.

While this representation has an easy formal interpretation as a fixpoint of the single variable R, it is awkward to use, principally because it violates the rule of higher-order abstract syntax: namely, that (function) variables be represented as variables in the meta-language. In some sense, this representation is an artifact of the MetaPRL term language: it is not possible, given the term language described in Section 6.1.3, to define more than one recursive variable at a time. We are currently investigating extending the meta-language to address this problem.

6.3.1 AST to IR conversion

The main difference between the abstract syntax representation and the IR is that intermediate expressions in the AST do not have to be named. In addition, the conditional in the AST can be used anywhere an expression can be used (for instance, as the argument to a function), while in the IR, the branches of the conditional must be terminated by a **return** a expression or tail-call.

The translation from AST to IR is straightforward, but we use it to illustrate a style of translation we use frequently. The term `IR`$\{e_1; v.e_2[v]\}$ (displayed as $[\![e_1]\!]_{IR} v.e_2[v]$) is the translation of an expression e_1 to an IR atom, which is substituted for v in expression $e_2[v]$. The translation problem is expressed through the following rule, which states that a program e is compilable if the program can be translated to an atom, returning the value as the result of the program.

$$\frac{\Gamma \vdash \textbf{compilable}([\![e]\!]_{IR} v.\textbf{return } v)}{\Gamma \vdash \textbf{compilable}(e)}$$

For many AST expressions, the translation to IR is straightforward. The following rules give a few representative examples. Note that the `add` and `set` rules perform substitution, which is specified implicitly using higher-order abstract syntax.

$$\begin{array}{ll}
\text{[int]} & [\![i]\!]_{IR} v.e[v] \longleftrightarrow e[i] \\
\text{[var]} & [\![v_1]\!]_{IR} v_2.e[v_2] \longleftrightarrow e[v_1] \\
\text{[add]} & [\![e_1 + e_2]\!]_{IR} v.e[v] \longleftrightarrow [\![e_1]\!]_{IR} v_1.[\![e_2]\!]_{IR} v_2.e[v_1 + v_2] \\
\text{[set]} & [\![e_1.[e_2] \leftarrow e_3]\!]_{IR} v.e_4[v] \\
\longleftrightarrow & [\![e_1]\!]_{IR} v_1.[\![e_2]\!]_{IR} v_2.[\![e_3]\!]_{IR} v_3.v_1.[v_2] \leftarrow v_3; e_4[\bot]
\end{array}$$

For conditionals, code duplication is avoided by wrapping the code after the conditional in a function, and calling the function at the tail of each branch of the conditional.

$$\begin{array}{ll}
\text{[if]} & [\![\,\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3]\!]_{IR} v.e_4[v] \\
\longleftrightarrow & \quad \textbf{let rec } R = \textbf{fun } g = \lambda v.e_4[v] \textbf{ and } \epsilon \textbf{ in} \\
 & [\![e_1]\!]_{IR} v_1.\,\textbf{if } v_1 \textbf{ then } [\![e_2]\!]_{IR} v_2.(R.g(v_2)) \textbf{ else } [\![e_3]\!]_{IR} v_3.(R.g(v_3))
\end{array}$$

For functions, the post-processing phase converts recursive function definitions to the record form, and we have the following translation, using the term $[\![d]\!]_{IR}$ to translate function definitions. In general, anonymous functions must be named *except* when they are outermost in a function definition. The post-processing phase produces two kinds of λ-abstractions, the $\lambda_p v.e[v]$ is used to label function parameters in recursive definitions, and the $\lambda v.e[v]$ term is used for anonymous functions.

$$
\begin{array}{ll}
\text{[letrec]} & [\![\,\textbf{let rec}\ R = d\ \textbf{in}\ e_1]\!]_{IR} v.e_2[v] \longleftrightarrow \textbf{let rec}\ R = [\![d]\!]_{IR}\ \textbf{in}\ [\![e_1]\!]_{IR} v.e_2[v] \\
\text{[fun]} & [\![\textbf{fun}\ l = e\ \textbf{and}\ d]\!]_{IR} \longleftrightarrow \textbf{fun}\ l = [\![e]\!]_{IR} v.\textbf{return}\ v\ \textbf{and}\ [\![d]\!]_{IR} \\
\text{[param]} & [\![\lambda_p v_1.e_1[v_1]]\!]_{IR} v_2.e_2[v_2] \longleftrightarrow \lambda v_1.([\![e_1[v_1]]\!]_{IR} v_2.e_2[v_2]) \\
\text{[abs]} & [\![\lambda v_1.e_1[v_1]]\!]_{IR} v_2.e_2[v_2] \\
\longleftrightarrow & \ \textbf{let rec}\ R = \textbf{fun}\ g = \lambda v_1.[\![e_1[v_1]]\!]_{IR} v_3.\textbf{return}\ v_3\ \textbf{and}\ \epsilon\ \textbf{in}\ e_2[R.g]
\end{array}
$$

6.3.2 CPS conversion

CPS conversion is an optional phase of the compiler that converts the program to continuation-passing style. That is, instead of returning a value, functions pass their results to a continuation function that is passed as an argument. In this phase, all functions become tail-calls, and all occurrences of **let** $v = a_1(a_2)$ **in** e and **return** a are eliminated. The main objective in CPS conversion is to pass the result of the computation to a continuation function. We state this formally as the following inference rule, which states that a program e is compilable if for all functions c, the program $[\![e]\!]_c$ is compilable.

$$\frac{\Gamma, c\colon exp \vdash \textbf{compilable}([\![e]\!]_c)}{\Gamma \vdash \textbf{compilable}(e)}$$

The term $[\![e]\!]_c$ represents the application of the c function to the program e, and we can use it to transform the program e by migrating the call to the continuation downward in the expression tree. Abstractly, the process proceeds as follows.

- First, replace each function definition $f = \lambda x.e[x]$ with a continuation form $f = \lambda c.\lambda x.[\![e[x]]\!]_c$ and simultaneously replace all occurrences of f with the partial application $f[\textbf{id}]$, where **id** is the identity function.
- Next, replace tail-calls $[\![f[\textbf{id}](a_1,\ldots,a_n)]\!]_c$ with $f(c, a_1,\ldots,a_n)$, and return statements $[\![\textbf{return}\ a]\!]_c$ with $c(a)$.

- Finally, replace inline-calls $[\![\, \mathbf{let}\, v = f[\mathbf{id}](a_1,\ldots,a_n)\ \mathbf{in}\ e]\!]_c$ with the continuation-passing version $\mathbf{let\ rec}\ R = \mathbf{fun}\ g = \lambda v.[\![e]\!]_c\ \mathbf{and}\ \epsilon\ \mathbf{in}\ f(g, a_1, \ldots, a_n)$.

For many expressions, CPS conversion is a straightforward mapping of the CPS translation, as shown by the following five rules.

$$
\begin{array}{ll}
[\text{atom}] & [\![\, \mathbf{let}\, v = a\ \mathbf{in}\ e[v]]\!]_c \longleftrightarrow \mathbf{let}\, v = a\ \mathbf{in}\ [\![e[v]]\!]_c \\
[\text{tuple}] & [\![\, \mathbf{let}\, v = (a_1,\ldots,a_n)\ \mathbf{in}\ e[v]]\!]_c \longleftrightarrow \mathbf{let}\, v = (a_1,\ldots,a_n)\ \mathbf{in}\ [\![e[v]]\!]_c \\
[\text{letsub}] & [\![\, \mathbf{let}\, v = a_1.[a_2]\ \mathbf{in}\ e[v]]\!]_c \longleftrightarrow \mathbf{let}\, v = a_1.[a_2]\ \mathbf{in}\ [\![e[v]]\!]_c \\
[\text{setsub}] & [\![a_1.[a_2] \leftarrow a_3; e[v]]\!]_c \longleftrightarrow a_1.[a_2] \leftarrow a_3; [\![e[v]]\!]_c \\
[\text{if}] & [\![\, \mathbf{if}\ a\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]_c \longleftrightarrow \mathbf{if}\ a\ \mathbf{then}\ [\![e_1]\!]_c\ \mathbf{else}\ [\![e_2]\!]_c
\end{array}
$$

The modification of functions is the key part of the conversion. When a **let rec** $R = d[R]$ **in** $e[R]$ term is converted, the goal is to add an extra continuation parameter to each of the functions in the recursive definition. Conversion of the function definition is shown in the *fundef* rule, where the function gets an extra continuation argument that is then applied to the function body.

In order to preserve the program semantics, we must then replace all occurrences of the function with the term $f[\mathbf{id}]$, which represents the partial application of the function to the identity. This step is performed in two parts: first the *letrec* rule replaces all occurrences of the record variable R with the term $R[\mathbf{id}]$, and then the *letfun* rule replaces each function variable f with the term $f[\mathbf{id}]$.

$$
\begin{array}{ll}
[\text{letrec}] & [\![\, \mathbf{let\ rec}\ R = d[R]\ \mathbf{in}\ e[R]]\!]_c \longleftrightarrow \mathbf{let\ rec}\ R = [\![d[R[\mathbf{id}]]]\!]_c\ \mathbf{in}\ [\![e[R[\mathbf{id}]]]\!]_c \\
[\text{fundef}] & [\![\mathbf{fun}\ l = \lambda v.e[v]\ \mathbf{and}\ d]\!]_c \longleftrightarrow \mathbf{fun}\ l = \lambda c.\lambda v.[\![e[v]]\!]_c\ \mathbf{and}\ [\![d]\!]_c \\
[\text{enddef}] & [\![\epsilon]\!]_c \longleftrightarrow \epsilon \\
[\text{letfun}] & [\![\, \mathbf{let}\, v = R[\mathbf{id}].l\ \mathbf{in}\ e[v]]\!]_c \longleftrightarrow \mathbf{let}\, v = R.l\ \mathbf{in}\ [\![e[v[\mathbf{id}]]]\!]_c
\end{array}
$$

Non-tail-call function applications must also be converted to continuation passing form, as shown in the *apply* rule, where the expression *after* the function call is wrapped in a continuation function and passed as a continuation argument.

$$
\begin{array}{ll}
[\text{apply}] & [\![\, \mathbf{let}\, v_2 = v_1[\mathbf{id}](a)\ \mathbf{in}\ e[v_2]]\!]_c \\
\longleftrightarrow & \quad \mathbf{let\ rec}\ R = \mathbf{fun}\ g = \lambda v.[\![e[v]]\!]_c\ \mathbf{and}\ \epsilon\ \mathbf{in}\ \mathbf{let}\, g = R.g\ \mathbf{in}\ f(g; a)
\end{array}
$$

In the final phase of CPS conversion, we can replace return statements with a call to the continuation. For tail-calls, we replace the partial application of the function $f[\mathbf{id}]$ with an application to the continuation.

$$\begin{array}{ll} \text{[return]} & [\![\mathbf{return}\ a]\!]_c \longleftrightarrow c(a) \\ \text{[tailcall]} & [\![f[\mathbf{id}](a_1,\ldots,a_n)]\!]_c \longleftrightarrow f(c,a_1,\ldots,a_n) \end{array}$$

6.3.3 Closure conversion

The program intermediate representation includes higher-order and nested functions. The function nesting must be eliminated before code generation, and the lexical scoping of function definitions must be preserved when functions are passed as values. This phase of program translation is normally accomplished through *closure conversion*, where the free variables for nested functions are captured in an environment as passed to the function as an extra argument. The function body is modified so that references to variables that were defined outside the function are now references to the environment parameter. In addition, when a function is passed as a value, the function is paired with the environment as a *closure*.

The difficult part of closure conversion is the construction of the environment, and the modification of variables in the function bodies. We can formalize closure conversion as a sequence of steps, each of which preserves the program's semantics. In the first step, we must modify each function definition by adding a new environment parameter. To represent this, we replace each **let rec** $R = d$ **in** e term in the program with a new term **let rec** R **with** $[Fr = ()]\ = d$ **in** e, where Fr is an additional parameter, initialized to the empty tuple $()$, to be added to each function definition. Simultaneously, we replace every occurrence of the record variable R with $R(Fr)$, which represents the partial application of the record R to the tuple Fr.

$$\begin{array}{ll} \text{[frame]} & \mathbf{let\ rec}\ R = d[R]\ \mathbf{in}\ e[R] \\ \longleftrightarrow & \mathbf{let\,rec}\,R\ \mathbf{with}\ [Fr = ()]\ = d[R(Fr)]\ \mathbf{in}\ e[R(Fr)] \end{array}$$

The second part of closure conversion does the closure operation using two operations. For the first part, suppose we have some expression e with a free variable v. We can abstract this variable using a call-by-name function application as the expression **let** $v = v$ **in** e, which reduces to e by simple β-reduction.

$$[\text{abs}] \qquad e[v] \longleftrightarrow \textbf{let}\, v = v\ \textbf{in}\ e[v]$$

By selectively applying this rule, we can quantify variables that occur free in the function definitions d in a term $\textbf{let rec}\, R\ \textbf{with}\ [\mathit{Fr} = \mathit{tuple}]\ = d\ \textbf{in}\ e$. The main closure operation is the addition of the abstracted variable to the frame, using the following rewrite.

$$\begin{array}{ll}
[\text{close}] & \textbf{let}\, v = a\ \textbf{in} \\
 & \textbf{let rec}\, R\ \textbf{with}\ [\mathit{Fr} = (a_1, \ldots, a_n)]\ = d[R; v; \mathit{Fr}]\ \textbf{in}\ e[R; v; \mathit{Fr}] \\
\longleftrightarrow & \textbf{let rec}\, R\ \textbf{with}\ [\mathit{Fr} = (a_1, \ldots, a_n, a)]\ = \\
 & \quad \textbf{let}\, v = \ \mathit{Fr}.[n+1]\ \textbf{in}\ d[R; v; \mathit{Fr}] \\
 & \textbf{in let}\, v = a\ \textbf{in}\ e[R; v; \mathit{Fr}]
\end{array}$$

Once all free variables have been added to the frame, the $\textbf{let rec}\, R\ \textbf{with}$ $[\mathit{Fr} = \mathit{tuple}]\ = d\ \textbf{in}\ e$ is rewritten to use explicit tuple allocation.

$$\begin{array}{ll}
[\text{alloc}] & \textbf{let rec}\, R\ \textbf{with}\ [\mathit{Fr} = \mathit{tuple}]\ = d[R; \mathit{Fr}]\ \textbf{in}\ e[R; \mathit{Fr}] \\
\longleftrightarrow & \textbf{let rec}\ R = \textbf{frame}(\mathit{Fr}, d[R; \mathit{Fr}])\ \textbf{in let}\ \mathit{Fr} = \ (\mathit{tuple})\ \textbf{in}\ e[R; \mathit{Fr}]
\end{array}$$

The final step of closure conversion is to propagate the subscript operations into the function bodies.

$$\begin{array}{ll}
[\text{arg}] & \textbf{frame}(\mathit{Fr}, \textbf{fun}\ l = \lambda v.e[\mathit{Fr}; v]\ \textbf{and}\ d[\mathit{Fr}]) \\
\longleftrightarrow & \textbf{fun}\ l = \lambda \mathit{Fr}.\lambda v.e[\mathit{Fr}; v]\ \textbf{and frame}(\mathit{Fr}, d[\mathit{Fr}]) \\
[\text{sub}] & \textbf{let}\, v_1 = \ a_1.[a_2]\ \textbf{in fun}\ l = \lambda v_2.e[v_1; v_2]\ \textbf{and}\ d[v_1] \\
\longleftrightarrow & \textbf{fun}\ l = \lambda v_2.\, \textbf{let}\, v_1 = \ a_1.[a_2]\ \textbf{in}\ e[v_1; v_2]\ \textbf{and let}\, v_1 = \ a_1.[a_2]\ \textbf{in}\ d[v_1]
\end{array}$$

6.3.4 IR optimizations

Many optimizations on the intermediate representation are quite easy to express. For illustration, we include two very simple optimizations: dead-code elimination and constant folding.

Dead code elimination Formally, an expression e in a program p is dead if the removal of expression e does not change the behavior of the program

p. We approximate this with the following rewrite rules.

[datom]	$\mathbf{let}\, v = a \;\mathbf{in}\; e \longleftrightarrow e$
[dtuple]	$\mathbf{let}\, v = (a_1, \ldots, a_n) \;\mathbf{in}\; e \longleftrightarrow e$
[dsub]	$\mathbf{let}\, v = a_1.[a_2] \;\mathbf{in}\; e \longleftrightarrow e$
[dcl]	$\mathbf{letc}\, v = a_1(a_2) \;\mathbf{in}\; e \longleftrightarrow e$

The syntax of these rewrites depends on the second-order specification of substitution. Note that the pattern e is *not* expressed as the second-order pattern $e[v]$. That is, v is *not* allowed to occur free in e.

Constant folding Another simple class of optimizations is constant folding. If we have an expression that includes only constant values, the expression may be computed at compile time. In the following rewrites the notation $[\![op]\!]$ is the interpretation of the arithmetic operator in the meta-language.

[binop]	$i\; binop\; j \longleftrightarrow [\![op]\!](i, j)$
[relop]	$i\; relop\; j \longleftrightarrow [\![op]\!](i, j)$
[ift]	$\mathbf{if}\ \top\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \longleftrightarrow e_1$
[iff]	$\mathbf{if}\ \bot\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \longleftrightarrow e_2$

6.4 Summary and Future Work

One of the points we have stressed in this presentation is that the formalized versions of these algorithms is easy, in fact easier than the definition using traditional general-purpose languages. By adding these algorithms to the FDL, the formal description is freely-available for use in a wide variety of applications. This will increase the reliability of these algorithms, because the algorithm description is *verifiable*, and it will make development easier, by providing a common implementation.

The formal development of these algoriths was eased for several reasons. MetaPRL provided a great deal of automation for frequently occurring tasks. In most cases, the implementation of a new compiler phase meant only the development of new rewrite rules. There is very little of the "grunge" code that plagues traditional implementations, such as the maintenance of tables that keep track of the variables in scope, code-walking procedures to apply a transformation to the program's subterms, and other kinds of housekeeping code.

As a basis of comparison, we can compare the formal compiler in this chapterto a similar native-code compiler for a fragment of the Java language

Description	Formal compiler		Java
	Rewrites	Total	
CPS conversion	44	347	338
Closure conversion	54	410	1076
Code generation	214	648	1012
Total code base	484	10000	12000

Figure 6.3: Code comparison

we developed as part of the Mojave project [83]. The Java compiler is written in OCaml, and uses an intermediate representation similar to the one presented in this paper, with two main differences: the Java intermediate representation is typed, and the x86 assembly language is not scoped.

Figure 6.3 gives a comparison of some of the key parts of both compilers in terms of lines of code, where we omit code that implements the Java type system and class constructs. The formal compiler columns list the total lines of code for the term rewrites, as well as the total code including rewrite strategies. The size of the total code base in the formal compiler is still quite large due to the extensive code needed to implemented the graph coloring algorithm for the register allocator. Preliminary tests suggest that performance of programs generated from the formal compiler is comparable, sometimes better than, the Java compiler due to a better spilling strategy.

The work presented in this paper took roughly one person-week of effort from concept to implementation, while the Java implementation took roughly three times as long. It should be noted that, while the Java compiler has been stable for about a year, it still undergoes periodic debugging. Register allocation is especially problematic to debug in the Java compiler, since errors are not caught at compile time, but typically cause memory faults in the generated program.

This work is far from complete. The current example serves as a proof of concept, but it remains to be seen what issues will arise when the formal compilation methodology is applied to more complex programming languages. For future work, we intend to approach the problem of developing and validating formal compilers in three steps. The first step is the development of typed intermediate languages. These languages admit a broader class of rewrite transformations that are conditioned on well-typed programs, and the typed language serves as a launching point for compiler validation. The second step is to develop a semantics of the intermediate language and validate the rewrite rules for a small source language similar to the one presented here. It is not clear whether the same properties should be applied to the assembly language—whether the assembly language should be typed, and whether it is feasible to develop a simple formal model of the target architecture that will allow the code generation and register allocations phases to be verified. The final step is to extend the source language to one resembling a modern general-purpose language.

6.5 Related work

Term rewriting has been successfully used to describe programming language syntax and semantics, and there are systems that provide efficient term representations of programs as well as rewrite rules for expressing program transformations. For instance, the `ASF+SDF` environment [174] allows the programmer to construct the term representation of a wide variety of programming syntax and to specify equations as rewrite rules. These rewrites may be conditional or unconditional, and are applied until a normal form is reached. Using equations, programmers can specify optimizations, program transformations, and evaluation. The `ASF+SDF` system targets the generation of informal rewriting code that can be used in a compiler implementation.

Liang [109] implemented a compiler for a simple imperative language using a higher-order abstract syntax implementation in λProlog. Liang's approach includes several of the phases we describe here, including parsing, CPS conversion, and code generation using a instruction set defined using higher-abstract syntax (although in Liang's case, registers are referred to indirectly through a meta-level store, and we represent registers directly as variables). Liang does not address the issue of validation in this work, and the primary role of λProlog is to simplify the compiler implementation. In contrast to our approach, in Liang's work the entire compiler was implemented in λProlog, even the parts of the compiler where implementation in a more traditional language might have been more convenient (such as register allocation code).

`FreshML` [156] adds to the ML language support for straightforward encoding of variable bindings and alpha-equivalence classes. Our approach differs in several important ways. Substitution and testing for free occurrences of variables are explicit operations in `FreshML`, while MetaPRL provides a convenient implicit syntax for these operations. Binding names in

FreshML are inaccessible, while only the formal parts of MetaPRL are prohibited from accessing the names. Informal portions—such as code to print debugging messages to the compiler writer, or warning and error messages to the compiler user—can access the binding names, which aids development and debugging. FreshML is primarily an effort to add automation; it does not address the issue of validation directly.

Previous work has also focused on augmenting compilers with formal tools. Instead of trying to split the compiler into a formal part and a heuristic part, one can attempt to treat the *whole* compiler as a heuristic adding some external code that would watch over what the compiler is doing and try to establish the equivalence of the intermediate and final results. For example, the work of Necula and Lee [138, 139] has led to effective mechanisms for certifying the output of compilers (e.g., with respect to type and memory-access safety), and for verifying that intermediate transformations on the code preserve its semantics.

There have been efforts to present more functional accounts of assembly as well. Morrisett *et. al.* [134] developed a typed assembly language capable of supporting many high-level programming constructs and proof carrying code. In this scheme, well-typed assembly programs cannot "go wrong."

Chapter 7

Samples of Content

Here we show examples of expository hybrid texts referencing formal content. Here they have been rendered through Latex as hardcopy, and the Latex code for formal text such as formulas was generated automatically rather than by hand (though in a few cases some adjustment for improved formatting in this book was made after the original latex-code generation). When rendered as on-line texts, references can be followed to the definitions and proofs.

7.1 Excerpt From Event Systems Article

The following is an excerpt from a paper about the logic of events. All the formulas and terms in this paper were generated automatically by inserting references to objects in the FDL into the Latex source document. In the source document we do not even use the Latex math mode – all the math is generated from the FDL.

7.1.1 Event Systems

We want an abstract model that can capture the observable features of a distributed system. The fundamental types are *locations* and *events* which we can think of as space and time coordinates. Information is stored at a location as the value of a state variable or an *observable* and information is passed from one location to another along *links* in the form of *messages*.

Locations, observables (state variables), local action kinds, and message tags are all represented as members of the type `Id`, and links have type

`IdLnk`. The types `Id`, and `IdLnk` are discrete types – equality on each type is decidable. The operations `source(l)` and `destination(l)` assign source and destination locations to the links, forming a graph structure on the locations and links.

A message will consist of a link, a tag, and a value whose type may depend on the link and the tag.

```
         Msg(M)  ≡  l:IdLnk × t:Id × (M l t)
      msg(l;t;v)  ≡  <l, t, v>
         mlnk(m)  ≡  m.1
         mtag(m)  ≡  m.2.1
         mval(m)  ≡  m.2.2
   haslink(l; m)  ≡  mlnk(m) = l
   Msg_sub(l; M)  ≡  {m:Msg(M)| haslink(l; m)}
    onlnk(l;mss)  ≡  filter(λms.mlnk(ms) = l;mss)
```

Every event will have a kind, a value, and a location. So an event is a point in spacetime. There are two kinds of events, local events have a kind that is an `Id` and the receipt of a message on a given link with a given tag is the other kind of event.

```
                           Knd  ≡  IdLnk × Id + Id
                      isrcv(k)  ≡  isl(k)
                    islocal(k)  ≡  ¬_b isl(k)
                    rcv(l; tg)  ≡  inl <l, tg>
                       locl(a)  ≡  inr a
                        lnk(k)  ≡  outl(k).1
                        tag(k)  ≡  outl(k).2
                        act(k)  ≡  outr(k)
  kindcase(k;a.f[a];l,t.g[l; t])  ≡  if islocal(k)
                                     then f[act(k)]
                                     else g[lnk(k); tag(k)]
                                     fi
```

An *event system* is a structure consisting of a discrete type `E` of events and a set of operations and axioms. Operations `loc(e)`, `kind(e)`, and `val(e)` extract the location, kind, and value from an event. Operations `(x when e)` and `(x after e)` observe the values of the observables at the points in spacetime. Operation `first(e)` is a boolean that is true of only the first event at each location. Messages must originate at some point in spacetime, and the operations `sends(l;e)`, `sender(e)`, and `index(e)` define this structure. The `sends(l;e)` of an event `e` on link `l` will be a list of messages on that link that originate at `e`. We build the semantics of message delivery into our model in a way that makes every link into a reliable fifo channel. Thus every message is eventually received, and for a receive event `e` the operations `sender(e)` and `index(e)` will provide the originator of the message received and the index of that message in the list that originated there. The temporal order structure on our spacetime is provided by two orderings on events `(e <loc e')` and `(e < e')`. The local ordering `(e <loc e')` is a total, discrete, well-founded, linear ordering on events with the same location. So, at each location, if there are any events, there must be a `<loc`-minimal event satisfying the predicate `first(e)`, and every non-minimal event `e` must have an immediate local predecessor `pred(e)`.

The causal ordering `(e < e')` is also well-founded and is the transitive closure of `(e <loc e')` and the relation that a receive event `e` is preceded by `sender(e)`.

Formally, an event systems is a member of the following dependent product type. This type is typical of the way structures are represented in type theory.

```
ES  ≡  E:𝕌
       × eq:EqDecider(E)
       × T:Id ⟶ Id ⟶ 𝕌
       × V:Id ⟶ Id ⟶ 𝕌
       × M:IdLnk ⟶ Id ⟶ 𝕌
       × unused:Top
       × loc:E ⟶ Id
       × kind:E ⟶ Knd
       × val:e:E ⟶ eventtype(kind;loc;V;M;e)
       × when:x:Id ⟶ e:E ⟶ (T (loc e) x)
       × after:x:Id ⟶ e:E ⟶ (T (loc e) x)
       × sends:l:IdLnk ⟶ E ⟶ (Msg_sub(l; M) List)
       × sender:{e:E| ↑isrcv(kind e)}  ⟶ E
       × index:e:{e:E| ↑isrcv(kind e)}  ⟶ ℕ||sends
                                          lnk(kind e)
                                          (sender e)||
       × first:E ⟶ 𝔹
       × pred:e':{e':E| ¬↑(first e')}  ⟶ E
       × causl:E ⟶ E ⟶ ℙ
       × p:ESAxioms(E;T;M;
                    loc;kind;val;
                    when;after;
                    sends;sender;index;
                    first;pred;
                    causl)
       × Top
```

The type Top at the end allows us to define subtypes of ES that have additional operators and axioms.

The axioms of the event system structure are the following

```
Trans(E;e,e'.(e <loc e'))                                (7.1)
SWellFounded((e <loc e'))                                (7.2)
∀e,e':E.                                                 (7.3)
  (loc(e) = loc(e')
  ⟺ (e <loc e') ∨ (e = e') ∨ (e' <loc e))
```

```
∀e:E. (↑first(e) ⟺ ∀e':E. (¬(e' <loc e)))             (7.4)
∀e:E                                                    (7.5)
  ((¬↑first(e))
  ⇒ ((pred(e) <loc e)
     ∧ (∀e':E
          (¬((pred(e) <loc e') ∧ (e' <loc e))))))
∀e:E                                                    (7.6)
  ((¬↑first(e))
  ⇒ (∀x:Id. ((x when e) = (x after pred(e)))))
Trans(E;e,e'.(e < e'))                                  (7.7)
SWellFounded((e < e'))                                  (7.8)
∀e:E                                                    (7.9)
  ((↑isrcv(e))
  ⇒ (sends(lnk(e);sender(e))[index(e)]
     = msg(lnk(e);tag(e);val(e))))
∀e,e':E.  ((e <loc e') ⇒ (e < e'))                     (7.10)
∀e:E. ((↑isrcv(e)) ⇒ (sender(e) < e))                  (7.11)
∀e,e':E.                                               (7.12)
  ((e < e')
  ⇒ (((¬↑first(e'))
     c∧ ((e < pred(e')) ∨ (e = pred(e'))))
     ∨ ((↑isrcv(e'))
       c∧ ((e < sender(e')) ∨ (e = sender(e'))))))
∀e:E                                                   (7.13)
  ((↑isrcv(e)) ⇒ (loc(e) = destination(lnk(e))))
∀e:E. ∀l:IdLnk.                                        (7.14)
  ((¬(loc(e) = source(l))) ⇒ (sends(l;e) = []))
∀e,e':E.                                               (7.15)
  ((↑isrcv(e))
  ⇒ (↑isrcv(e'))
  ⇒ (lnk(e) = lnk(e'))
  ⇒ ((e <loc e')
     ⟺ (sender(e) <loc sender(e'))
         ∨ ((sender(e) = sender(e'))
           ∧ (index(e) < index(e')))))
```

```
∀e:E. ∀l:IdLnk. ∀n:ℕ||sends(l;e)||.                      (7.16)
  ∃e':E
   ((↑isrcv(e'))
   c∧ ((lnk(e') = l)
      ∧ (sender(e') = e)
      ∧ (index(e') = n)))
```

Consequences of the axioms

We state as lemmas some properties that follow from the axioms.

```
∀e:E. (¬(e <loc e))                                          (7.17)
∀e:E. (¬(e < e))                                             (7.18)
∀e,e':E.                                                     (7.19)
  ((e <loc e')
  ⟺ (¬↑first(e')) ∧ ((e = pred(e')) ∨ (e <loc pred(e'))))
∀e,e':E.                                                     (7.20)
  (((e <loc e') ∧ (∀e1:E. (¬((e <loc e1) ∧ (e1 <loc e')))))
  ⇒ (e = pred(e')))
∀e',e:E.  Dec((e <loc e'))                                   (7.21)
∀e',e:E.  Dec((e < e'))                                      (7.22)
∀e:E. ∀l:IdLnk. ∀m:Msg.                                      (7.23)
  ((m ∈ sends(l;e))
  ⇒ ∃e':rvc(l,mtag(m),v).(e < e')
     ∧ ((msgtype(m) = valtype(e')) c∧ (v = mval(m))))
```

proofs: Lemmas 7.17 and 7.18 follow from the general fact that

$$WellFounded(Rel) \;\Rightarrow\; AntiReflexive(Rel)$$

Suppose `(e <loc e')`. From axiom (7.4) we conclude `¬↑first(e')`, and from axiom (7.5) we conclude

```
(pred(e') <loc e')
∧ (∀e'':E
     (¬((pred(e') <loc e'') ∧ (e'' <loc e'))))
```

So `¬(pred(e') <loc e)` and hence, from axiom (7.3), `e` $\leq$ `pred(e')` , which proves lemma 7.19. If we also have

```
∀e1:E. (¬((e <loc e1) ∧ (e1 <loc e')))
```

then

$$\neg\texttt{(e <loc pred(e'))}$$

so `e = pred(e')`, which proves lemma 7.20.

We may now prove lemma 7.21 by induction, using axiom (7.2). By lemma 7.19 it's enough to decide `(¬↑first(e')) ∧ e ≤ pred(e')` , but this is decidable by the induction hypothesis, and the decidability of equality in `E`. The proof of lemma 7.22 is similar. Using the other axioms we can show that axiom (7.12) can be proved as an if and only if statement, and hence it is enough to show that its righthand side is decidable. This follows from the induction hypothesis and the decidability of equality in `E`.

If `(msg(l;tg;v) ∈ sends(l;e))` then for some `n < ||sends(l;e)||`,

```
msg(l;tg;v) = sends(l;e)[n]
```

By axiom (7.16) there is an e' such that

```
((↑isrcv(e')) ∧ (lnk(e') = l)) ∧ (sender(e') = e) ∧ (index(e') = n)
```

So, by axiom (7.9),

```
(val(e') = mval(msg(l;tg;v)))
∧ (tg = mtag(msg(l;tg;v)))
```

That implies that `kind(e') = rcv(l; tg)` and since `e = sender(e')` we have `(e < e')` by axiom (7.11). This proves lemma 7.23.

Local histories

An event system is a rich enough structure that we can define various "history" operators that list or count previous events having certain properties. Because we can define operators like these we do not need to add "history variables" to the states in order to write specifications and and prove them.

The basic history operator lists all the prior events at a location.

Definition

```
before(e)  ≡  if first(e)
              then []
              else before(pred(e)) @ [pred(e)]
              fi
  [e, e']  ≡  filter(λev.es-ble{i:l}(es;e;ev);before(e') @ [e'])
  rcvs(l;before(e'))  ≡  filter(λe.haslnk(l;e);before(e'))
   snds(l;before(e))  ≡  concat(map(λe.sends(l;e);before(e)))
snds(l, before(e,n))  ≡  snds(l;before(e)) @ firstn(n;sends(l;e))
```

Using these operators we can state (and prove) the following important lemma.

Lemma Fifo

```
∀e:E
((↑isrcv(e))
⇒ (snds(lnk(e), before(sender(e),index(e)))
= msgs(lnk(e);before(e))))
```

proof: The proof is by induction on $<_{loc}$. The full proof is in then FDL.

□

Event system shorthands

We make some shorthand notations:

```
           ∀e@i.P[e]  ≡  ∀e:E. ((loc(e) = i) ⇒ P[e])
           ∃e@i.P[e]  ≡  ∃e:E. ((loc(e) = i) ∧ P[e])
      @i always.P[x]  ≡  ∀e@i.P[(x when e)]
 @i always.P[x1; x2]  ≡  ∀e@i.P[(x1 when e); (x2 when e)]
     ∃e=k(v).P[e; v]  ≡  ∃e:E. ((kind(e) = k) ∧ P[e; val(e)])
∃e:rvc(l,tg,v).P[e; v]  ≡  ∃e:E
                            ((↑isrcv(e))
                            ∧ (lnk(e) = l)
                            ∧ (tag(e) = tg)
                            ∧ P[e; val(e)])
```

7.1.2 Worlds

Definition of World

A *world* is a generalized trace of the execution of a distributed system. Time is modeled as the natural numbers ℕ. By observing the system at every location `i` and every time `t`, we have a state `s(i;t)`, an action `a(i;t)`, and a list of messages `m(i;t)`. The state `s(i;t)` is the state of the part of the system at location `i` at time `t`. We assume that the type of the state at location `i` does not change with time, and we use a general model of state as a record. A record is a dependent function. A world contains a type X of state variable names and and a type assignment $T : \mathit{Loc} \to X \to \mathbb{U}$. The state at location i of the world will have type $\mathit{Record}(X, T(i))$.

The action `a(i;t)` is the action that was chosen by the system to be executed next at location `i` and time `t`. It will always be possible that no action was taken at `i`,`t` so we must have a null action. Other action will be local actions with names taken from a type of action names A, and also the action of receiving a message. Every action will have a kind of one of these three forms (null, local, or receive), and also a value whose type depends on the kind and location of the action.

```
action(dec)  ≡  Unit + (k:Knd × (dec k))
  isnull(a)  ≡  isl(a)
    kind(a)  ≡  outr(a).1
     val(a)  ≡  outr(a).2
 isrcv(l;a)  ≡  (¬_b isnull(a))
                ∧_b isrcv(kind(a))
                ∧_b lnk(kind(a)) = l
```

The messages `m(i;t)` are the list of messages sent from location `i` at time `t`. For messages, we use the message type `Msg(M)` defined earlier.

```
World  ≡  T:Id ⟶ Id ⟶ 𝕌
          × TA:Id ⟶ Id ⟶ 𝕌
          × M:IdLnk ⟶ Id ⟶ 𝕌
          × s:i:Id ⟶ ℕ ⟶ x:Id ⟶ (T i x)
          × a:i:Id ⟶ ℕ ⟶ action(w-action-dec(TA;M;i))
          × m:i:Id
              ⟶ ℕ
              ⟶ ({m:Msg(M)| source(mlnk(m)) = i}  List)
          × Top
```

If $w : \mathit{World}$ is a world, then we write w_{Loc}, w_{Lnk}, $\ldots$, w_s, w_a, and w_m for the components of w.

Fair-Fifo Worlds

We next define a *fair-fifo* world. We first note that, given world w, we can find all the messages sent on link l and all and receive actions that have occurred on link l before time t:

```
   m(i;t)  ≡  (w.2.2.2.2.2.1) i t
   m(l;t)  ≡  onlnk(l;m(source(l);t))
snds(l;t)  ≡  concat(map(λt1.m(l;t1);upto(t)))
rcvs(l;t)  ≡  filter(λa.isrcv(l;a);map(λt1.a(destination(l);t1);
              upto(t)))
```

The send and receive messages before time t define an implicit queue, and we can test whether the queue for link l is empty and for whether message ms is at the head of the queue for its link:

```
queue(l;t)  ≡  nth_tl(||rcvs(l;t)||;snds(l;t))
```

The predicate `FairFifo` is the conjuntion of the following four clauses

```
∀i:Id. ∀t:ℕ. ∀l:IdLnk.                                          (7.24)
  ((¬(source(l) = i)) ⇒ (onlnk(l;m(i;t)) = []))
∀i:Id. ∀t:ℕ.                                                    (7.25)
  ((↑isnull(a(i;t)))
  ⇒ ((∀x:Id. (s(i;t + 1).x = s(i;t).x)) ∧ (m(i;t) = [])))
∀i:Id. ∀t:ℕ. ∀l:IdLnk.                                          (7.26)
  ((↑isrcv(l;a(i;t)))
  ⇒ ((destination(l) = i)
     ∧ ((||queue(l;t)|| ≥ 1 )
       c∧ (hd(queue(l;t)) = msg(a(i;t))))))
∀l:IdLnk. ∀t:ℕ.                                                 (7.27)
  ∃t':ℕ
   ((t ≤ t')
   ∧ ((↑isrcv(l;a(destination(l);t')))
     ∨ (queue(l;t') = [])))
```

The first clause says that location i can only send message on links whose source is i. The second clause says that a null action leaves the state unchanged and sends no messages. The third clause says that a receive action at location i must be on a link whose destination is i and whose message is at the head of the queue. The fouth clause is the fairness clause. It says that for every queue, infinitely often either the queue is empty or a receive event occurs at its destination.

Event System of a World

If w is a fair-fifo world, then we can construct an event system from w. We have to define the type of events and define all the operations on events and show that the axioms are satisfied. Our events will be the points $\langle i, t\rangle$ in spacetime at which an action occured in w.

$$
\begin{array}{rcl}
\texttt{E} & \equiv & \{\texttt{p:Id} \times \mathbb{N}\texttt{|}\ \neg\uparrow\texttt{isnull(a(p.1;p.2))}\} \\
\texttt{loc(e)} & \equiv & \texttt{e.1} \\
\texttt{time(e)} & \equiv & \texttt{e.2} \\
\texttt{Action(i)} & \equiv & \texttt{action(w-action-dec(w.TA;w.M;i))} \\
w_{state}(\langle i,t\rangle) & = & w_s(i,t) \\
w_{state'}(\langle i,t\rangle) & = & w_s(i,t+1) \\
w_{init}(i) & = & w_s(i,0) \\
w_{msgs}(\langle i,t\rangle) & = & w_m(i,t)
\end{array}
$$

For and event $e \in w_E$ we have $\neg isnull(w_{action}(e))$ so we may define

$$
\begin{array}{rcl}
\texttt{kind(a)} & \equiv & \texttt{outr(a).1} \\
\texttt{val(a)} & \equiv & \texttt{outr(a).2}
\end{array}
$$

The type of the value of an event can be determined from its location and kind using the type assignments w_{TA} and w_M as follows:

$$
\texttt{V(i;k)} \quad \equiv \quad \texttt{kindcase(k;a.(w.2.1) i a;l,tg.(w.2.2.1) l tg)}
$$

The observation operators are defined in the obvious way:

$$
\begin{array}{rcl}
\texttt{(x when e)} & \equiv & \texttt{s(e.1;e.2).x} \\
\texttt{(x after e)} & \equiv & \texttt{s(e.1;(e.2) + 1).x} \\
\texttt{sends(l;e)} & \equiv & \texttt{onlnk(l;m(loc(e);time(e)))}
\end{array}
$$

The local ordering operations are also straightforward.

```
  first(e)   ≡  if (time(e) $=_z$ 0) then tt
                if isnull(a(loc(e);time(e) - 1))
                  then first(<loc(e), time(e) - 1>)
                else ff
                fi
   pred(e)   ≡  if isnull(a(loc(e);time(e) - 1))
                then pred(<loc(e), time(e) - 1>)
                else <loc(e), time(e) - 1>
                fi
 e <loc e'   ≡  (loc(e) = loc(e')) ∧ (time(e) < time(e'))
```

To define the *sender* and *index* operations that match a receive event to its origin, we first define a *match* with the same *snds* and *rcvs* functions used in defining $FairFifo$.

```
  match(l;t;t')  ≡  ||snds(l;t)|| ≤z ||rcvs(l;t')||
                    $\wedge_b$ ||rcvs(l;t')|| <z ||snds(l;t)||
                       + ||onlnk(l;m(source(l);t))||
```

Then, we define *sender* and *index* as follows

```
  sender(e)  ≡  <source(lnk(kind(e)))
                , mu(λt.match(lnk(kind(e));t;time(e)))
                >
   index(e)  ≡  ||rcvs(lnk(kind(e));time(e))||
                - ||snds(lnk(kind(e));time(sender(e)))||
```

Finally, the causal ordering $\prec$ is defined as a transitive closure

```
e <c e'  ≡  e
            λe,e'.
             (e <loc e'
             ∨ ((↑isrcv(kind(e'))) c∧ (e = sender(e'))))^+ e'
```

Putting all of these defined operations together, we have the event structure defined by the world and we can prove that the constructed event system satisfies all the event system axioms.

Theorem (World-Event-System)

$\forall$the$_w$:World
(FairFifo
$\Rightarrow$ ESAxioms(E;$\boldsymbol{\lambda}$i,x.vartype(i;x);the$_w$.M;
$\boldsymbol{\lambda}$e.loc(e);$\boldsymbol{\lambda}$e.kind(e);$\boldsymbol{\lambda}$e.val(e);
$\boldsymbol{\lambda}$x,e.(x when e);$\boldsymbol{\lambda}$x,e.(x after e);
$\boldsymbol{\lambda}$l,e.sends(l;e);$\boldsymbol{\lambda}$e.sender(e);$\boldsymbol{\lambda}$e.index(e);
$\boldsymbol{\lambda}$e.first(e);$\boldsymbol{\lambda}$e.pred(e);
$\boldsymbol{\lambda}$e,e'.e <c e'))

proof: The full proof is in then FDL.

□

7.1.3 Message-Automata

Event systems and worlds are infinite objects, but they arise from the behaviors of distributed systems where, at each location, only a finite program constrains the behavior. We call our representations of these finite programs message-automata. To make our representations finite we need to replace infinite things like total type assignments with finite approximations, so we need some notation for finite partial functions.

Finite partial functions

The type `a:A fp-> B[a]` is the type of finite partial functions `f` from `A` to `B[a]` . Its domain is `dom(f)`, and we define

f(x)?z $\equiv$ if x $\in$ dom(f) then f(x) else z fi
z != f(x) ==> P[a; z] $\equiv$ ($\uparrow$x $\in$ dom(f)) $\Rightarrow$ P[x; f(x)]

For finite partial functions `f` and `g` we define:

f $\subseteq$ g $\equiv$ $\forall$x:A
 (($\uparrow$x $\in$ dom(f))
 $\Rightarrow$ (($\uparrow$x $\in$ dom(g)) c$\wedge$ (f(x) = g(x))))
f || g $\equiv$ $\forall$x:A
 ((($\uparrow$x $\in$ dom(f)) $\wedge$ ($\uparrow$x $\in$ dom(g)))
 $\Rightarrow$ (f(x) = g(x)))

```
f ⊕ g  ≡  <(f.1) @ filter(λa.(¬_b a ∈ dom(f));g.1)
          , λa.f(a)?g(a)
          >
```

Note that there is an empty partial function (with an empty domain) that is compatible with every finite partial function and is an identity operator with respect to `f ⊕ g`.

lemma

```
  ∀f,g:a:A fp-> B[a].  (f || g ⇒ {f ⊆ f ⊕ g ∧ g ⊆ f ⊕ g})
```

lemma

```
  ∀f,g:a:A fp-> B[a]. ∀x:A. ∀P:a:A ⟶ B[a] ⟶ ℙ.
    (g ⊆ f ⇒ z != f(x) ==> P[x;z] ⇒ z != g(x) ==> P[x;z])
```

Definition of Message-Automata

The message-automata share with the worlds and the event systems the same spaces of names for state variables, local action kinds, and message tags. But, where a world has, at each location i, type assignments $T(i) : \rightarrow X\mathbb{U}$, $TA(i) : \rightarrow A\mathbb{U}$, and $M : Lnk \rightarrow Tag \rightarrow \mathbb{U}$, a message-automaton will have finite type assignments (declarations)

$$ds : \texttt{x:Id fp->}\ \mathbb{U}$$
$$da : \texttt{a:Knd fp->}\ \mathbb{U}$$

The domain of ds is the set of declared state variables, the domain of da is the set of declared action kinds, both local actions and send/receive actions.

The state of a message-automaton will be the record defined by its declarations ds. We can define this as follows:

```
State(ds)  ≡  x:Id ⟶ ds(x)?Top
```

Here we extend the finite partial function ds to a total function by assigning the type Top to any undeclared state variable.

Every action has a value whose type depends only on the action kind. The type of the value associated with an action of kind k is defined by

```
Valtype(da;k)  ≡  da(k)?Top
```

In addition, to its declarations, the message-automaton does the following things *init* It constrains the initial values of the declared state variables. So, it has a finite partial function *init* of type `x:Id fp-> ds(x)?Void`. Thus, if x is in the domain of *init* then x is a declared state variable and $init(x)$ is a value of the declared type $ds(x)$ of state variable x.

pre It declares preconditions on its local actions. So, it has a finite partial function *pre* of type

```
a:Id fp-> State(ds) ⟶ Valtype(da;locl(a)) ⟶ ℙ
```

Thus, if a is in the domain of *pre* then a is a declared local action and $pre(a)$ is a predicate on the state and the declared type $da(a)$ of the action.

ef It declares the effects of actions (local and input) on state variables. So, it has a finite partial function *ef* of type

```
kx:Knd × Id fp-> State(ds)
⟶ Valtype(da;kx.1)
⟶ ds(kx.2)?Void
```

Thus, if $\langle k, x\rangle$ is in the domain of *ef* then k is an action kind and x is a declared state variable, and $ef(\langle k, x\rangle)$ is a function from the state and the action value to the type $ds(x)$ of x. This function defines how the new value of x will be computed from the current state and the value of the action.

send It declares the messages sent by actions. So, it has a finite partial function *send* of type

```
kl:Knd × IdLnk fp-> (tg:Id
× (State(ds)
  ⟶ Valtype(da;kl.1)
  ⟶ (da(rcv(kl.2; tg))?Void List))) List
```

Thus, if $\langle k, l\rangle$ is in the domain of *send* then k is an action kind and l is link. $snd(\langle k, l\rangle)$ is a list of pairs of type

```
tg:Id × (State(ds)
        ⟶ Valtype(da;kl.1)
        ⟶ (da(rcv(kl.2; tg))?Void List))
```

For each pair $\langle tg, f\rangle$ in this list, the function f when applied to the current state and the value of the action gives a list of values of the type declared for link l and tag tg. The concatenation of all of these lists is the list of messages the action will send. This form for the send clause allows us to have conditional sends since the list returned by f may be empty. Also, a single action may send multiple messages on a link (and it may send on multiple links if there are other send clauses).

$frame$ It declares implicit effects. By convention, the effects that are explicitly given are the only actions that affect the given state variables. So the implicit effect of any other action is to leave the state of variable unchanged. Since we want each clause of a message-automaton to be meaningful on its own, we can't depend on such contextual conventions, so we have to make the implicit effects explicit in so-called *frame* clauses. The message-automaton has a finite partial function $frame$ of type `x:Id fp-> Knd List`. So if x is in the domain of $frame$ then x is a declared state variable and $frame(x)$ is a list of actions kinds that contains all the kinds that affect x.

$sframe$ It declares implicit sends. By convention, the sends that are explicitly given are the only actions that send messages on the given link with the given tag. So the implicit sends of any other action is to send no messages of the given link and tag. We make the implicit sends explicit in *sframe* clauses. The message-automaton has a finite partial function $sframe$ of type `ltg:IdLnk × Id fp-> Knd List`. So if $\langle l, tg\rangle$ is in the domain of $sframe$ then l is an output link and $sframe(\langle l, tg\rangle)$ is a list of actions kinds that contains all the kinds that send messages with tag tg on link l.

Putting all of these pieces into a structure we define the type of message-automata:

```
MsgA  ≡  ds:x:Id fp-> 𝕌
         × da:a:Knd fp-> 𝕌
         × init:x:Id fp-> ds(x)?Void
         × pre:a:Id fp-> State(ds)
               ⟶ Valtype(da;locl(a))
               ⟶ ℙ
         × ef:kx:Knd × Id fp-> State(ds)
              ⟶ Valtype(da;kx.1)
              ⟶ ds(kx.2)?Void
         × send:kl:Knd × IdLnk fp-> (tg:Id
                × (State(ds)
                  ⟶ Valtype(da;kl.1)
                  ⟶ (da(rcv(kl.2; tg))?Void List))) List
         × frame:x:Id fp-> Knd List
         × sframe:ltg:IdLnk × Id fp-> Knd List
         × Top
```

Message-Automata `M1` and `M2` are compatible `M1 || M2` or satisfy the relation `M1` $\subseteq$ `M2` the eight finite partial functions, *ds*, *da*, *init*, *pre*, *ef*, *snd*, *frame*, and *sframe* of `M1` and `M2` are compatible or are related by $\subseteq$. And we define `M1` $\oplus$ `M2` by applying the $\oplus$ operation to each of the components.

lemma

```
∀M1,M2:MsgA. M1 ⊆ M1 ⊕ M2
∀M1,M2:MsgA. (M1 || M2 ⇒ M2 ⊆ M1 ⊕ M2)
```

Note that there is an empty Message-Automaton in which every component is the empty partial function. The empty automaton is compatible with every automaton and is the identity wrt the join operation.

Distributed Systems

A distributed system is an assignment of a message automaton to every location. The message automaton at a location may be the empty automaton and the distributed system has finite support if the automaton at all but a finite number of locations is the empty automaton.

```
Dsys  ≡  i:Id ⟶ MsgA
```

We say that `D2` *extends* `D1` if

$$\texttt{D1} \subseteq \texttt{D2} \equiv \forall\texttt{i:Id.}\ \ \texttt{M(i)} \subseteq \texttt{M(i)}$$

Semantics of Distributed Systems and Message-Automata

The semantics of a distributed system D is the set of possible worlds w that are consistent with it. To be consistent, w must have the same signature as D, be a fair-fifo world, and respect the meanings of the six components *init*, *pre*, ef, *send*, $frame$, and $sframe$ of the message-automata at each location. The predicate `PossibleWorld(D;w)` is defined to be the conjunction of the following clauses

```
FairFifo                                            (7.28)
∀i,x:Id.  (vartype(i;x) ⊆r M(i).ds(x))              (7.29)
∀i:Id. ∀a:Action(i).                                (7.30)
  ((¬↑isnull(a))
  ⇒ (valtype(i;a) ⊆r M(i).da(kind(a))))
∀l:IdLnk. ∀tg:Id.                                   (7.31)
  ((w.M l tg) ⊆r M(source(l)).da(rcv(l; tg)))
∀i,x:Id.  M(i).init(x,s(i;0).x)                     (7.32)
```

```
∀i:Id. ∀t:ℕ.                                                    (7.33)
  ((¬↑isnull(a(i;t)))
  ⇒ (((↑islocal(kind(a(i;t))))
     ⇒ M(i).pre(act(kind(a(i;t))),λx.s(i;t).x,
        val(a(i;t))))
    ∧ (∀x:Id
         M(i).ef(kind(a(i;t)),x,λx.s(i;t).x,
         val(a(i;t)),s(i;t + 1).x))
    ∧ (∀l:IdLnk
         M(i).send(kind(a(i;t));l;λx.s(i;t).x;
         val(a(i;t));withlnk(l;m(i;t));i))
    ∧ (∀x:Id
         ((¬M(i).frame(kind(a(i;t)) affects x))
         ⇒ (s(i;t).x = s(i;t + 1).x)))
    ∧ (∀l:IdLnk. ∀tg:Id.
         ((¬M(i).sframe(kind(a(i;t)) sends
          <l,tg>))
         ⇒ (w-tagged(tg; onlnk(l;m(i;t)))
            = [])))))
```

```
∀i,a:Id. ∀t:ℕ.                                                  (7.34)
  ∃t':ℕ
   ((t ≤ t')
   ∧ (((¬↑isnull(a(i;t')))
      c∧ (kind(a(i;t')) = locl(a)))
    ∨ (¬a declared in M(i))
    ∨ unsolvable M(i).pre(a,λx.s(i;t').x)))
```

The following result is crucial to our theory:
Theorem

```
∀A,B:Dsys.
  (A ⊆ B
  ⇒ (∀w:World
       (PossibleWorld(B;w)
       ⇒ PossibleWorld(A;w))))
```

proof: For every $i \in Loc$, $M_1 = D_1(i) \ \subseteq \ M_2 = D_2(i)$. The definition of $PossibleWorld$ uses the automata $M \in \{M_1, M_2\}$ only in the context of

conditional application of the finite partial functions, $M.init$, $M.pre$, $M.ef$, $M.send$, $M.frame$, and $M.sframe$, and also in some equality propositions over types $State(X, M.ds)$, $M.ds(x)$, and $List(Message(Lnk, Tag, M.dout))$. The conditional applications all occur positively, and so the statement for M_2 implies the statement for M_1, by the definition of $M_1 \subseteq M_2$ and the lemma on conditional application of finite partial functions. The equalities also occur positively, and, so the equality for M_2 implies the equality for M_1 because $State(X, M_2.ds)$ is a subtype of $State(X, M_1.ds)$, and similarly, $M_2.ds(x)$ is a subtype of $M_1.ds(x)$ and $List(Message(Lnk, Tag, M_2.dout))$ is a subtype of
$List(Message(Lnk, Tag, M_1.dout))$.

7.2 Some Lessons about Counting

These are explanations of elementary facts about counting developed for another project. They were selected for presentation here because the elementary content is presumably already well understood, allowing the reader to focus on the connection exhibited between formal and intuitive mathematical texts. Still, elementary theorems of the kind represented here, such as the pigeonhole principle and facts about injections and bijections, are used in more advanced developments as well.

7.2.1 Introduction to Counting

Counting is finding a function of a certain kind. When we count a class of objects, we generate an enumeration of them, which we may represent by a One-to-One Correspondence (section 7.2.2) from a standard class having that many objects to the class being counted. Our standard class of n objects, for $n \in \mathbb{N}$, will be $\mathbb{N}_n$, which is the class $\{k{:}\mathbb{Z} \mid 0 \leq k < n\}$ of natural numbers less than n. A more familiar choice of standard finite classes might have been $\{k{:}\mathbb{Z} \mid 1 \leq k \leq n\}$, but there is also another tradition in math for using $\{k{:}\mathbb{Z} \mid 0 \leq k < n\}$.

So, a class A has n members just when

$\exists f{:}(\mathbb{N}_n \to A).\ \mathrm{Bij}(\mathbb{N}_n;\ A;\ f)$

which may also be expressed as

$(\mathbb{N}_n \sim A)$

since $(\exists f{:}(A \rightarrow B).\ \mathrm{Bij}(A;\ B;\ f)) \Leftrightarrow (A \sim B)$, or as

$(A \sim \mathbb{N}_n)$ since $(A \sim B) \Rightarrow (B \sim A)$.

Now, since counting means coming up with an enumeration, we may ask whether counting in different ways, i.e., coming up with different orders, will always result in the same number, as we assume. Of course, we know this is so, but there are different degrees of knowing. It is not necessary to simply accept this as an axiom; there is enough structure to the problem to make a non-trivial proof.

$(A \sim \mathbb{N}_m) \Rightarrow (A \sim \mathbb{N}_k) \Rightarrow m = k$

Gloss (section 7.2.1.2)

This theorem is closely related to what is sometimes called the "pigeon hole principle," which states the mathematical content of the fact that if you put some number objects into fewer pigeon holes, then there must be at least two objects going into the same pigeon hole. Number the pigeon holes with the members of $\mathbb{N}_k$, and the objects with the members of $\mathbb{N}_m$; then a way of putting the objects into the holes is a function in $\mathbb{N}_m \rightarrow \mathbb{N}_k$:

$\forall m,k{:}\mathbb{N},\ f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ k<m \Rightarrow (\exists x,y{:}\mathbb{N}_m.\ x \neq y\ \&\ f(x) = f(y))$

Gloss (section 7.2.1.3)

If you examine the proofs of these theorems, you will notice that they both cite the key lemma

$(\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ \mathrm{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)) \Rightarrow m \leq k$.

Gloss (section 7.2.1.1)

Counting Indirectly (section 7.2.3) is a strategy more common than explicitly describing the a counting function for a problem.

7.2.1.1 Proof of a Fundamental theorem for Finite Injections

Show $(\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ \mathrm{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)) \Rightarrow m \leq k$.

This will be proved using induction on m, varying k. The base case, $0 \leq k$, is trivial, so we move on to the induction step, assuming $0<m$, and assuming the induction hypothesis:

$\forall k'{:}\mathbb{N}.\ (\exists f'{:}(\mathbb{N}_{m-1} \rightarrow \mathbb{N}_{k'}).\ \mathrm{Inj}(\mathbb{N}_{m-1};\ \mathbb{N}_{k'};\ f')) \Rightarrow m{-}1{\le}k'.$

The problem is then to show that $m{\le}k$, given some $f \in \mathbb{N}_m \rightarrow \mathbb{N}_k$ such that $\mathrm{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)$.

Obviously, $m{\le}k$ will follow from $m{-}1{\le}k{-}1$, so by applying the induction hyp to $k{-}1$, our problem reduces to finding an $f' \in \mathbb{N}_{m-1} \rightarrow \mathbb{N}_{k-1}$ such that $\mathrm{Inj}(\mathbb{N}_{m-1};\ \mathbb{N}_{k-1};\ f')$. Such a construction is

Replace k by $f(m)$ in f $\equiv_{\mathrm{def}}$ Replace x s.t. $x{=}_2k$ by $f(m)$ in f

(Replace x s.t. $P(x)$ by y in f)(i) $\equiv_{\mathrm{def}}$ if $P(f(i)) \rightarrow y$ else $f(i)$ fi

$\mathrm{Inj}(\mathbb{N}_{m+1};\ \mathbb{N}_{k+1};\ f) \Rightarrow \mathrm{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;$ Replace k by $f(m)$ in $f)$

This last theorem is sufficient for concluding our argument.
QED

Note: Considering $f \in \mathbb{N}_{k+1} \rightarrow \mathbb{N}_{j+1}$ as a sequence of $k{+}1$ values selected from the first $j{+}1$ natural numbers, (Replace j by $f(k)$ in f) $\in \mathbb{N}_k \rightarrow \mathbb{N}_j$ removes the entry for the largest value, namely j, and replaces it with the last value of the sequence, namely $f(k)$, if necessary.

This is the key lemma to the proofs of the uniqueness of counting, and the pigeon hole principle, i.e.,

$(A \sim \mathbb{N}_m) \Rightarrow (A \sim \mathbb{N}_k) \Rightarrow m = k$

Gloss (section 7.2.1.2)

$\forall m,k{:}\mathbb{N},\ f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ k{<}m \Rightarrow (\exists x,y{:}\mathbb{N}_m.\ x \ne y\ \&\ f(x) = f(y))$

Gloss (section 7.2.1.3)

7.2.1.2 Proof that Counting is Unique

Show $(A \sim \mathbb{N}_m) \Rightarrow (A \sim \mathbb{N}_k) \Rightarrow m = k$.

See Introduction to Counting (section 7.2.1).

We shall build this argument from a lemma about injections on standard finite types,

$(\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ \mathrm{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)) \Rightarrow m{\le}k,$

and exploiting the connection between $X \sim Y$ and injections, and the connection between $x{\le}y$ and equality.

Assume a class has size m, but also has size k; show m must actually be k. Each direction of the equality, namely,

$m \leq k$ and $k \leq m$,

may be proved in the same way, so we generalize the goal slightly to

$\forall x,y{:}\mathbb{N}.\ (\mathbb{N}_x \sim \mathbb{N}_y) \Rightarrow x \leq y.$

Once this is proved, $m \leq k$ and $k \leq m$ follow easily from it since $\mathbb{N}_m \sim \mathbb{N}_k$ and $\mathbb{N}_k \sim \mathbb{N}_m$ follow easily from our assumptions and the fact that $X \sim Y$ is an equivalence relation, i.e.,

EquivRel X,Y:Type. $X \sim Y$.

So, it is enough to show that $x \leq y$, assuming that $\mathbb{N}_x \sim \mathbb{N}_y$. The connection between $X \sim Y$ and bijection is established by

$(\exists f{:}(A \rightarrow B).\ \mathrm{Bij}(A;\ B;\ f)) \Leftrightarrow (A \sim B).$

And since being an injection is part of being a bijection, i.e.,

$\mathrm{Bij}(A;\ B;\ f)\ \equiv_{\mathrm{def}}\ \mathrm{Inj}(A;\ B;\ f)\ \&\ \mathrm{Surj}(A;\ B;\ f),$

$x \leq y$ follows from the lemma

$(\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ \mathrm{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)) \Rightarrow m \leq k.$

QED

A corrolary is $\forall a,b{:}\mathbb{N}.\ (\mathbb{N}_a \sim \mathbb{N}_b) \Leftrightarrow a = b.$

7.2.1.3 Proof of the Pigeonhole principle

Show $\forall m,k{:}\mathbb{N},\ f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ k<m \Rightarrow (\exists x,y{:}\mathbb{N}_m.\ x \neq y\ \&\ f(x) = f(y)).$

That is, putting a finite collection of m objects into fewer pigeonholes ($k<m$) means there are two distinct objects put into the same hole. Without loss of generality it is enough that

$\exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y)$

i.e. that the second object y precedes the first object x. By a lemma

$\forall m{:}\mathbb{N},\ f{:}(\mathbb{N}_m \rightarrow \mathbb{Z}).\ \neg\mathrm{Inj}(\mathbb{N}_m;\ \mathbb{Z};\ f) \Rightarrow (\exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y))$

Gloss (section 7.2.1.4)

it is enough to show that

$\neg\text{Inj}(\mathbb{N}_m;\ \mathbb{Z};\ f)$

(since $f \in \mathbb{N}_m \rightarrow \mathbb{N}_k$ automatically puts $f \in \mathbb{N}_m \rightarrow \mathbb{Z}$ as well).
But according to our core lemma

$(\exists f{:}(\mathbb{N}_m \rightarrow \mathbb{N}_k).\ \text{Inj}(\mathbb{N}_m;\ \mathbb{N}_k;\ f)) \Rightarrow m{\leq}k$

Gloss (section 7.2.1.1)

$\text{Inj}(\mathbb{N}_m;\ \mathbb{Z};\ f)$ would contradict our assumption that $k{<}m$, hence

$\neg\text{Inj}(\mathbb{N}_m;\ \mathbb{Z};\ f)$.

QED

7.2.1.4 Proof of a lemma for the Pigeonhole principle

Show $\forall m{:}\mathbb{N},\ f{:}(\mathbb{N}_m \rightarrow \mathbb{Z}).\ \neg\text{Inj}(\mathbb{N}_m;\ \mathbb{Z};\ f) \Rightarrow (\exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y))$.

That is, given a non-injective assigment of integers to a finite collection we can find two objects (with the second objects y preceding the first value x) that are assigned the same value. This reduces to showing the contrapositive, namely

$\neg(\exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y)) \Rightarrow \text{Inj}(\mathbb{N}_m;\ \mathbb{Z};\ f)$

(computationally, since there are only finitely many choices for x and y in the specified domain, one could just exhaustively try them all and see if there is a pair whose assignments by f agree). So assuming that

(1) $\neg(\exists x{:}\mathbb{N}_m,\ y{:}\mathbb{N}_x.\ f(x) = f(y))$

and employing the definition

$\text{Inj}(A;\ B;\ f) \equiv_{\text{def}} \forall a_1,a_2{:}A.\ f(a_1) = f(a_2) \in B \Rightarrow a_1 = a_2$

our proof reduces to showing that if

(2) $f(a_1) = f(a_2)$

then $a_1 = a_2$. This further reduces to showing that

$\neg a_1{<}a_2$ & $\neg a_2{<}a_1$

which each follow from assumptions (1) and (2).
QED

This is a lemma for a proof of the pigeonhole principle

$\forall m{,}k{:}\mathbb{N},\ f{:}(\mathbb{N}_m\rightarrow \mathbb{N}_k).\ k{<}m \Rightarrow (\exists x{,}y{:}\mathbb{N}_m.\ x \neq y\ \&\ f(x) = f(y))$

Gloss (section 7.2.1.3)

7.2.2 One-to-One Correspondence

A one-to-one correspondence between two classes is a way of matching them member-for-member. Examples:

- At an event with assigned seating one would expect a one-to-one correspondence between the seats and the tickets, the correspondence being between each seat and the ticket printed with its location code.
- Planning for a formal dinner entails choosing a one-to-one correspondence between the guests and their places at the table.
- Counting a finite class of objects typically consists of finding a one-to-one correspondence between the items of the class and the numbers from 1 to however many there are.
- A major use for finding one-to-one correspondences is to establish, without actually counting, that some class has the same size as another class of known size.
- In programming, one often introduces, what are called variously, pointers, indices, or handles to be used in place of data objects they point to. The relation between these pointers and their objects is often expected to be a one-to-one correspondence.

Notice that a one-to-one correspondence between two classes will not be the only one if the classes have more than one member each.

As usual, this concept of a method for matching is not built-in to our growing library of mathematics, and may be introduced by definition in a variety of ways.

We shall use functions for these methods of matching: a function will be used to match each input with its output. For example, the function $f \in$

$\mathbb{N} \rightarrow \mathbb{N}$ such that $f(n) = 2{\cdot}n$, matches each natural number with its double, and is a one-to-one correspondence between the natural numbers $\mathbb{N}$, and the even natural numbers, $\{i{:}\mathbb{N} \mid \text{is\_even}(i)\ \}$.

But not every function expresses a one-to-one correspondence. For example, $f \in \mathbb{N} \rightarrow \mathbb{N}$, such that $f(i) = i$ for even i, and $f(i) = 2{\cdot}i$ for odd i. Notice that, again, every natural number gets mapped to an even number, and every even number is mapped to itself, but it is not a one-to-one correspondence because both 1 and 2 get "matched" against 2.

So, both of these functions are in $\mathbb{N} \rightarrow \{i{:}\mathbb{N} \mid \text{is\_even}(i)\ \}$, but we need something more to narrow such functions to those that are one-to-one correspondences. This can be done is various ways. See One-to-One Correspondence via Inverses (section 7.2.2.1) for the our first approach via inverse functions.

To see a discussion of the concept of Counting and its relation to One-to-One Correspondence, see Introduction to Counting (section 7.2.1).

7.2.2.1 One-to-One Correspondence via Inverse Functions

As was discussed in One-to-One Correspondence, we are trying to characterize the functions that determine one-to-one correspondences. We shall give two such "definitions."

One observation is that when a function $f \in A{\rightarrow}B$ matches A and B member-for-member, then one can reverse this function to get some $g \in B{\rightarrow}A$ which matches these classes in the opposite direction. When functions f and g have this inverse relation between them, we say they are "inverses" and write InvFuns(A;B;f;g). We define this relation between functions as

$$\text{InvFuns}(A;B;f;g) \ \equiv_{\text{def}}\ (\forall x{:}A.\ g(f(x)) = x)\ \&\ (\forall y{:}B.\ f(g(y)) = y)$$

Notice that this inverse relation between functions is symmetric, i.e.,

$$\text{InvFuns}(A;B;f;g) \Rightarrow \text{InvFuns}(B;A;g;f)$$

(just swap the conjuncts in the def)

Cancellation When in the course of reasoning one uses the fact that $(g(f(x)) = x)$ to rewrite $g(f(x))$ to x, this is sometimes called "cancellation" of f by g. Inverse functions cancel each other. For example, subtracting an integer (from something) and adding that same integer are inverse functions, so you may use one to cancel the other, by rewriting $x-a+a$ or $x+a-a$ to x.

Uniqueness As you can imagine, if a function has an inverse, that inverse is unique. Here is a theorem to that effect:

InvFuns(A;B;f;g) $\Rightarrow$ InvFuns(A;B;f;h) $\Rightarrow g = h$

As usual, some persons who have doubts about our choice of definition for InvFuns(A;B;f;g) might use this fact as further justification of the definition; whereas those who already find the definition adequate might use the proof either to complement the imagination, or resolve doubts about uniqueness.

We adopt the following definition of one-to-one correspondence:

f is 1-1 corr $\equiv_{\text{def}}$ $\exists g{:}(B{\rightarrow}A)$. InvFuns($A$;$B$;$f$;$g$)

There is another characterization of one-to-one correspondence, involving "bijection," which may well seem more obviously right. To follow this development, see One-to-One Correspondence via Bijection.

7.2.2.2 One-to-One Correspondence via Bijection

Our discussion of one-to-one correspondence started with One-to-One Correspondence (section 7.2.2), and continued in One-to-One Correspondence via Inverses, where the explanation culminated in the definition

f is 1-1 corr $\equiv_{\text{def}}$ $\exists g{:}(B{\rightarrow}A)$. InvFuns($A$;$B$;$f$;$g$)

Here we shall give a different, and to some readers more natural, "definition" of the expression (f is 1-1 corr).

We shall still use a function to express a correspondence, so we must still find a way to characterize which functions from $A{\rightarrow}B$ are one-to-one, but we shall do so in a more descriptive way than to stipulate that there is an inverse function.

Consider a function $f \in A{\rightarrow}B$ that is supposed to match A and B member-for-member. Every member of A gets paired with some member of B, but there are a couple of things that might go wrong. First, we might miss some member of B; so a one-to-one correspondence must be what we shall call a "surjection":

Surj(A; B; f) $\equiv_{\text{def}}$ $\forall b{:}B.\ \exists a{:}A.\ f(a) = b$

I.e., every member of B is reachable through f from some member of A.

The other way that a function could fail to match two classes one-to-one is if it paired two different members of one class against just one member of the other. Of course, a function can only pair one value with each argument, but in general could have the same value for different arguments, so this is what must be excluded. We call a function having a unique argument for each value an injection, and define it thus:

$$\mathrm{Inj}(A;\ B;\ f)\ \equiv_{\mathrm{def}}\ \forall a_1{,}a_2{:}A.\ f(a_1) = f(a_2) \in B \Rightarrow a_1 = a_2$$

I.e., if the outputs are equal, then the inputs are equal.

A function having both of these properties, and so one that is a one-to-one correspondence, is called a bijection:

$$\mathrm{Bij}(A;\ B;\ f)\ \equiv_{\mathrm{def}}\ \mathrm{Inj}(A;\ B;\ f)\ \&\ \mathrm{Surj}(A;\ B;\ f)$$

So, if we had not already defined (f is 1-1 corr) we could define it as being a bijection, i.e., Bij(A; B; f)

This situation is typical of making definitions. Still, we can approximate making a second definition simply by showing that

$$\mathrm{Bij}(A;\ B;\ f) \Leftrightarrow f \text{ is 1-1 corr}$$

Let us consider Bijection vs Inverses.

7.2.2.3 One-to-One Correspondence: bijections vs inverses

This discussion continues from the introduction of an alternative definition of one-to-one correspondence in One-to-One Correspondence via Bijection.

We have given three "definitions," of sorts, to the concept of one-to-one correspondence between two classes, denoted by (f is 1-1 corr).

We gave a suggestive description of the concept informally, then gave a different, purportedly equivalent, description in terms of the existence of inverse functions, then a third explanation was bijection. In order make formal reasoning about (f is 1-1 corr) possible we must somehow add new "axioms" about this predicate. We could add some new primitive axioms that declare the existence of this new predicate and some basic facts about it. But as usual, we have elected to find a combination of concepts that we can informally understand to be equivalent to one-to-one correspondence, and that we can already formally reason about.

When there are several candidates for defining a new symbol, it is typical to pick one as the principal definition, and demonstrate the equivalence of the others. In our case, we chose to use

f is 1-1 corr $\equiv_{\text{def}}$ $\exists g{:}(B{\rightarrow}A).$ InvFuns$(A;B;f;g)$

as the principal definition, and prove

Bij$(A;\ B;\ f) \Leftrightarrow f$ is 1-1 corr

as a theorem, although we could just as easily have chosen the reverse. The use of this theorem will be pretty much as if it were a definition of (f is 1-1 corr), but the "content" of the proof is essentially that being a bijection is equivalent to having an inverse.

Examination of the proof will reveal that it reduces to two lemmas expressing the opposite directions of the equivalence, namely,

Bij$(A;\ B;\ f) \Rightarrow (\exists g{:}(B{\rightarrow}A).$ InvFuns$(A;B;f;g))$

InvFuns$(A;B;f;g) \Rightarrow$ Bij$(A;\ B;\ f)$

The proof of the first theorem shows how to construct an inverse of a function given that it's a bijection. The second uses the inverse of a function to show that it is a bijection.

7.2.3 Counting Indirectly - integer ranges.

Counting the number of members of a class A is often accomplished indirectly by finding another class B that we already know how to count, and showing that $A \sim B$. This is an alternative to explicitly describing a function that counts a class directly according to Introduction to Counting (section 7.2.1). Then counting B, establishing for some $k \in \mathbb{N}$ that $B \sim \mathbb{N}_k$, justifies the inference that $A \sim \mathbb{N}_k$, since $(A \sim B) \Rightarrow (B \sim C) \Rightarrow (A \sim C)$.

Thus developing the skill of counting abstractly specified classes depends partly on developing a few basic forms that one knows how to count, and partly on developing the skill of "translating" new descriptions of classes into these forms one already knows how to count.

If you take a finite range of consecutive integers and add the same number to them all then you get a new collection with the same size as the old one.We will normally indicate such consecutive integer collections either with

$\{i\ldots j\} \equiv_{\mathrm{def}} \{k{:}\mathbb{Z} \mid i{\le}k \;\&\; k{\le}j\ \}$

or

$\{i..j^{-}\} \equiv_{\mathrm{def}} \{k{:}\mathbb{Z} \mid i \le k < j\ \}.$

The notation $\{i..j^{-}\}$, which mentions the least member and the number *after* the largest member, turns out often to be more convenient to use than the more natural notation $\{i\ldots j\}$ which mentions both ends.[1]

We shall also adopt $\{0..k^{-}\}$ as our "standard" finite class having k members, rather than the more every-day counting set $\{1\ldots k\}$. We tend to use this standard when precisely expressing facts about class size. We shall abbreviate $\{0..k^{-}\}$ as $\mathbb{N}_k$, connoting "the first k members of $\mathbb{N}$."

$\forall a{,}b{:}\mathbb{N}.\ (\mathbb{N}_a \sim \mathbb{N}_b) \Leftrightarrow a = b$

(corrolary of $(A \sim \mathbb{N}_m) \Rightarrow (A \sim \mathbb{N}_k) \Rightarrow m = k$)

Here are some theorems about the sizes of these finite integer ranges.

$\forall a{,}b{:}\mathbb{Z}.\ \{a..b^{-}\} \sim \mathbb{N}_{b-a}$

$\forall a{,}b{:}\mathbb{Z}.\ \{a\ldots b\} \sim \mathbb{N}_{1+b-a}$

Underlying these theorems is a lemma

$\forall a{,}b{,}a'{,}b'{:}\mathbb{Z}.\ b{-}a = b'{-}a' \Rightarrow (\{a..b^{-}\} \sim \{a'..b'^{-}\})$

which is proved by explicitly giving the functions that map back and forth between $\{a..b^{-}\}$ and $\{a'..b'^{-}\}$. Recall that

$\mathrm{InvFuns}(A;B;f;g) \equiv_{\mathrm{def}} (\forall x{:}A.\ g(f(x)) = x) \;\&\; (\forall y{:}B.\ f(g(y)) = y)$

and that one way to give a one-one correspondence is to provide such a pair of inverse functions. The proof of $\{a..b^{-}\} \sim \{a'..b'^{-}\}$ specifies

$(\lambda x.x{+}a'{-}a) \in \{a..b^{-}\}{\to}\{a'..b'^{-}\}$

and

$(\lambda x.x{+}a{-}a') \in \{a'..b'^{-}\}{\to}\{a..b^{-}\}$

then shows these functions to be inverses by simply computing

[1] With both notations there are infinitely many ways of specifying the empty class, by picking ends in the "wrong" order, e.g., $\{4\ldots 3\}$, $\{1\ldots \text{-}2\}$, $\{0..0^{-}\}$, or $\{0..(\text{-}3)^{-}\}$.

$(\lambda x.x{+}a{-}a')((\lambda x.x{+}a'{-}a)(x))$ down to $x{+}a'{-}a{+}a{-}a'$, which is obviously equal to x, and
$(\lambda x.x{+}a'{-}a)((\lambda x.x{+}a{-}a')(y))$ down to $y{+}a{-}a'{+}a'{-}a$, which is obviously equal to y.

Another fundamental method is to reduce a counting problem to Counting Ordered Pairs.

7.2.4 Counting Indirectly - ordered pairs

Often the objects of a class have some structure that we can exploit in our attempt to ascertain how many there are. One simple structuring method is the formation of "ordered pairs" $\langle x, y\rangle$.

For example, one can determine how many squares are on a chessboard by knowing that each square can be uniquely identified by a rank and a file, and that every rank of every file has a square. The modern convention for naming squares is to give a pair consisting of a letter from "a" to "h" and a number from 1 to 8. So, there is a one-one correspondence between squares and pairs $\langle x, y\rangle \in AH{\times}\{1 \ldots 8\}$, if AH is the class of letters from"a" to "h." There are, of course, 64 squares, but let's proceed to probe the math.

Generally the "Cartesian product" $A{\times}B$ is the class of pairs $\langle x, y\rangle$ such that $x \in A$ and $y \in B$, and for finite A and B, the size of $A{\times}B$ is the product of the sizes of A and B, or to be precise,

$$\forall a{,}b{:}\mathbb{N}.\ (A \sim \mathbb{N}_a) \Rightarrow (B \sim \mathbb{N}_b) \Rightarrow ((A{\times}B) \sim \mathbb{N}_{a \cdot b})$$

It is no accident that the conventional notation "$A{\times}B$" for the Cartesion product of two classes resembles a standard notation for multiplication of numbers. This will happen with some other notations as well.

We can convey this relation between Cartesian product and numeric product more succinctly by resorting to our standard finite classes:

$$\forall a{,}b{:}\mathbb{N}.\ (\mathbb{N}_a{\times}\mathbb{N}_b) \sim \mathbb{N}_{a \cdot b}$$

This is typical of how we shall express our core counting principles, and it is possible in this case because of the fact that

$$(A \sim A') \Rightarrow (B \sim B') \Rightarrow ((A{\times}B) \sim (A'{\times}B'))$$

i.e., the product classes are in one-one correspondence if the respective component classes are, which means we can perform rewriting of class expressions preserving size.

To resume our chessboard problem, just as we shall not try to formally make the connection between chessboard structure, and the rank file pairs, leaving this claim only informally assumed, so we shall not here formalize the class of letters “a” to “h,” and we shall assume that AH has 8 members, i.e., $AH \sim \mathbb{N}_8$.

So, we have informally “reduced” the problem of how many squares a chessboard has to how many members $AH{\times}\{1...8\}$ has for arbitrary $AH \sim \mathbb{N}_8$.

The priniciples used in the proof that follows are these:

$\forall a,b{:}\mathbb{Z}.\ \{a...b\} \sim \mathbb{N}_{1+b-a}$

$\forall a,b{:}\mathbb{N}.\ (\mathbb{N}_a{\times}\mathbb{N}_b) \sim \mathbb{N}_{a\cdot b}$

$(A \sim A') \Rightarrow (B \sim B') \Rightarrow ((A{\times}B) \sim (A'{\times}B'))$

$(A \sim A') \Rightarrow (B \sim B') \Rightarrow ((A \sim B) \Leftrightarrow (A' \sim B'))$

$A \sim A$

These equivalences have been applied and assembled into a proof of

$(AH \sim \mathbb{N}_8) \Rightarrow ((AH{\times}\{1...8\}) \sim \mathbb{N}_{64})$

Please read it along with the Remark (section 7.2.4.2) for this proof.

Here we have considered how to count pairs selected independently from two classes. This leads us to consider Counting Tuples (section 7.2.5) and Counting Dependent Pairs (section 7.2.6), where the second element is drawn from a class selected by the choice of the first element.

7.2.4.1 The Chessboard Proof

This is the formal proof mentioned in Counting Ordered Pairs.

$\vdash \forall AH{:}\text{Type}.\ (AH \sim \mathbb{N}_8) \Rightarrow ((AH{\times}\{1...8\}) \sim \mathbb{N}_{64})$ by Auto

1. AH : Type
2. $AH \sim \mathbb{N}_8$

$\vdash (AH{\times}\{1...8\}) \sim \mathbb{N}_{64}$ by Rewrite by $AH \sim \mathbb{N}_8$

$\vdash (\mathbb{N}_8{\times}\{1\ldots 8\}) \sim \mathbb{N}_{64}$ by Rewrite by Thm$*$ $\forall$a,b:$\mathbb{Z}$. $\{\mathrm{a}\ldots\mathrm{b}\} \sim \mathbb{N}_{\mathrm{1+b-a}}$

$\vdash (\mathbb{N}_8{\times}\mathbb{N}_{1+8-1}) \sim \mathbb{N}_{64}$ by Reduce Concl

$\vdash (\mathbb{N}_8{\times}\mathbb{N}_8) \sim \mathbb{N}_{64}$ by Rewrite by Thm$*$ $\forall$a,b:$\mathbb{N}$. $(\mathbb{N}_{\mathrm{a}}{\times}\mathbb{N}_{\mathrm{b}}) \sim \mathbb{N}_{\mathrm{a\cdot b}}$

$\vdash \mathbb{N}_{8\cdot 8} \sim \mathbb{N}_{64}$ by Reduce Concl

$\vdash \mathbb{N}_{64} \sim \mathbb{N}_{64}$ by Auto

7.2.4.2 Remarks on the Chessboard Proof

We discuss The Chessboard Proof.

$(AH \sim \mathbb{N}_8) \Rightarrow ((AH{\times}\{1\ldots 8\}) \sim \mathbb{N}_{64})$

By a series of rewrites and arithmetic simplifications (the Reduce tactic), the goal is successively reduced down to $\mathbb{N}_{64} \sim \mathbb{N}_{64}$, which is implicitly justified by Auto using $A \sim A$,

The theorems

$\forall a,b{:}\mathbb{Z}.\ \{a\ldots b\} \sim \mathbb{N}_{1+b-a}$ and

$\forall a,b{:}\mathbb{N}.\ (\mathbb{N}_a{\times}\mathbb{N}_b) \sim \mathbb{N}_{a\cdot b}$

are invoked explicitly for rewriting, as is the assumption $AH \sim \mathbb{N}_8$. For example, the last rewrite reduces the goal

$(\mathbb{N}_8{\times}\mathbb{N}_8) \sim \mathbb{N}_{64}$ to

$\mathbb{N}_{8\cdot 8} \sim \mathbb{N}_{64}$

rewriting one part by matching against

$\forall a,b{:}\mathbb{N}.\ (\mathbb{N}_a{\times}\mathbb{N}_b) \sim \mathbb{N}_{a\cdot b}$

to get the instance

$(\mathbb{N}_8{\times}\mathbb{N}_8) \sim \mathbb{N}_{8\cdot 8}$.

The principle $(A \sim A') \Rightarrow (B \sim B') \Rightarrow ((A \sim B) \Leftrightarrow (A' \sim B'))$ is also implicitly used to justify performing this rewrite inside the "? $\sim$?" operator, as is the principle $(A \sim A') \Rightarrow (B \sim B') \Rightarrow ((A \sim B) \Leftrightarrow (A' \sim B'))$ for rewriting inside the "?×?" operator.

A person assembling a proof without this strict top-down method, might establish the following sequence of facts:

$AH \sim \mathbb{N}_8$	[by hypothesis]
$\{1...8\} \sim \mathbb{N}_8$	[$\forall a,b{:}\mathbb{Z}.\ \{a...b\} \sim \mathbb{N}_{1+b-a}$ and arithmetic]
$(AH\times\{1...8\}) \sim (\mathbb{N}_8\times\mathbb{N}_8)$	[$(A \sim A') \Rightarrow (B \sim B') \Rightarrow ((A\times B) \sim (A'\times B'))$ and above]
$(\mathbb{N}_8\times\mathbb{N}_8) \sim \mathbb{N}_{8\cdot 8}$	[$\forall a,b{:}\mathbb{N}.\ (\mathbb{N}_a\times\mathbb{N}_b) \sim \mathbb{N}_{a\cdot b}$]
$\mathbb{N}_{8\cdot 8} \sim \mathbb{N}_{64}$	[arithmetic]
$(AH\times\{1...8\}) \sim \mathbb{N}_{64}$	[chaining through the last 3 facts using $(A \sim B) \Rightarrow (B \sim C) \Rightarrow (A \sim C)$]

7.2.5 Counting Indirectly - tuples

A distinguishing feature of ordered pairs of objects discussed in Counting Ordered Pairs (section 7.2.4), as opposed to unordered pairs, which we have not discussed, is that two ordered pairs can have the same components, but in reverse order. Also, an ordered pair might have the same object occupying both its places.

This can only happen if the classes from which the first and second components are drawn overlap. That couldn't happen for the modern chessboard notation of Counting Ordered Pairs (section 7.2.4) because letters fill the first place and numbers fill the second. But of course that was not necessary – the designers of the chess notation *could* have decided to use numbers from $\{1...8\}$ instead of letters, and simply required that the first number indicates the file and the second the rank; that would be harder to read though, and people might have trouble remembering the order of rank and file. So $\neg(\langle 1,2\rangle = \langle 2,1\rangle)$, and $\langle 2,2\rangle \in \mathbb{N} \times \mathbb{N}$.

We won't go into it here, but to emphasize the distinction, and to portend subtle issues of how to distinguish objects to be counted, consider a buffet that, for a fixed price, allows you to have two servings of a topping on rice, your choice. While you might order chicken before you order tofu, or tofu before chicken, these would normally be understood as two ways to order the same meal. The order you place might be considered an ordered pair of toppings, but the meal you choose would not. So our methods so far tell us how many ways we may place an order, but not how many meals we have to choose from. (My experience in these businesses is that you are permitted to have two of the same topping, so you cannot just say there are two ways of specifying each meal.)

Back to ordered pairs. Suppose we want to consider ordered triples. Do we need to redevelop the machinery for counting pairs from the start, and then again for 4-tuples, etc? Well, we certainly could add triples to our precise mathematical language. If we were to add triples we would soon start trying to adapt our facts about pairs by recognizing that you can "think" of a triple (x,y,z) as a pair $\langle x, \langle y, z\rangle\rangle$, whose first component is the first of the triple, but whose second component is itself a pair $\langle y, z\rangle$, built of the second and third components of the triple. Not only can you "think" of triples that way, but you could define a bijection between $A{\times}(B{\times}C)$ and the corresponding "triple-product." Naturally, once you do this you're ready to iterate the method to 4-tuples and so forth.

In fact, using the iterated pairs to simulate tuples has proven to be so convenient and simple that it is normal for mathematical systems to stop at the pairs and binary products and consider the higher tuples as informal concepts, or perhaps as themselves being "defined" as iterated pairs. We will do this, too.

$\langle x, y, z\rangle$ is just a suggestive notational abbreviation for $\langle x, \langle y, z\rangle\rangle$

$\langle u, x, y, z\rangle$ is just a suggestive notational abbreviation for $\langle u, \langle x, \langle y, z\rangle\rangle\rangle$

etc.

$A{\times}B{\times}C$ is just a suggestive notational abbreviation for $A{\times}(B{\times}C)$

$A{\times}B{\times}C{\times}D$ is just a suggestive notational abbreviation for $A{\times}(B{\times}(C{\times}D))$

etc.

We could have done something similar instead, such as grouping the tuple components through the other position, i.e., letting $\langle\langle x, y\rangle, z\rangle$ be our encoding of the triple (x,y,z), but these pairs are different objects and so we ought to stick to a convention.

Indeed, the obvious conversion between left-nested and right-nested triples is exploited to prove this theorem: $(A{\times}B{\times}C) \sim ((A{\times}B){\times}C)$ Remark

Amusingly, by combining this purely structural fact with

$\forall a{,}b{:}\mathbb{N}.\ (\mathbb{N}_a \times \mathbb{N}_b) \sim \mathbb{N}_{a\cdot b}$

and

$\forall a{,}b{:}\mathbb{N}.\ (\mathbb{N}_a \sim \mathbb{N}_b) \Leftrightarrow a = b$

one can show $\forall a{,}b{,}c{:}\mathbb{N}.\ a{\cdot}b{\cdot}c = (a{\cdot}b){\cdot}c$.

7.2.6 Counting indirectly - dependent ordered pairs

Now we advance from Counting Ordered Pairs (section 7.2.4) and Counting Tuples to a more complex structural possibility. What if we want to count a collection of pairs, but the collection is not simply the Cartesian product of two (or more) other collections? What if the choices permitted for the second component depend on the choice already made for the first component?

Suppose you want to determine the number of pairs of persons (from a given group) where the second person must be younger than the first? Or suppose you want to count the number of ordered pairs from this group where the person in the second position must simply be someone other than the one in the first.[2]

This way of limiting a class of pair constructions is denoted "$x{:}A{\times}B(x)$".

$\langle a, b\rangle \in x{:}A{\times}B(x)$ just when $a \in A$ and $b \in B(a)$.

The "x" of "$x{:}A{\times}B(x)$" is a binding variable that binds in "$B(x)$", just like the quantifier notations "$\forall x{:}A.\ B(x)$" and "$\exists x{:}A.\ B(x)$". The position of the $B(x)$ in $x{:}A{\times}B(x)$ should be occupied by an expression that stands for a class, no matter what expression of type A is put for x. The collection from which the second component b is chosen may DEPEND on the VALUE of the first component a. Examples:

- $(x{:}G{\times}\{y{:}G|\ age(y){<}age(x)\ \})$ is the collection of age-restricted pairs of persons described above where G is the collection of relevant persons.

[2] There's nothing physical intended about these "positions." Maybe there's only one cupcake and one cookie left, and you want to count the number of ways to dole them out. Then we may stipulate that the first "position" is getting the cookie, and the second is getting the cake.

- (x:G×{y:G| ¬$x = y$ }) is the collection of no-duplicate pairs of persons described above.
- (x:G×y:G×{z:G| ¬$x = y$ & ¬$y = z$ & ¬$z = x$ }) is the collection of no-duplicate triples, construing triples as nested pairs. (ala Counting Tuples (section 7.2.5))
- (i:ℕ×$\mathbb{N}_i$) is the collection of pairs of natural numbers where the second is less than the first. There is no $\langle 0, k\rangle \in i{:}\mathbb{N} \times \mathbb{N}_i$ since $\mathbb{N}_0$ is empty.

The standard notation for this operation on classes is "Σx:A. $B(x)$". The similarity to the standard notation for summing a numeric function over a range of values, such as "Σ i:{k..m^-}. $f(i)$" , is again no accident. Here the two concepts are related:

$$\forall a{:}\mathbb{N},\ b{:}(\mathbb{N}_a \rightarrow \mathbb{N}).\ (i{:}\mathbb{N}_a \times \mathbb{N}_{b(i)}) \sim \mathbb{N}_{\Sigma\ i{:}\mathbb{N}a.\ b(i)}$$

The numeric sum notation "Σ i:ℕk. $f(i)$" is just an abbreviation for "Σ i:{0..k^-}. $f(i)$". The notational similarity could be either confusing or helpful. The two notations could not logically be used in the same contexts since they take different kind of arguments and have different kinds of values:

> The value of a numeric summation expression "Σ i:{a..b^-}. $f(i)$" is numeric, and the functional argument $f(i)$ is numeric.
> The value of a dependent-pair type expression "x:A×$B(x)$" is a class whose members are pairs, and the functional argument $B(x)$ stands for a family of classes, one for each member of class A.

The priniciple above very nearly tells us that if we assign to each *member* of a finite class, another finite class, then we can count the pairs of the dependent-pair-type by simply adding up the sizes of all the classes in the family of classes.

7.3 Formal Implementation of the Red–Black Trees

In this section we will show an example how one can formally define an abstract data structure in the formal constructive type theory (implemented in MetaPRL, [78]), then give an efficient implementation of this data structure (or possibly different implementations of the same abstract data structure), and formally prove that the implementation satisfies the intended specification of the data structure.

We will consider an example of the popular data structure *Set*, which represent collections of elements of a certain type. There are different implementations of this data structure. For example, one can implement sets using some sort of balanced binary search trees. The most popular balanced binary search trees are red–black trees [73]. We will show how the implementation of red–black trees could be written as a term in type theory, and provide the type for this term.

7.3.1 Dependent Record Type

To represent data structures we need *dependent records*. In general, records are tuples of labeled fields, where each field may have its own type. In dependent records (or more formally dependently typed records) the types of components may depend on values of the other components. Since we have the universe type (a type of types) $\mathbb{U}$, values of record components may be types. This makes the notion of dependent records very powerful. Dependent records may be used to represent algebraic structures (such as groups) and programming data structures.

For example one can define the signature for ordered set as a dependent record type:

$$OrderSig = \{\mathtt{car} : \mathbb{U}; \mathtt{less} : \mathtt{car} \longrightarrow \mathtt{car} \longrightarrow \mathbb{B}\}$$

(where $\mathbb{B}$ is the boolean type). This definition can be understood as an algebraic structure as well as an interface of a module in a programing language.

We can also define ordered sets as structures with the above signature satisfying the axioms of ordered sets:

$$Order = \{\mathtt{car} : \mathbb{U}; \mathtt{less} : \mathtt{car} \longrightarrow \mathtt{car} \longrightarrow \mathbb{B}; OrdRelation(\mathtt{car}, \mathtt{less})\}$$

where $OrdRelation(T, <)$ is a predicate stating that $<$ is a linear order relation on T.

Dependent record types could be defined in type theory [100]. They allow us to represent signatures and data structures as ordinary types in the type theory.

7.3.2 Abstract Data Structures

Basically a data structure is some type with basic operations on elements of this type that satisfy some axioms. Let us consider for example the data structure $Set(T)$ for collection of elements of type T. To define the data structure for sets we need to provide the type carrier (`car`) for elements that will represent sets and some basic operations such as `insert`, `delete` and `member`. These operations have the certain type. For example `member` tests whether an element is a member of a set. That is, it is a function of the type $\mathtt{car} \rightarrow T \rightarrow \mathbb{B}$.

Thus we can give the *signature* for the data structure $Set(T)$. We say that the signature is the following *record* type:

$$
\begin{aligned}
&Set\_sig\{T\} = \\
&\quad \{\mathtt{car} : \mathbb{U}; \\
&\quad \mathtt{empty} : \mathtt{car}; \\
&\quad \mathtt{member} : \mathtt{car} \rightarrow T \rightarrow \mathbb{B}; \\
&\quad \mathtt{insert} : \mathtt{car} \rightarrow T \rightarrow \mathtt{car}; \\
&\quad \mathtt{delete} : \mathtt{car} \rightarrow T \rightarrow \mathtt{car}\}
\end{aligned}
$$

This signature is just an particular type in the type theory. Implementations of the data structure are elements of this type. This is similar to signatures in ML-like functional programming languages.

But not all structures of this signature adequately represent actual set structure. The set data structure should satisfy some axioms (specifications). For example, for any set S and for any elements b, the set $\mathtt{insert}\ S\ b$ should contain all elements of S and b and nothing more. Usual typed programming languages could not express those axioms. Our type theory is powerful enough to define the type of set data structures satisfying these axioms. We will glue in those specification in the definition of the data structure *Set*. That is, we define $Set(T)$ to be a set of structures of signature $Set\_sig(T)$ satisfying certain specifications. Thus, $Set(T)$ is a subtype of $Set\_sig(T)$. The formal definition is the following:

$$
\begin{aligned}
&Set\{T\} = \\
&\quad \{\mathtt{car} : \mathbb{U}; \\
&\quad \mathtt{empty} : \mathtt{car}; \\
&\quad \mathtt{member} : \mathtt{car} \rightarrow T \rightarrow \mathbb{B};
\end{aligned}
$$

```
insert : car → $T$ → car;
delete : car → $T$ → car  |
(∀$a$ : $T$. ¬(↑ (member empty $a$)))  |
(∀$S$ : car
     ∀$a$ : $T$
          ∀$b$ : $T$
               (member (insert $S$ $b$) $a$)  ⇔  (member $S$ $a$)
                ∨  ($a$ = $b$ ∈ $T$))  |
(∀$S$ : car
     ∀$a$ : $T$
          ∀$b$ : $T$
               (member (delete $S$ $b$) $a$)  ⇔  (member $S$ $a$)
                ∧  (¬($a$ = $b$ ∈ $T$)))}
```

More discussion could be found in [99, 82].

7.3.3 Implementations of Data Structures

We can implement the set data structure in several ways. The simplest but inefficient implementation of sets uses lists. Each set is represented by an unordered list. Formally we take `car` to be T List, `empty` to be *nil* and define operations `insert`, `delete` and `member` correspondingly. In this implementation functions `insert`, `delete` and `member` take $O(n)$ time, where n is a number of elements of the set.

A more efficient implementation of sets is binary search trees. Each set is represented by a binary tree, where elements stored at the nodes, such that the element at any given node is greater than each element in its left subtree and less than each element in its right subtree. In this implementation functions `insert`, `delete` and `member` take $O(h)$ time, where h is a height of the tree. On random data the heights of the tree is $log(n)$. But in worst case the tree will be very imbalanced, and an individual operation will take up to $O(n)$ time.

The solution to this problem is to implement balanced binary trees. In the next section we will consider one of the most popular balanced binary trees, red–black trees [73].

7.3.4 Red–Black Trees

In a red–black tree each node is colored either red or black. A red–black tree should satisfy the following invariants:

- Any child of a red color is black
- All paths from the root to any leaf have the same number of black nodes.

It follows from these invariants that the height of any red–black tree h is less or equal than $2\log(n)$, where n is a number of nodes. Therefore searching in this tree takes $O(\log n)$ time. It is also alway possible to insert and delete elements from the tree, and then rebalance it to keep the invariants. It also takes $O(h) = O(\log n)$ time. So any single operation in this implementation takes $O(\log n)$ time.

The red–black tree could be defined on a *ordered* type T. We should write a *functor* from structure *ord* of the type *Order* to a structure *rbtree_str* of the type $Set(ord.\mathtt{car})$. In our setting functors are just functions. That is, to provide the implementation of red–black trees we must to provide a function $rbtree\_str(ord)$ of the type

$$ord : Order \longrightarrow Set(ord.\mathtt{car})$$

The proof of the statement

$$rbtree\_str \in ord : Order \longrightarrow Set(ord.\mathtt{car})$$

is a proof of the correctness of the implementation.

We use implementation of red-black trees in a functional programming setting similar to [145]. We will not show here the whole implementation. Interested readers can find it in [99, 82] To give the flavor of this implementation we just show the insert function that maintains the red–black tree invariants.

```
$ins\{a;\ t;\ ord\} \longleftrightarrow$
    $match\ t\ with$
        $NIL -> \ tree(\{a;$
                    left $=$ $NIL;$
                    right $=$ $NIL;$
```

```
                color $=$ $red\})$
    $|$ $L.R.tree(self)$ $->$ $Compare\ in\ ord:$
            $a$.data $<$ $self$.data $->$ $lbalance$
            $\{\{self;$ left $=$ $L\}\}$
            $a$.data $=$ $self$.data $->$ $tree(\{a;$
            left $=$ $self$.left;
            right $=$ $self$.right;
            color $=$ $self$.color$\})$
            $a$.data $>$ $self$.data $->$ $rbalance$
            $\{\{self;$ right $=$ $R\}\}$
```

When we insert a new node we color it red. This may break the invariant stating that any child of red node should be black. To repair such possible violation we run a balance function. The `lbalance` and `rbalance` functions check all possible cases of violation of the invariant and rebalance the tree to satisfy the invariant. One can find these functions in [99, 82].

Chapter 8

Future Plans

In this chapter, we consider our plans for the next three years. These plans are informed by some of the discoveries we made over the past 27 months and the new capabilities we have created and explored tentatively.

Vision The goal that inspires us for the long term is nearly a decade away (seven years with strong funding for the area, perhaps less with exceptional funding). There will eventually be a federated digital library of formal and informal algorithmic knowledge from the US, EC, and Japan, call it the FDL for now. To it, a dozen or more interactive theorem provers will be connected, and several of those will include powerful decision procedures and fully automated provers such as JProver, Otter, and EQP. These provers will have contributed over 100,000 formally proved theorems that support many detailed models of hardware, virtual machines like the JVM, and semantic accounts of programming languages like Java and ML that have relatively clean semantics. *The provers will draw computing power from the grid and knowledge from the FDL and thus extraordinary proving power will be available as needed to protect our software infrastructure.*

CIP applications The FDL and the grid will be used to support several critical infrastructure software systems – these will be large (a million lines and up) and distributed. Their core functionality will be hardened by formal verification and checking, and documentation will be an interactive FDL-supported mixture of formal and intuitive knowledge (hyfi documents). The system and it documentation will be part of a logical version control system (LVC) of which the FDL will be an integral part. The LVC will guarantee that all changes to the critical parts of the systems remain secure and that the documentation is completely current and formally correct.

When extension are made or when new vulnerabilities in the unhardened parts are detected, we will be able to coordinate via the FDL a dozen provers and support groups to quickly harden a section in a few months – this would translate into several years measured in terms of 2004 computing and information capabilities. Information-intensive and computation-intensive verification technology will provide an advanced response capability that is not practical now. It is unlikely that the federal government could create a central facility in less than a dozen years that could do the same job. It could not be created without the research efforts in which we are engaged.

The resulting technology will be very advanced, and this will be a further deterrent to terrorist attack on our infrastructure of the kind we sketched in Chapter 1.

DoD systems Military systems will be among those protected by the new verification technology, e.g. JBI and UAV. As with network centric warfare, we will be years, perhaps decades, ahead of our enemies. We must be preeminent and unsurpassed in this technology because we are the most vulnerable. We must explore and prepare all the promising technologies for high reliability and security in software systems. In this case, our research group is a key part of one of the few scientific approaches to CIP/SW. We would ask the question, Is there a better scientific vision for a more secure future? We are asking for an additional $1.8M to continue this promising approach – this is about 15 minutes of effort per year at the rate of the cost of the errors in day-to-day business. It is only a few seconds at the cost of a major disaster or the kind we imagined in Chapter 1 – a cost that could exceed $60 billion per month.

Benefits to science An FDL will benefit science generally because a great deal of the collected formalized mathematics will be general purpose. The European Community efforts will contribute deep theorems such as the Fundamental Theorem of Algebra, the Prime Number Theorem, the Graph Minor theorem, and so forth. The Semantic MathML language that we will help improve will allow physicists, chemists, biologists, and engineers to use elements of mathematics in their models, markup languages, problem solving environments, and libraries. For example, the Chemical Markup Language, CML, needs mathematical elements. The Library of Living Systems of the NSDL will reference mathematical results. Scientists will be able to search for needed mathematical theories and even modify existing models to suit their needs.

Benefits to computer science Physical scientists observe natural phenomena and ask 'why?". Computer scientists imagine what computers might do for people and ask "why not?". The FDL will allow more informed and systematic exploration of systems of the future. It will be a space in which precise high-level descriptions of systems will be kept. In addition, like the Library of Living Systems, by 2020 we will be able to produce a Library of Artificial Systems, including intelligent systems. From these scientists will be to imagine more boldly and reach further. There will be a credible effort to preserve the best of our experimental systems in an executable state; even reference algorithms and systems will execute fast enough in thirty years to give the experience of actual performance.

Research component We are exploring now those capabilities needed to make the FDL of 2010, not merely the FDL of 2004. We have discovered capabilities and services that are highly plausible to implement in the next few years and which will show the way toward and nucleate the systems and communities that must be formed.

8.1 New capabilities

The new capabilities we want to explore arise from the existence of formal mathematical content from multiple systems stored in a uniform abstract syntax and easily accessible to a variety of basic library services implemented in the FDL.

We have also laid the groundwork for building an experimental reference distributed system and an experimental formal compiler on which we can bring together results from HOL, PVS, Isabelle, and Nuprl. We will be adding considerable content in this category based on our verification work.

We have explored the value of a new kind of object for the FDL, the hybrid formal/intuitive document, *hyfi document*. It is based on the observation that it is easier to share definitions, theorems, and reference algorithms than it is to share proofs. So, we created the kind of document that will allow people to more easily include formal definitions, theorems, and algorithms in their articles. We intend to build substantially more library services for hyfi documents.

We have produced experimental translations services that we intend to exploit and make part of an advanced service suite.

We have explored a technique for theory modification that works in multiple passes. We will see whether this can be made a service as well.

We have begun a more systematic study of clustering methods based on the hyperlink reference structure and on citations. [96]

8.2 Future plans

We are ahead of the research schedule agreed upon in our proposal, so our plans now are a bit more ambitious. Moreover, our work has benefited from corresponding work in the EC and from newly funded NSF work. So there are more opportunities to examine as we plan for the final two years.

Goals: We want to attract more content providers so that by the end of the five year effort, several groups are submitting content including PVS users in distributed protocol verification, HOL users in program extraction, and so forth. This will be a challenge since there are political and institutional barriers to be overcome. We will need to create the position of a collector as we proposed. We will need to work with groups such NRL and NASA and other universities to facilitate collection of relevant content from various provers. Our ties to the NATO science community in the European Community will be of significant help, especially our strong ties to Munich and Potsdam.

We will continue to use the FDL directly in a CIP activity, and we hope that at least one other MURI will as well. This is extremely likely given the current state of activity. For these efforts and the related content collection, we must remain active players in the TPHOLs community while keeping our connections with IJCAR.

We want to be a major player in the European Community effort in this area, which will surely include OMDoc and Helm. Our goal is to have at the end of five years a federated mathematics library of which our FDL is a part. To achieve this we must remain active members of the Mathematical Knowledge Management (MKM) community and its North American branch as well. We have offered to host a meeting in Ithaca, and we will request funds for this purpose.

We want to attract authors of mathematics and computer science articles. We will attempt to use the FDL in the context of the FDL to do this. For this activity, it will be important to become active in the National Science Digital Library community, and we have begun attending meetings. We also have funding from NSF for this activity.

We want to attract dataminers and machine learning experts to search the FDL for interesting patterns.

Here is a breakdown of our plans for the future, according to the four categories of our work. It is clear that progressively more of our effort will go into experiments with the FDL and the services that we can provide across the collections. Some of these services are already remarkable as we will demonstrate at the review.

Content foundations In this category, we will attempt to extend Moran's isomorphism to bring new proof methods from set theory into type theory, thus further combining the proof theoretic power of set theory with the expressiveness of type theory especially for structuring and combining systems.

We will continue our work with event systems and protocol verification, including our work with other CIP/SW MURI's. In this topic area we will be aided by a sabbatical visitor, Uri Abraham who has written a book on distributed system verification [2]. He also happens to be an eminent set theorist, so he might help with the first topic.

We will continue our theoretical work on and writings on reflection, looking to eventually provide it as an operator on FDL formulas. This will be an exceedingly useful service to all systems that are hosted in the FDL.

Content experiments The corporation ATC-NY has offered to pay us to include a large amount of formal mathematics that they produced using the Larch prover. It includes general mathematics and hardware models. We have worked out a detailed plan for collecting this material in the FDL. It will add nearly a thousand definitions and proofs to the FDL in standard multi-sorted first-order logic. Such material can interoperate with the other collections, and it includes many data structures such as sets, bags, lists, etc. and advanced results in set theory (with classes), and numerous specifications of computing tasks. It also includes a formal version of VHDL.

We intend to add more content directly with HOL, and we are hoping that the OMDoc collaboration will provide us with Coq libraries and Mizar. We need to see how this plays out before we know for sure how Coq will be supported, since we are prepared to import it directly if necessary. We will also continue adding Nuprl and MetaPRL content that is related to software support. Some of our content will be like red/black trees and leader election, namely reference algorithms and protocols. We expect to continue the graph theory and use it in the distributed systems theory.

We intend to collect further results from ACL2. HOL, Isabelle, and PVS on protocol verification and security [21, 15, 14, 13, 57, 62, 84, 105, 112, 4, 175, 104, 159] We will consider various ways in which we can share formal specifications and computing models, such as IOA and Message Automata.

We will bring more of the FDL's capabilities to bear on the support of experimental systems such as a formal compiler and a reference distributed system.

We intend to experiment with knowledge based security protocols as part of our work with Joe Halpern and Sabina Petride.

The ATC-NY corporation will fund the restoration of many Larch libraries. These include many theories about data structures, set theory, and properties of VHDL. This activity illustrates another value of the FDL – it can restored the fruits of a very large human effort and make them available indefinitely.

We intend to make the editing of hybrid documents, those with formal and intuitive content, extremely attractive in the FDL so that anyone writing a paper in the fields of computing and mathematics we cover will want to draw material from the FDL. We think that our basic services will attract others to help us enhance these services because they will extremely useful to authors of technical material, much in the way that Google is used now to find references and insert them into text.

Infrastructure foundations We will continue to explore the reflection capability as a basic service on FDL formulas. We will explore the addition of editing services that take advantage of more than one binding structure for terms. We will take up the implications of reflection services for the existing accounting mechanisms.

We will explore the MetaPRL notion of theory implementations to formulate translation results such as those of Howe and Moran cited above. This is one of the most plausible ways to record these metamathematical results without a full-scale reflection of a logic.

We may have time to examine a possible embedding of ACL2 [94] into CTT or into a MetaPRL theory, exploiting the constructive character of ACL2.

Infrastructure experiments Our approach to experimenting at this stage is to proceed breadth first and look several aspects of the FDL before going deeply into one. We have experimented with trying to load PVS using the binding structure of the FDL terms and by not supporting binding structure at the lowest level. We found that later approach more flexible and effective. We have built API's and know better what is required to make them more useful. We have harvested metadata on logical dependencies and know what help we want from contributors. We have experimented with clustering algorithms and annotation schemes to aid search, and we have used pattern based search. We know that we want to explore clustering greater depth.

We have stored reference algorithms from three systems, and provided good formal documentation. We have written articles that are semantically anchored in the FDL and connect intuitive and formal knowledge. We are assembling the elements of a distributed system in the FDL and exploring the support of both design and verification.

We have used two systems to develop protocols and we have tried to bring PVS theorems to bear on this problem. These efforts taught us about technical problems and the political problems of acquiring material. Even sharing tasks between different implementations of the same logic has shown the need for certificates in cases that seemed simple without them. We have explore a translation service and see its value and limitations. We have maintained a real software system inside the MetaPRL library and know that formal documentation is useful.

We will continue to incorporate Allen's ideas for abstract object identifiers, closed maps, operations on these maps, and certificates more deeply into the FDL mechanisms.

We will further automate FDL submission services and expand the API's accordingly.

We will provide more dynamic pure structure editing on the Web using the DPS editor that we are now using internally.

We will consider a separate database implementation of the core FDL services.

We will provide additional clustering and search methods.

Chapter 9

Conclusion

Here we summarize the reasons we think that DoD, ONR, and science will benefit from funding the two-year option to our project.

9.1 Summary of our case

Here are seven strong reasons why we should be given the optional two years of funding:

1. Our proposal and results squarely meet the BAA requirements.
   In Chapter 2, we presented the case that we have made significant progress on all ten of the research concentration areas of the BAA.
2. Our work contributes a missing piece of verification technology important to CIP/SW and to Navy missions (e.g. NRL work).
   In Chapter 3, we made an extensive case for the value of the FDL in protecting the nation's and the Navy's software. The FDL is the basis of an information network which will complement the computing grid. There is basically no such work in the US except ours, yet it is attending to a serious deficiency in our nation's ability to protect its infrastructure.
3. Our work is already being used to help in DoD missions, including Navy and Air Force missions, for example: AFOSR work in IAI, and support for NRL work on IO automata in PVS.
   Not only are we helping DoD missions, we are working with two other MURI projects. This indicates the fundamental nature of our work. In Chapter 7 we demonstrate work on protocol verification using state

machines similar to IO Automata that is closely related to efforts at the Naval Research Laboratory. Our work is valuable to the Navy for the very same reasons that the NRL work is, we help create reliable and adaptable software.

4. We are demonstrably very productive in all categories mentioned in the BAA.
   We have produced a good level of ordinary scientific research, and in addition we have built a substantial new system and experimented with it in scientifically sound ways.
5. We have been effective in opening a new fundable research area that no other agency is funding at present.
   We were the first US project in this area, now the NSF is funding three others that support it. We have been invited to speak about our work at leading centers of computer science research.
6. Our continued involvement will speed progress toward the DoD goals articulated in the BAA.
   We are leaders in this new area and have invested a large amount of our discretionary resources into this project. At Cornell it is our top priority project.
7. We are highly qualified for this work, and our progress and results are excellent, as will become clearer as time goes on; for now we can point to good conferences, invited lectures, and followers.
   Our qualifications are evident from the fact that our proposal was the only one funded in its category.

9.2 Discussion

The Cornell/Caltech/Wyoming ONR MURI project was top rated and the only one funded in its category; in May 2002 it passed well a full day technical review by a ten person external panel that was encouraging of our effort. Since then we have added extensive new services to the software system we built, called a Formal Digital Library (FDL). The FDL is a unique resource – based on extensive design discussions recorded in notes presented in Chapter 4. Recently we have also made a significant scientific breakthrough in discovering unexpected compatibilities and relationships between the logical theories we expect in the FDL [133, 45].

Essentially we are ahead of schedule with results found nowhere else in the world. We have been asked to speak about our work at Berkeley, Stanford, Edinburgh and at five conferences, four in Europe where there is related activity; one was a meeting of the Association for Symbolic Logic. There is basically no such work in the US except ours, yet it is attending to a serious deficiency in our nation's ability to protect its infrastructure.

The ATC-NY corporation has agreed to fund the inclusion of extensive formal mathematical files it owns into the FDL for use in the public domain. This shows industrial use of our product by the end of the second year. The NSF is funding work connecting our FDL to the National Science Digital Library, showing the increasing impact of our work and a recent peer evaluation of its quality. The European Community effort to create a Web based digital library of mathematics wants to cooperate with us and use the FDL.

It is increasingly clear that ONR's initiative in this area will be recognized for its pace setting investment in innovative technologies – two years ahead of the NSF and the EC.

Based on the current state of this project and on our May 2002 full day review and on subsequent endorsements from the scientific community since then, there is solid justification for continuing this project for the full five year proposed term.

We think that ONR has a major opportunity to advance basic science in service of the Navy, DoD, and national security. Cornell, Cal Tech, and the University of Wyoming have a major opportunity to work with a level of funding that is uncommon for DoD supported basic science. The subject of the project is our highest priority and one for which we are especially well qualified.

Chapter 10

Glossary of FDL Terminology

Short Glosses

Abstract Identifiers	identifiers treated abstractly and as atomic, serving as **Object** names in a **Closed Map**
Certificate	an **Object** attesting to some fact established by the **FDL** process
Client	an entity, such as a person, supposed normally to exist and perhaps persist outside the **FDL** process, with which the **FDL** interacts **Sessions** and for which it maintains certain privileges
Closed Map	a finite collection of named **Objects**; the names are **Abstract Identifiers**, the contents are **Texts**; objects can refer to objects
Current Closed Map	the main part of the state in a **Session** with a **Client** of the **FDL**, 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 **FDL**; "external" emphasizes the contrast with abstract identifiers which are "internal" to an FDL
FDL	Formal Digital Library; a repository of a certain kind with a process for using it
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 **Propositions**
Name Server	an association of "names" with **Closed Maps** or **Objects**
Native Language	a programming notation executable by the **FDL** process
Object	the unit of **FDL** content; abstractly named **Texts**
Pro-textual Constituents	the constituents of **Texts** that are not themselves texts
Proof	a complex of appropriately related **Inference Steps**
Proposition	an expression used as a conclusion or premise of an **Inference Step**
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; the unit of coherency in **Proofs**
Session	a subprocess of the **FDL**, with associated state, for communication with a **Client**
Tactic	a program describing an inference by composing primitive inferences
Term	synonym for **Text**; the main form used in the **FDL** for structured data such as expressions
Text	the main form used in the **FDL** for structured data such as expressions
Text Server	the aspect of an **FDL** as a process exchanging **Texts** with **Clients**

Long Glosses

Abstract Identifiers

[identifiers 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 **Texts** 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 (or possibly by comparing their referents). See Abstract Ids & Closed Maps (section 4.4.1) and see **Text Server**.

Certificate

[an **Object** attesting to some fact established by the **FDL** process]

As the name suggests, a certificate attests that certain **FDL** 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 **FDL** process following a procedure specific to the "kind" of certificate in question. See Certificates (section 4.5.1).

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 **FDL** 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 the 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 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. See Proof Organization (section 4.5.11.1).

A certificate will be reconsidered either by explicit demand or automatically when any **Object** it refers to is modified. See Altering Certificates (section 4.5.9.1).

Client

[an entity, such as a person, supposed normally to exist and perhaps persist outside the **FDL** process, with which the **FDL** interacts **Sessions** and for which it maintains certain privileges]

A client accesses and modifies the **FDL** via a **Session**. A client is represented in the FDL as a persistent FDL object, specifically, as a **Certificate** associated with a **Native Language** procedure for defining a criterion for inserting the client identifier into the "client-slot" of a **Session**. That is the basic computational significance of client identity, and further significance of client identity is attributed by persons through their understanding of this client identification procedure.

The privileges assigned to a client include access to certain FDL objects maintained for the client, and the maintenance of **Name Server** objects allowing the client to specify an association of **External Names** with **Closed Maps** or **Objects**.

Closed Map

[a finite collection of named **Objects**; the names are **Abstract Identifiers**, the contents are **Texts**; objects can refer to objects]

A closed map is a map of type $D \rightarrow \text{Text}(D)$ from some finite index set D to **Object** contents. Reference between objects consists of the occurrence of the referent's index in the referring object's content **Text**. Object indices are treated as **Abstract Identifiers**. The **FDL** 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 **FDL**. See Abstract Ids & Closed Maps (section 4.4.1) and Closed Map Operations (section 4.4.2).

Current Closed Map

[the main part of the state in a **Session** with a **Client** of the **FDL**, being a distinguished, variable **Closed Map**]

See Current Closed Maps (section 4.5.2) for explanations of the operations that can be performed to update the current closed map. A **Client** can preserve the current closed map for later access, modulo uniform change of object identifiers (see Abstract Ids & Closed Maps (section 4.4.1)).

Definition

[an explicit eliminable definition of a mathematical operator or concept]

In the sense of a definition of a mathematical operator or concept, we mean an *eliminable* definition 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.

External Name

[a concrete name whose association to **Abstract Identifiers** is maintained by the **FDL**; "external" emphasizes the contrast with abstract identifiers which are "internal" to an FDL]

While the principle name space used by the **FDL** consists of **Abstract Identifiers**, a relatively small number of identifiers must be maintained that make sense outside the FDL process in order to identify FDL entities outside the FDL process. When connected to the FDL, **Clients** can refer to abstract ids and communicate them via the FDL to other connected clients. But in order for clients to communicate identifiers outside the FDL, one must have a concrete substitute that can be later resolved when connected to the FDL. For example, one person may wish to tell another person via e-mail how to find an object in the FDL (from which many other objects can be found); to do so the first person simply needs to have the FDL 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. External names are bound to **Objects** or **Closed Maps** via **Name Server** objects.

FDL

[Formal Digital Library; a repository of a certain kind with a process for using it]

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 FDL as a process (section 4.7) for maintaining, accessing, and modifying the repository.

Actually we usually mean here an architecture suitable as a basis for building such libraries, and use the term (the “L” part anyway) with some reservations, but it continually indicates one of our main intentions.

One should expect that multiple independent FDLs (section 4.3.2) 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 an FDL are specifications for how to verify formal content. It is not part of the FDL design to govern what counts as legitimate validation, and indeed FDLs are intended to be radically open. This requires accounting for what has been verified and by what means.

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

See Impartiality (section 4.1), Formal vs Informal (section 4.2), Logical Libraries (section 4.3.1)

Formal

[having 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.

As is explained in Formal vs Informal (section 4.2), our special interest regarding **FDL** design is “formally grounded” content, i.e. both formal material such as **proofs** and programs as well as informal material that relates formal material to wider informal practice and understanding.

Inference Engine

[a process that can verify (or generate) an **Inference Step**]

The **FDL** process builds and applies inference engines according to instructions stored in the **FDL** 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 **Refiner** is an inference engine that generates premises from a conclusion **Proposition** and a **Justification**.

Inference Step

[expression of an inference from zero or more premises]

By this we mean the kind of data typically intended to express a logical inference from zero or more premises to a conclusion. An **Inference Engine** is the device that is employed by the **FDL** process to generate or check an inference step.

An inference step is represented as an individual **Object** (rather than being simply a subexpression of a **Proof**) because the cost of individual inference steps can be expected to dominate the cost of checking a proof. The organization of inference steps into proofs is rather cheap, and there is benefit to making inferences checkable independently of the proofs they occur in.

An **FDL** process checks or generates an inference step by applying an **Inference Engine**, which is usually a process whose creation is according to user specified methods. The inference step comprises expressions for each **Proposition** of the conclusion or premises, as well as a **Justification** which may be used by inference engines to restrict what counts as an inference of that type. The same inference can be verified by any number of different inference engines since the fact that a given inference has been verified by a given engine is expressed as the content of a **Certificate** which is external to the inference and refers to it.

Justification

[data provided to an **Inference Engine** in an **Inference Step** in addition to its **Propositions**]

It would be a somewhat simple or else exceedingly slow **Inference Engine** that simply used the conclusion and premises of a prospective **Inference Step** to ascertain whether it was acceptable. Realistically, there is some further specification that narrows the class of inferences that must be considered.

For example, if there is some pattern matching involved, especially second-order pattern matching, one might achieve significant savings by specifying the substitution explicitly or at least providing some hints. A more interesting case would be the use of a **Refiner** which actually generates the premises from the conclusion and the justification.

The **FDL** design does not say what the content of the justification part of an inference step must be; that is a matter specific to the introduction of an **Inference Engine**. The difference in what would constitute a natural form of justification is one major indicator of difference in the natures between inference engines. Closely related engines may share the same forms of justification.

Name Server

[an association of "names" with **Closed Maps** or **Objects**]

A means for referring to **Closed Maps** and **Objects** stored in the **FDL** which one does not already have in hand is essential.

This mechanism can also be used to provide a basic access control; if one **Client** is the owner of the name server, then that client specifies a partial function that takes a client/name pair as argument, and if it returns a value at all it returns a **Closed Map**, an **Object** or another name server as result (enabling a name server chain). The name server is applied by the FDL in service to a client which is provided as the client argument mentioned above.

The "name" supplied to the name server can be any **Text**, including an **External Name** such as a string. This is how references to **FDL** contents may be communicated outside the **FDL**.

The partial function associated with a name server by the **FDL** can be changed by the owner of the name server object, and so names via name servers do not provide the same referential fixity as object references themselves.

Native Language

[a programming notation executable by the **FDL** process]

Clients of the **FDL** must be able to stipulate programs executed by the FDL process. Request for execution of such programs and returning their results is a basic interaction between clients and the **FDL**. Most work of certifying inference steps is expected to be done outside the **FDL** by inference engines, the **FDL** process simply invoking those engines and recording the results. A native language should provide generic computational methods (the glue) as well as some basic **FDL**-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 (such as inference engines).

The execution of native language programs, which are **Texts**, is implemented as part the **FDL**, 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 **FDL**.

A generic native language macro facility is provided as well. These macros are expanded by simple match against a left-hand-side **Text** whose immediate subtexts, and perhaps some constituent **Pro-textual Constituents**, are then substituted into a right-hand-side.

Object

[the unit of **FDL** content; abstractly named **Texts**]

The notion is technically dependent on that of **Closed Map**, which is a map from some finite index set to object contents. 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

Text and the object is identified by an index in the closed map.

Pro-textual Constituent Values

[the constituents of **Texts** that are not themselves texts]

Examples would be integers, strings and **Abstract Identifiers**. See Pro-textual Constituents (section 4.4.5) for more explanation.

Proof

[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 **Proposition** 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 **FDL**, and so one must be careful when certifying a proof that one limits ones inferences by stipulation to acceptable and compatible methods, for example by employing **Sentinel** expressions.

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 propositions at each inference are derived partly from the whole block-form proof as a context.

See Proof Organization (section 4.5.11.1).

Proposition

[an expression used as a conclusion or premise of an **Inference Step**]

We employ the term "proposition" because it has fewer misleading connotations in this domain than "sentence," "assertion," "statement" or "proof goal."

It is not part of the **FDL** design what the structure of an proposition must be, except that it must be an expression of the general form used in the **FDL**, ie, a **Text**.

A proposition 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 a proposition *might* depend on an inference in which it occurs. For example, there may be some sort of schematic variables that are shared by the premises and conclusion of an inference step indicating a generality such as that any way of *uniformly* instantiating the schematic variables throughout the inference is also a legitimate inference. A "proposition" with such schematic variables might not then really mean anything independently.

In a simple Hilbert-style system the propositions might be the kinds of formulas that could be further combined by standard sentential operators such as disjunction. In a block structured natural deduction system a proposition might be the sort of formula just mentioned combined with a context for assumptions. In a sequent calculus the propositions would be whole sequents. The FDL conception of propositional form, however, is purely programmatic, and consists simply of its role in **inferences**; the aforementioned notions of proposition would be instances.

When a new kind of **Inference Engine** is introduced into the **FDL** one must design the form of propositions 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 proposition 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 propositions 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.

Refiner

[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.

Sentinel

[an expression occurring in a **Certificate** validating an **Inference Step** identifying which primitive logical resources are permitted in justifying it; the unit of coherency in **Proofs**]

The connotation of "sentinel" is that it guards against the intrusion of untrusted entities into an inference, which is essential in a collection that comprises multiple incompatible logics. As explained in Proof Sentinels (section 4.5.11.3), the sentinel expression constitutes the criterion of coherency among inferences organized into a **Proof**.

Session

[a subprocess of the **FDL**, with associated state, for communication with a **Client**]

The main parts of a session are the "client-slot," the **Current Closed Map** of the session, the method of communication with the **Client** via the session, and the parts of state supporting that communication.

The "client-slot" indicates which **Client** is the owner of the session, which will be supplied as an implicit parameter in the execution of **Native Language** programs by the **FDL** on behalf of the **Client**; various privileges of access and control are indexed by **Client** identity.

Tactic

[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 be that of **Proof** as used in the **FDL**. The principal use of the tactic prover by the **FDL** 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 **FDL Proof**, an alternative use of a tactic prover by the **FDL** would be to call the tactic prover's bluff and demand that the tactic prover produce for the **FDL** the smaller inferences that it claims existed. A tactic prover providing this alternative access by the **FDL** 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**.

Term

[synonym for **Text**; the main form used in the **FDL** for structured data such as expressions]

The nomenclature "Term" has some currency among those concerned with logical syntax for expressions.

Text

[the main form used in the **FDL** for structured data such as expressions]

Texts used as **FDL** content are simple recursive structures, ie, they are abstract structures rather than character 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. See Formal vs Informal (section 4.2) and Words vs Formality (section 4.2.1).

For purposes of accounting the nature of texts is important only insofar as there is a determinate notion of occurrences within texts of **Abstract Identifiers**. For purposes of typical syntactic analysis used in formal languages, however, the subexpression is dominant and so we have adopted a syntactic form for texts in which that relation is dominant and explicit.

The text structure is iterated operators on subtexts. In addition to its subtexts, a text contains a sequence of labelled values presupposed by the construction of texts, which we call **Pro-textual Constituents**; character strings and **Abstract Identifiers** are among these pro-textual values. An individual text consists of a sequence of zero or more pro-textual constituents together with a sequence of zero or more immediate subtexts; further, any such pair of sequences constitutes a text. The sequence of pro-textual constituents may be construed as identifying the "operator" of which the text is an instance, or sometimes this sequence together with the number of subtexts is construed as the operator.

Binding structure, however, ie, which expressions are variables and which become bound in which texts, is *not* considered part of the text structure as far as the FDL is concerned. Binding structure is attributed to texts by the **Clients**. 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 FDL clients, that they will be the subject of further innovations, and that disputes should be waged among clients and not between FDL designers and clients.

Text Server

[the aspect of an **FDL** as a process exchanging **Texts** with **Clients**]

An API to an **FDL** used by **Clients** to access or contribute **Texts** must provide a means of realizing **Abstract Identifiers** as concrete values for the duration of a **Session**. The fact that across different sessions, the representation of identifiers internal to the **FDL** is likely to change may be seen as the enforcement of abstractness.

Of course, the method for representing the structure of **Texts** for communication with the **FDL** is another aspect of such an API. An XML presentation of **Texts** would be a reasonable candidate.

Acknowledgements

We want to thank Juanita Heyerman for helping produce this manuscript during her visit to Ithaca and under a tight time pressure. Her knowledge of the tools we use and the people on the project are invaluable.

We also want to thank Alexei Kopylov for contributing a section; Lori Lorigo for providing feedback; and Aleksey Nogin for insightful comments on the manuscript and contributions to a section as well.

Cornell University has been very supportive of this project, providing additional funding to help create this document. The Cornell University Library has also shown strong support by offering to indefinitely host our collection.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer Verlag, 1996.

[2] Uri Abraham. *Models for Concurrency*, volume 11 of *Algebra, Logic and Applications Series*. Gordon and Breach, 1999.

[3] ACL2 home page. `http://www.cs.utexas.edu/users/moore/acl2`.

[4] D. A. Agarwal, O. Chevassut, M. R. Thompson, and G. Tsudik. An integrated solution for secure group communication in wide-area networks. In *Proceedings of the 6th IEEE Symposium on Computers and Communications, Hammamet, Tunisia, July 3–5*, pages 22–28, 2001.

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

[6] Stuart Allen and Robert L. Constable. Enabling large scale coherency among mathematical texts in the NSDL. Technical report, Cornell University, Computer Science Department, 2003.

[7] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

[8] Stuart F. Allen. Abstract identifiers and textual reference. Technical Report TR2002-1885, Cornell University, Ithaca, New York, 2002.

[9] R. Alur and D. L. Dill. Automata-theoretic verification of real-time systems. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for RealTime Computing*, pages 55–82. Wiley, 1996.

[10] R. Alur, L. Fix, and T. A. Henzinger. Event-clock automata: a determinizable class of timed automata. In *Proc. 6th Int. Conf. Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 11–13. Springer-Verlag, 1994.

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

[12] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[13] Myla Archer and Constance Heitmeyer. Mechanical verification of timed automata: A case study. Technical report, Naval Research Laboratory, Washington, DC 20375, May 19, 1997. A shorter version of this report was presented at RTAS '96, Boston, MA, June 10–13, 1996.

[14] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.

[15] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata. In *User Interfaces for Theorem Provers*, Eindhoven, The Netherlands, July 1998. Informal proceedings available at `http://www.win.tue.nl/cs/ipa/uitp/proceedings.html`.

[16] Brian Aydemir, Adam Granicz, and Jason Hickey. Formal design environments. In Carreño et al. [40], pages 12–22.

[17] Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in Nuprl. In Carreño et al. [40], pages 23–32.

[18] Eli Barzilay, Stuart Allen, and Robert Constable. Practical reflection in Nuprl. In Phokion Kolaitis, editor, *18th Annual IEEE Symposium on Logic in Computer Science, June 22–25, Ottawa, Canada*, 2003.

[19] David A. Basin and Robert L. Constable. Metalogical frameworks. In G. Huet and G. Plotkin, editors, *Logical Environments*, chapter 1, pages 1–29. Cambridge University Press, Great Britain, 1993.

[20] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[21] Giampaolo Bella, Cristiano Longo, and L. C. Paulson. Verifying second-level security protocols. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13^{th} International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 352–366. Springer-Verlag, 2000.

[22] T. Bemers-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 279(5):34–43, May 2001.

[23] 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.

[24] Ralph Benzinger. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming*, 11(1):3–31, 2001.

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

[26] Gustavo Betarte and Alvaro Tasistro. Extension of Martin Löf's type theory with record types and subtyping. In Giovanni Sambin and Jan Smith, editors, *Twenty-Five Years of Constructive Type Theory*, chapter 2, pages 21–39. Oxford Science Publications, 1998.

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

[28] Mark Bickford. Experiments with theory modification in the FDL. *In Progress*, 2003.

[29] Mark Bickford and Robert L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.

[30] Mark Bickford and Alexei Kopylov. Verification of protocols by combining provers using the FDL. *In Progress*, September 2003.

[31] Nina Bohr and Robert L. Constable. Defining inductive and co-inductive types using union and intersection types. Draft, August 2003.

[32] Frederick P. Brooks and Ivan E. Sutherland. *Evolving the High Performance Computing and Communications Initiative to Support the Nation's Information Infrastructure.* National Academy Press, 1995. 136 pages.

[33] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[34] 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.

[35] Ricky Butler, James Caldwell, and Ben Di Vito. Design strategy for a formally verified reliable computing platform. In *Sixth Annual Conference on Computer Assurance*, COMPASS91, Gaithersburg, MD, June 1991.

[36] James Caldwell. Moving proofs-as-programs into practice. In *12th IEEE International Conference Automated Software Engineering.* IEEE Computer Society, 1997.

[37] James Caldwell. Classical propositional decidability via Nuprl proof extraction. In Jim Grundy and Malcolm Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 105–122, Canberra, Australia, September-October 1998. Springer.

[38] James L. Caldwell and John Cowles. Representing Nuprl proof objects in ACL2: toward a proof checker for Nuprl. In D. Borrione, M. Kaufmann, and J. Moore, editors, *Proceedings of Third International Workshop on the ACL2 Theorem Prover and its Applications.* TIMA Laboratory, 2002.

[39] James L. Caldwell, Ian P. Gent, and Judith Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1–2), 2000.

[40] Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors. *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*. National Aeronautics and Space Administration, 2002.

[41] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:55–68, 1940.

[42] 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.

[43] Manuel Clavel, Francisco Duran, Steven Eker, Jose Meseguer, and Mark-Oliver Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcook, and J. Davies, editors, *FM'99, The World Congress On Formal Methods In The Development Of Computing Systems*, number 1709 in Lecture Notes in Computer Science, pages 1684–1703. Springer Verlag, 1999.

[44] 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.

[45] Robert L. Constable. Information-intensive proof technology; lecture notes for the marktoberdorf nato summer school. Cornell University, Ithaca, NY, 2003. `http://www.nuprl.org/documents/Constable/marktoberdorf03.html`.

[46] Robert L. Constable and Karl Crary. Computational complexity and induction for partial computable functions in type theory. In W. Sieg, R. Sommer, and C. Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in Honor of Solomon Feferman*, Lecture Notes in Logic, pages 166–183. Association for Symbolic Logic, 2001.

[47] Robert L. Constable and J. E. Donahue. A hierarchical approach to formal semantics with application to the definition of PL/CS. *ACM Trans. on Programming Languages and Systems*, 1(1):98–114, July 1979.

[48] Robert L. Constable et al. *Implementing Mathematics with the* Nuprl *Proof Development System.* Prentice-Hall, NJ, 1986.

[49] Robert L. Constable and Jason Hickey. Nuprl's Class Theory and its Applications. In Friedrich L. Bauer and Ralf Steinbrueggen, editors, *Foundations of Secure Computation*, NATO ASI Series, Series F: Computer & System Sciences, pages 91–116. IOS Press, 2000.

[50] Robert L. Constable, S. Johnson, and C. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*, volume 135 of *Lecture Notes in Computer Science.* Springer-Verlag, NY, 1982.

[51] Robert L. Constable and S. D. Johnson. A PL/CV précis. In *Principles of Programming Languages*, pages 7–20. ACM, NY, 1979.

[52] Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[53] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, Berlin, 1990.

[54] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *Conference on Programming Language Design and Implementation*, 1999.

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

[56] David Dill. The Murphi verification system. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer Verlag, 1996.

[57] Ekaterina Dolginova and Nancy Lynch. Safety verification for automated platoon maneuvers: A case study. In *International Workshop on Hybrid and Real-Time System*, Grenoble, France, June 1997.

[58] 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.

[59] H. Egli and Robert L. Constable. Computability concepts for programming language semantics. *Theoretical Computer Science*, 2:133–145, 1976.

[60] Ulfar Erlingsson and Fred Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[61] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. Massachusetts Institute of Technology, 1995.

[62] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. In *Proceedings*. PODC, 1997.

[63] Amy Felty and Douglas Howe. Hybrid Interactive Theorem Proving using Nuprl and HOL. In W. McCune, editor, *14th Conference on Automated Deduction*, number 1249 in Lecture Notes in Artificial Intelligence, pages 351–365. Springer Verlag, 1997.

[64] 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.

[65] Paul Ginsparg. Creating a global knowledge network. In *Second Joint ICSU Press-UNESCO Expert Conference on Electronic Publishing in Science*, 2001.

[66] J-Y. Girard. The system F of variable types: Fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[67] J-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[68] K. J. Goldman. *Distributed Algorithm Simulation using Input/Output Automata*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, July 1990.

[69] Michael Gordon and Tom Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, Cambridge, 1993.

[70] Adam Granicz and Jason Hickey. Phobos: A front-end approach to extensible compilers. In *36th Hawaii International Conference on System Sciences*. IEEE, 2002.

[71] J. Guard, F. Oglesby, J. Bennett, and L. Settle. Semi-automated mathematics. *Journal of the ACM*, 16:49–62, 1969.

[72] Ramanathan V. Guha and Douglas B. Lenat. Enabling agents to work together. *Communications of the ACM*, 37(7):126–142, 1994.

[73] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[74] Joshua D. Guttman and Mitchell Wand, editors. *VLisp, A Verified Implementation of Scheme*, volume 8, Nos. 1 & 2. Kluwer Academic Publishers, Norwell, Massachusetts, March 1995.

[75] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe (FME '96), Oxford, UK*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681. Springer-Verlag, 1996.

[76] HELM: An hypertextual electronic library of mathematics. Home page `http://helm.cs.unibo.it`.

[77] 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.

[78] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

[79] Jason Hickey, Aleksey Nogin, Adam Granicz, and Brian Aydemir. Formal compiler implementation in a logical framework. In *MERλIN, Second ACM SIGPLAN Workshop on MEchanized Reasoning about Languages with varIable biNding*, 2003.

[80] Jason Hickey, Justin D. Smith, Brian Aydemir, Nathaniel Gray, Adam Granicz, and Cristian Tapus. Process migration and transactions using a novel intermediate language. Technical Report caltechCSTR:2002.007, California Institute of Technology, Computer Science, August 2002.

[81] Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[82] Jason J. Hickey, Brian Aydemir, Yegor Bryukhov, Alexei Kopylov, Aleksey Nogin, and Xin Yu. A listing of MetaPRL theories. `http://metaprl.org/theories.pdf`.

[83] Jason J. Hickey et al. Mojave research project home page. `http://mojave.caltech.edu/`.

[84] Jason J. Hickey, Nancy Lynch, and Robbert Van Renesse. Specifications and proofs for Ensemble layers. In W. Rance Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 1999.

[85] Jason J. Hickey, Aleksey Nogin, Alexei Kopylov, et al. MetaPRL home page. `http://metaprl.org/`.

[86] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, 1995.

[87] Amanda Holland-Minkley, Regina Barzilay, and Robert L. Constable. Verbalization of high-level formal proofs. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 277–284. AAAI, July 1999.

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

[89] Douglas J. Howe. The computational behaviour of Girard's paradox. In D. Gries, editor, *Proceedings of the 2^{nd} IEEE Symposium on Logic in Computer Science*, pages 205–214. IEEE Computer Society Press, June 1987.

[90] Douglas J. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125, of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, Berlin, 1996.

[91] Douglas J. Howe. Semantic foundations for embedding HOL in Nuprl. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology*, volume 1101 of *Lecture Notes in Computer Science*, pages 85–101. Springer-Verlag, Berlin, 1996.

[92] W.A. Hunt, Jr. *FM8501: A Verified Microprocessor*. PhD thesis, University of Texas at Austin, 1985.

[93] Steven C. Johnson. Yacc — yet another compiler compiler. Computer Science Technical Report 32, AT&T Bell Laboratories, July 1975.

[94] Matt Kaufmann and J. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[95] Matt Kaufmann and J Strother Moore. ACL2 home page. `http://www.cs.utexas.edu/users/moore/acl2/`.

[96] J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[97] J. Kleinberg. Navigation in a small world. *Nature*, 406:845, 2000.

[98] J. Kleinberg. Small-world phenomena and the dynamics of information. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 14; Proceedings of the 2001 Neural Information Processing Systems (NIPS) Conference*, Cambridge, MA, 2002. MIT Press.

[99] Alexei Kopylov. Verified implementation of red-black trees. FDL Algorithms Collection. `http://www.nuprl.org/Algorithms`.

[100] Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science*, pages 86–95, 2003. To appear.

[101] 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.

[102] 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.

[103] 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.

[104] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In A. Arora, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems, Austin, TX*, pages 33–40. IEEE Computer Society Press, 1999.

[105] Butler W. Lampson, Nancy A. Lynch, and Jørgen F. Søgaard-Andersen. Correctness of at-most-once message delivery protocols. In

Richard L. Tenney, Paul D. Amer, and M. Ümit Uvar, editors, *Formal Description Techniques, VI: Proceedings of the IFIP TC6/WG6.1 6th International Conference on Formal Description Techniques*, pages 385–400, Boston, MA, October 1993. FORTE'93, North Holland, 1994.

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

[107] Douglas B. Lenat. CYC: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

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

[109] Chuck C. Liang. Compiler construction in higher order logic programming. In *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 47–63, 2002.

[110] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 34 of *Operating Systems Review*, pages 80–92, 1999.

[111] Xiaoming Liu, Christoph Kreitz, Robbert van Renesse, Jason J. Hickey, Mark Hayden, Kenneth Birman, and Robert Constable. Building reliable, high-performance communication systems from components. In *17th ACM Symposium on Operating Systems Principles (SOSP'99)*, volume 33(5) of *Operating Systems Review*, pages 80–92. ACM Press, December 1999.

[112] Xiaoming Liu, Robbert van Renesse, Mark Bickford, Christoph Kreitz, and Robert Constable. Protocol switching: Exploiting meta-properties. In Luis Rodrigues and Michel Raynal, editors, *International Workshop on Applied Reliable Group Communication (WARGC 2001)*, pages 37–42. IEEE, 2001.

[113] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[114] Nancy Lynch and Mark Tuttle. An introduction to Input/Output automata. *Centrum voor Wiskunde en Informatica*, 2(3):219–246, September 1989.

[115] Donald MacKenzie. *Mechanizing Proof.* MIT Press, Cambridge, 2001.

[116] Conal L. Mannion and Stuart F. Allen. A notation for computer aided mathematics. Department of Computer Science TR94-1465, Cornell University, Ithaca, NY, November 1994.

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

[118] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[119] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

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

[121] The MathBus Term Structure. `www.nuprl.org/mathbus/mathbusTOC.htm`.

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

[123] William McCune. Solution of the Robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.

[124] William McCune. Well-behaved search and the Robbins problem. In H. Comon, editor, *8th International Conference on Rewriting Techniques and Applications (RTA), Sitges, Spain*, number 1232 in LNCS, pages 1–7. Springer-Verlag, 1997.

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

[126] Jose Meseguer and Mark-Oliver Stehr. The HOL-Nuprl connection from the viewpoint of general logics, 1999.

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

[128] Andrew M. Mironov and Virendra C. Bhavsar. A new approach for specification and verification of distributed agents. Technical Report TR98-12a, University of New Brunswick, June 2000.

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

[130] E. Moggi. Computational lambda calculus and monads. Technical Report ECS-LFCS-88-86, Univ. of Edinburgh, Edinburgh, UK, 1988.

[131] J.S. Moore. A mechanically verified language implementation. *J. of Automated Reasoning*, 5(4):461–492, 1989.

[132] J.S. Moore. Piton: A verified assembly-level language. Technical Report CLI-22, Computational Logic, Inc., Austin, TX, June, 1988.

[133] Evan Moran. *Adding Intersection and Union Types to Howe's Model + of Type Theory [working title]*. PhD thesis, Cornell University, 2003.

[134] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *Principles of Programming Languages*, 1998.

[135] Chetan Murthy. An evaluation semantics for classical proofs. In *Proceedings of Sixth Symposium on Logic in Comp. Sci.*, pages 96–109. IEEE, Amsterdam, The Netherlands, 1991.

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

[137] Pavel Naumov. Importing Isabelle formal mathematics into Nuprl. Technical Report TR99-1734, Cornell University. Department of Computer Science, 1999.

[138] George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.

[139] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

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

[141] Aleksey Nogin and Jason Hickey. Sequent schema for derived rules. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002)*, volume 2410 of *LNCS*, pages 281–297. Springer-Verlag, 2002.

[142] Nuprl home page. `http://www.nuprl.org`.

[143] Nuprl web libraries. `http://www.nuprl.org/Nuprl4.2/Libraries/Welcome.html`.

[144] Office of Naval Research. Critical Infrastructure Protection and High Confidence, Adaptable Software (CIP/SW) Research Program of the University Research Initiative (URI) – Topic #8 Digital Libraries for Constructive Mathematical Knowledge. As published in the Commerce Business Daily; see also `http://www.onr.navy.mil/02/baa/expired/00_015.htm`, June 19 2000. Solicitation Number: BAA 00-015.

[145] Chris Okasaki. Red-black trees un a functional setting. *Journal of Functional Programming*, 9(4):471–477, May 1999.

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

[147] Jens Otten and Christoph Kreitz. T-string-unification: Unifying prefixes in non-classical proof methods. In U. Moscato, editor, *5th 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.

[148] 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*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer Verlag, 1996.

[149] L. C. Paulson. Mechanizing a theory of program composition for UNITY. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(5):626–656, 2001.

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

[151] Ivars Peterson. *Fatal Defect: Chasing Killer Computer Bugs.* Vintage Books, 1996.

[152] Sabina Petride. Knowledge-based specifications in the logic of events. Draft paper and PRL seminar notes, 2003.

[153] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23(7) of *SIGPLAN Notices*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[154] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2), 1994.

[155] 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/`.

[156] Andrew M. Pitts and Murdoch Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[157] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–59, 1975.

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

[159] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods: PROCOMET'98*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

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

[161] RTI. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Planning Report 02-3. National Institute of Standards and Technology, Research Triangle Park, NC, May 2002.

[162] Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems*, 19(5), September 1997.

[163] 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.

[164] 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.

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

[166] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

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

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

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

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

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

[172] David Tarditi. *Design and implementation of code optimizations for a type-directed compiler for Standard ML.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.

[173] Jeffrey D. Ullman. *Elements of ML Programming.* Prentice Hall, 1998.

[174] Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, July 2002.

[175] R. Vitenberg, I. Keidar, G. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report CS99-31, Comp. Sci. Inst., The Hebrew University of Jerusalem, September 1999. Also MIT Technical Report MIT-LCS-TR-790.

[176] Tjark Weber and James Caldwell. Constructively characterizing fold and unfold. In *13th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2003), held August 25-27 in Uppsala, Sweden*, 2003.

[177] Pierre Weis and Xavier Leroy. *Le langage Caml.* Dunod, Paris, 2nd edition, 1999. In French.

[178] J.L. Welch, L. Lamport, and N. Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. Laboratory for Computer Science MIT/LCS/TM-361, Massachusetts Institute of Technology, Cambridge, MA, June 1988.

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

[180] 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, *10th Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 485–499. Springer Verlag, 1990.

[181] Xin Yu, Aleksey Nogin, Alexei Kopylov, and Jason Hickey. Formalizing abstract algebra in type theory with dependent records. Accepted to TPHOLs 2003 "Track B", 2003. `http://mojave.cs.caltech.edu/nogin/papers/formalaa.html`.