

A Innovative Claims

This is a proposal to increase the capabilities of Logical Programming Environments (LPE's). These systems provide formal support to the programming and software evolution process, including advanced type checking and formal verification of code. They comprise a programming language, a specification language, a programming logic and a theorem prover.

The core innovation is to link three theorem provers, HOL, Nuprl and PVS to an LPE. The first new capability enabled is sharing libraries of basic mathematical facts. This is a technically challenging task, but accomplishing it advances many of the goals articulated in DARPA's BAA 98-10, indeed it appears to be essential to achieving some of these goals. Moreover accomplishing it will enlarge the formal design space available for managing complex software.

The second new capability is that combinations of tools, provers, decision procedure and model checkers can cooperate on system verifications. Moreover, a new concept of a multi-logic proof can be explored, and we propose new ways of guaranteeing the soundness of these extended proofs. In particular we propose to provide logic-based accounting mechanisms to inform users of formal tools exactly what assumptions they depend on and what level of verification has been achieved for them. Additionally we will provide a means to harden the decision procedures and model checkers that are now often operated without high assurance, as ordinary pieces of software.

The Logical Programming Environment will be based on type theory which has proven itself so well over the past two decades as a unifying semantic framework, providing concepts that have been successful in modeling actual software systems as well as in providing the semantics for a wide variety of programming logics. Moreover, implementations of type theories have supported large constellations of formal tools proven to be useful in analyzing and reasoning about such software. The type theory provides a natural basis for tools to interoperate.

By building on Logical Programming Environments, we will improve the methods to link running code to its specifications and to link formal models of what the code is doing. This involves opening the tools used to correctly transform programs and also improving the link between code and specifications that our group at Cornell pioneered, namely the extraction of programs from proofs that programming tasks are tractably solvable. Finally we want to explore some very new ideas for treating formal theories as software systems, a generalization of the notion that proofs are like programs.

The results we produce will enable DARPA to eventually concentrate an unprecedented amount of formal power on the task of building more reliable and secure software and to demonstrate new capabilities for controlling complex software. We will demonstrate this on verifications we are now doing under DARPA contract on distributed computing middleware.

B Technical Rationale

B.1 Background and problems to be solved

B.1.1 Background

Formal methods are needed now more than ever because military infrastructure depends on complex software working correctly, and the nation's security also depends on secure computing infrastructure. There are no better alternatives for providing high levels of confidence. Demands placed on software require that formal methods capabilities be substantially increased. This requires research, education and training, and deployment of more integrated tools.

Technology based on formal methods has now reached into standard computing and design practice (in the normal time frame of 17 to 20 years since their appearance in research labs). Type inference algorithms are used in programming languages along with decision procedures to detect array bounds errors; model checkers are components in industrial CAD tools, and specification languages are part of programming environments. In all cases these tools have brought greater assurance that hardware and software will not fail. It is now widely recognized that formal methods define the highest possible levels of trust, assurance, accountability and security. But much more can be done — both to further develop the best ideas discovered in the 80's and 90's and to investigate new ideas of great promise.

B.1.2 Problems

The technical problems that DARPA and its research community now face are these:

1. **Verified code** We need sound links between specifications, verified algorithms and actual running code. A key goal is to run verified code and formally connect models (algebraic, automata, etc.) to the code.
2. **Sound tools** We need to ground a variety of light-weight tools in sound mathematical foundations. Another step that should be investigated soon is a way to guarantee the reliability of decision procedures and enable sharing them as well.
3. **Shared tools and models** The labor intensive nature of the work requires that research groups cooperate as much as practical. Since there are a small number of serious efforts in the US, this is possible. The enabling capability is to share libraries of basic mathematical facts. This will also provide sharing of models.
4. **Scalable environments** The methodology and tools must be scaled to deal with larger and more complex software systems — verification in the large.
5. **Integrated tools** To make the technology useful and acceptable it must be integrated with other tools and with standard programming environments and problem solving environments.

In the future DARPA will face new questions. As formal tools penetrate into software practice and people come to rely on them, the tools will become the objects of *intense scrutiny*. It must be possible to defend them as ideal instances of reliable, very high assurance software. Our work addresses this issue as well.

B.1.3 Cornell and Bell Labs work

The verification work at Cornell and Bell Labs is grounded in significant practice. Moreover the tools we use have been constantly evolving toward more open architectures. The tools we now use for Ensemble are formally based. We have interoperable representations between type theory and ML.

Our research over the years has been characterized by improving our tools, especially to more open systems, more flexible logical frameworks and richer type theories. We know how to attack the technical problems based on our 15 years experience in the area. We have assembled a research team that can make significant progress.

B.1.4 Outline of proposal

Section B2 outlines our approach to the above 5 problem areas. Section B3 looks at the design aspects of our work and B4 talks about the implementation issues. Finally section B5 discusses the new capabilities that will result from this work.

B.2 Proposed solutions

Our approach to the problems enumerated above is based on years of experience with the cycle of designing and building a verification system based on the latest advances in theory, experimenting with it on real problems, distilling insights and results into new theory and new designs and repeating the process.

This process has convinced us that a programming environment based on a real programming language at its core is the way to tie verifications to running code. The latest success from this approach is our ability to verify the stack compression technique used in the Ensemble group communications system [].

Another significant insight that drives this proposal is that we know how to combine the verification systems HOL and Nuprl to the extent that they can share a common data base of facts. We judge this to be a significant advance illustrated by the preliminary results using the system [?]. Our experience with this prototype convinces us that we can at least add facts proved by PVS. But much more seems possible as we outline in B3 and B4.

The additional capabilities derive from our work with the Nuprl library mechanism. We know how to keep track of logical dependencies between theories using proof sentinels. and thus how to create proofs whose inference steps can be done by different systems. Moreover, we have designed and tested a general module mechanism for relating theories that is the basis for extending the HOL/Nuprl prototype. The module system will also help us attack the problem of creating verified light-weight tools as we describe in below.

B.2.1 Verified code problem — linking code to models and specifications.

The results in this section address the problems discussed in BAA 98-10 under the heading of Software Model Creation as well as under the Frameworks for Interoperability heading.

In a Logical Programming Environment, we can reason directly about running code. Nuprl uses a dialect of ML as its programming language, and it has another mechanism for linking

specifications to code that the Cornell group pioneered, namely, programs can be synthesized from constructive proofs that a programming task is solvable. The rules for the logic guarantee that the extracted code meets the specifications. This is an example of the kind of constructive method that derives software from specifications which is mentioned in the BAA. In section B5 we propose a way to enhance this capability for a small cost.

B.2.2 Sound tools problem — a semantic basis for modeling interface.

When we speak about a verification system we mean software that can check arguments for correctness against specific sets of axioms and inference rules. The system can also produce justification showing what fact depends on what others. A basic criterion for such a system is that there is a simple program (about 300 lines) that can check the *primitive proofs*, i.e. those proofs that invoke only the axioms and inference rules but no decision procedures. This simple program can be written in many languages and proved correct by hand. (Sometimes this is part of the definition of a foundational system.) This defines the highest standard of rigor. There is the opportunity for foundational provers to verify the code that implements the basic checking for other such provers.

There are other logical tools such as decision procedures and model checkers and type checkers that are not foundational. They are pieces of code that perform logical operations, but they might be difficult to understand and verify. It is important to know that these methods “conform to a logical theory.”

As an illustration, consider the well-known SupInf algorithm that has been widely used in theorem provers or the Arith procedure used in Nuprl. These procedures are used in theorem provers. Decision procedures are complex and a potential source of unsoundness. When they are used, the prover is not foundational unless there is a reduction. Surprisingly there are no formal proofs of correctness for these procedures and worse, all of the published informal proofs are either incorrect or grossly incomplete — critical cases are missing.

There are two basic ways to use decision procedures soundly. We say that a procedure is run in *safe mode* if it generates primitive proofs (those that do not invoke decision procedures). It is run in *unsafe mode* if it just decides whether or not there is a proof, but does not exhibit one. One method of using these procedures is (a) to first run them in unsafe mode, which is fast, and later, off-line run them in safe mode. HOL does this. Another approach, (b) is to formally verify the procedure (without using it in the verification) and possibly bind it by reflection [1].

We propose to explore capability (b) more thoroughly. It serves a dual purpose of being a test case of verification by techniques of the community. It might allow the establishment of a new standard. We need to know that it can be done.¹

B.2.3 Shared tools and models problem — sharing basic theories

Interactive theorem provers require large libraries of formalized mathematics to be effective. Building them is very time consuming and labor intensive, yet every theorem proving research group reproves basic facts that other groups have already done, especially numerical facts—for

¹Also useful to compiler ML technology—as Pugh theory is to C.

N, Z, Q, or the reals R, also theories of lists, trees, graphs, automata and other basic data types and models.

An obvious approach to avoid this costly duplication is to share a data base of basic facts. It turns out that doing this is difficult. A great deal of theory has been directed toward elegant solutions [?], but it is not clear what is practical. The difficulties are that mathematics is represented slightly differently in all systems, in older systems there are very idiosyncratic definitions, even of the natural numbers! Systems use different logics and different primitive notions. The provers can store state with the facts.

Despite these problems, we have shown that it is possible to share results between two major interactive provers based on very different logics but sharing the same theorem proving style, namely HOL and Nuprl can share. Not only this, PVS is the same type of prover, so sharing with it seems very plausible as we propose to show.

So while we expect that there will always be several major verification systems, some in industry, some in academia, some in Europe, etc., and while they will need to develop their own specialized theories, it is possible for them to share a large core of basic mathematics. We believe that this capability and the technology need to achieve it is essential to achieving DARPA's goals as stated in BAA 98-10.

The model we envision is shown in Figure 1. All mathematics is exchanged over a common channel, called *MathBus* ??²; the *logical library* serves as a formal knowledge base for the mathematics from the heterogenous systems; the *logical environment* provides organization of results in the library; and *logic editors* are used to view the mathematics in the system. Each of the components is part of the research we are proposing.

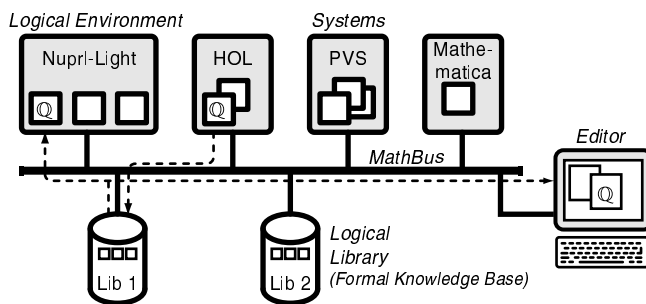


Figure 1: Logical Programming Environment

B.2.4 Scalable environments problem — verifications in the large

The software industry has proved the design value of modular component technology. Modern programs are composed of objects and modules that capture generic software features. This model is valuable for the verification technology as well, and we build our own tools using the modular mechanisms (and our designs are becoming more modular as we build new systems like Nuprl-Light).

²MathBus has been funded under a previous DARPA grant ??. The current version provides a common *syntax* for exchanging mathematics; we propose to extend it to provide a common communication protocol.

In order to apply formal tools to large software systems, verification technology needs a logical interpretation of modules. We propose a LPE where modules are viewed as logical *theories*, and modular software verification is accomplished through *reflection*. Just as modular software design provides lightweight flexible lightweight programs, the modular theory design provides lightweight formal design. We are currently applying these mechanisms to provide run-time optimization of Ensemble, a strictly modular communication system.

The technology we describe here solves some of the key problems in the task of building frameworks for interoperable representations.

B.2.5 Integration of tools problem — industrial environments

We propose to work on this problem through a subcontract to GrammaTech. They have an SBIR contract from ONR to provide a Formal Methods Environment (FME) for integrating such tools. They are especially concerned with the following problem.

User-Interface inadequacies: existing formal methods are hobbled by weak user-interfaces, and the cost of making better ones often dominates the cost of implementing the core method.

Our proposal directly attacks these problems. We have worked with GrammaTech before to build a standard user interface for Nuprl. The prototype they built for us is the basis for their plans to build one for PVS with SBIR funding. We are proposing that they expand their plans to include an interface to the LPE we are building.

Our previous work with GrammaTech also helped us test our design of an open theorem proving architecture. We wrote an editor interface that will support foreign as well as resident editors.

B.3 Design Issues

Some of the work we propose requires careful theoretical analysis and design. For example, a proposal we have made for linking HOL and Nuprl required a new analysis of the foundations of computational type theories. Howe [?] was able to show that a substantial subset of the Nuprl type theory was consistent with HOL. We can use the module mechanisms for the library to enforce this subset in the link to HOL. There is work to be done in extending Howe's analysis to other type constructors such as intersection types and recursive types. To begin a study of links to PVS, we need to analyze that type theory and related it to subsets of the computational type theories.

B.3.1 Details of relating type theories

In order to allow sharing of mathematics between Nuprl and systems like HOL and PVS [24] that use classical logic, a fundamental extension to Nuprl was required. Of course, it is trivially possible in principle to embed any one of these systems in another by, for example, formalizing the proof theory, but we were after a practical connection. For example, if some new theorems about the integers are proven in HOL, we want to automatically import them as theorems about the *Nuprl* integers. More generally, want to *interpret* HOL in Nuprl, using specific objects of interest for parts of the interpretation.

Because Nuprl has an expressive and flexible type system, it is straightforward to naturally interpret the type systems of classical type theories such as HOL and PVS. In particular, the function type is interpreted as Nuprl’s function type. The difficulty is in dealing with the classical nature of the theories, which is captured in the “epsilon” operator, which given a type A and a predicate P over A chooses some member of A satisfying P (an arbitrary value if no such value exists).

Nuprl, with the original semantics, is a logic of computable functions. All members of a function type $A \rightarrow B$ must be computable. The semantics assigns to $A \rightarrow B$ a set of untyped functional programs which have the right input-output behaviour. The behavior of programs is given operationally. Since the epsilon operator can be used to build non-computable functions, to interpret it appropriately we need augment the Nuprl semantics.

This was the primary motivation for the work in [14] and [16]. This work shows how to combine untyped functional programming languages with objects from conventional set theory such as functions-as-graphs, equivalence classes, and types as sets. Programs are tightly integrated with the set-theoretic objects. For example, function types contain both programs and set-theoretic graphs of functions. The semantic explanation is an extension of the original operational semantics. The language of programs is extended with constants from a set theoretic universe, and “evaluation” rules are added to the semantics to explain, for example, how to apply a graph of a function to an arbitrary program.

The new semantics goes a long way toward merging classical and constructive logic. If one sticks to the original Nuprl rules, programs can be still be synthesized from proofs. In addition to giving a conventional set-theoretic interpretation of mathematics formalized in Nuprl, the new semantics immediately justifies new rules, such as the law of the excluded middle, and the existence of an epsilon operator. This is enough to fully interpret the simple type theory of HOL. It should also be enough to fully interpret the more expressive type theory of PVS, although this has not yet been worked out.

In addition to integrating classical and constructive notions of functions, the new semantics also combines the respective treatments of equality. Classically, new equalities are introduced by taking equivalence classes. These are accounted for in the semantics, but by themselves they have no computational sense, so, we also have a data constructor $[\cdot]$ that packages a single representative of an equivalence class. An object $[e]$ can be thought of as being usable in place of any equivalence class containing e as a member. Giving a coherent semantic account of both equivalence classes and objects $[e]$ was the most difficult part of the work.

The extended logic should be sufficient to interpret the logics of most, if not all, existing theorem-proving systems, even the calculus of constructions, via the set-theoretic interpretation given by Werner [34].

However, there is room for substantial improvement, and this will require new theoretical work. The new semantics, as is, rules out “intersection types”. For example, the intersection, over all types A , of $A \rightarrow A$, is empty in set theory, but in a programming language context we want it to contain the polymorphic identity function $\lambda x.x$. The intersection type is useful for expressing implicit polymorphism and for representing features of object-oriented programming languages.

It would also be useful to have a recursive type constructor along the lines of Mendler [22]. This can be justified in the original semantics, but in the new semantics it is ruled out on grounds of impredicativity. Finally, it may be possible to extend the new semantics, possibly by trying to incorporate more of the old semantics, in order to gain some of the impredicative expressive power of the logics studied by Mason and Talcott (e.g. see [33]).

B.3.2 Libraries for mathematics

Once the theoretical tasks have been accomplished for translating knowledge between different formalisms, a general system for sharing mathematics requires a repository where mathematics can be archived in a common language. For this purpose, we intend to develop what might be called a “Common Logical Library,” by which we mean a library of objects with logically significant relations maintained between them. For example, a theorem about the natural numbers may depend on basic lemmas (possibly from different systems), and the logical library is required to maintain those inter-theory dependencies.

One might invent a lot of different notions of dependency of one object on another, but the design of a logical library aims at facilitating the use of such notions to support “logical” claims about collections of objects. For example, we might claim that a certain proof in the library is valid if the set of rules it depends on are valid. The logical library architecture must facilitate such arguments. This paradigm must be elaborated since there may be many ways in which a proof depends on rules and other kinds of objects.

Another aim of the logical library is that it should be able to effectively maintain and relate rule sets that may not even be compatible.

In a foundational system, the basic guarantee one needs to be able to make of a specific proof is that it can be made by composing a known, limited set of rules. A flexible accounting method is required since the rules in the library are not static—they may grow as more knowledge is introduced. Although one may well ask what rules are actually used to carry out a given proof, one often cares *not* exactly which rules were used, but whether all those rules fall within some familiar or distinguished set like ZF set theory, or one of the known theories. Also, when one calls upon a prover to prove some goal, it’s not to attractive to see after the fact whether it restricted itself to the desired set of rules, it is better to restrict it when the proof is being made.

We use what we call “Proof Sentinels” to express the stipulations limiting proofs to a given set of rules, axioms, and perhaps other objects on which these may depend. We shall embody these proof sentinels in expressions with these key features:

- There is a set of rules, etc., determined by the expression.
- Typically, the user should be able to become familiar with particular sentinel expressions. For example, brief names or descriptions are better than long lists of rules.
- If there are several sentinel expressions of interest to the user, then it must be easy to automatically discern which of those sentinels is adequate for any given proof.

Carrying out a given proof may require many resources from the library, such as definitions, tactics, and lemmas, but the sentinel is supposed to indicate a reduction of validity to those

things, such as rules, whose criteria of correctness lie outside the formal system. For example, one should be assured the proof is correct so long as the rules in the sentinel are.

The sentinel also provides the basis for accountability. One should normally be able to scrutinize the system’s justifications for various semantic or logical claims. For example, it should be possible to replay proofs, and get hold of the primitive proofs whose existence is promised by refiners. Even if not every system “claim” can be accounted for, it is key that there be a distinction made between those that are accountable and those that aren’t – one wants to know if no accountability may be expected in individual cases. For claims that are indicated as accountable, trust but verify. Independent checkers running in the background would be a paradigm of verifying accountability.

B.3.3 Modular Construction

The library provides the foundation for relating logics, and the next step is to provide a well-known, usable mechanism for lifting the formal knowledge into standard practice. The model we are proposing is based on a lightweight module system that organizes the logical library. Modules have long been used for organizing software, and they are grounded in long standing methods of organization in mathematics, such as those used in abstract algebra [?].

The modules we use represent logics and theories (possibly from different systems), and the knowledge from different systems can be related by finding a relation between their modules. In the architecture we are proposing (Figure 1), the MathBus is used as a communication mechanism, and mathematical knowledge is stored in multiple shared libraries. The MathBus provides a standardized protocol for sharing mathematics at the *syntactic* level; the library maintains logical dependencies between the knowledge, and the logical environment collects the results of the library into modules that represent generic components of logical knowledge.

The problems that arise in the module system revolve around isolating and managing theories and their components, including the following:

- developing common *presentations* of knowledge that relate results in a common language,
- isolating logics so that unverified results cannot leak between theories,
- partitioning theories, so that relations can be made on supertheories,
- providing a formal framework for incrementally expressing and developing the relations themselves,
- automatically inheriting results in a theory to all its subtheories.

The modular environment we are proposing is called Nuprl-Light. The Nuprl-Light environment is a framework for defining and relating mathematical tools and domains, but it is itself independent of any particular formal system. Nuprl-Light is light-weight in this sense: only the knowledge that is needed for the task at hand is loaded into the system. The resource requirements increase only as the problem size increases.

B.4 Implementation

We have a fast path to achieving the LPE implementation because of the prototype HOL/Nuprl system and because of on-going work in the evolution of the open Nuprl architecture. We

have built a version of the Common Logical Library that has many of the features needed to keep track of the logical dependencies between theories. We also have a prototype implementation of the NL module system. Our plan is to build these elements into a Logical Programming Environment with the features discussed. In this Section we discuss some of the implementation issues, beginning with our experience establishing an initial HOL/Nuprl connection.

B.4.1 Relating Large Libraries of Mathematics

We now have some experience using our HOL/Nuprl connection. We have imported classical theories of combinatory logic and minimal logic from HOL, and used them in Nuprl to prove a constructive normalization property [9]. More recently, we have been using a large collection imported facts about basic data structures in our verification of the SCI cache coherence protocol.

There are three clear lessons to be drawn from this experience. First and foremost, sharing formal knowledge between different verification systems can be practical. Once the underlying machinery was set up, it was far easier for us to import than to reprove the results we wanted in Nuprl. Second, the syntactic connection between HOL and Nuprl, i.e. the machinery that transports objects from HOL to Nuprl, is only part of what is needed. Also essential are mechanisms for manipulating the raw imported mathematics, in a sound way, into a form more suitable for application in Nuprl. Although the extended Nuprl semantics can interpret HOL, there are many differences between the way mathematics is represented in the two systems, and as a result, the directly imported theorems are usually of a form that makes them useless for application in Nuprl proofs. The third lesson is that massaging the theorems into the desired form is possible, and is should be largely automatable using Nuprl's theorem proving facilities.

Importation of mathematics from HOL into Nuprl is done at the theory level. An HOL theory consists of some type and individual constants, some axioms (usually definitional) constraining the constants, and a set of theorems following from the axioms (and the axioms of ancestor theories). To import a theory, one *interprets* the type constants with Nuprl types and the term constants with members of the appropriate types, and then proves the axioms. When this is done, the theorems can then all be accepted immediately as Nuprl theorems. Type-checking is undecidable in Nuprl, so the well-typedness of terms must be proven explicitly. This means that in addition to the axioms/definitions, it is also required to prove that each object associated with a type constant is a (non-empty) type, and that each object associated with a term constant has the type specified in HOL.

For many of the basic HOL theories, the choice of Nuprl terms used in the interpretation of HOL constants must be explicitly provided by the user. However, there is a large class we can compute automatically. HOL has an extensively-used package for making a limited form of ML-style recursive type definitions. The package takes as input a specification of the type that has a syntactic form close to ML's. The result of running the package in HOL is an extension to the set of constants and axioms of the current theory, and also new definitions and theorems for reasoning about the type. There is also a package for making inductive definitions of predicates and relations. It was not difficult to write an ML program within

Nuprl to automatically compute the interpretations needed for the extensions made by these packages.

To illustrate what kind of transformations are needed on directly imported mathematics, consider an example from list theory. The following is a raw import of a HOL theorem stating that a non-empty list is a cons. Because Nuprl currently has a single flat namespace, the names of all imported constants have an “h” prepended to avoid conflicts with Nuprl objects. The outermost quantifier quantifiers over the type `S` of all (small) non-empty types (this quantifier is implicit in HOL).

```

∀'a:S ↑(hall (λl:hlist('a).
           himplies (hnot (hnull l))
                   (hequal (hcons (hhd l) (htl l)) l)))

```

Apart from the outermost quantifier, the logical connectives themselves are imported constants. The transformed, “nuprl-friendly” theorem generated from the above is

```

∀'a:S. ∀l:'a List.    ¬mt(l) ⇒ hd(l)::tl(l) = l.

```

The logical connectives in HOL are all boolean-valued functions, possibly taking functional arguments, as in the case of the quantifiers. The interpretations of these connectives use boolean logic defined within Nuprl. The boolean connectives are rewritten in the second theorem to Nuprl’s normal logical connectives, which are defined using a propositions-as-types correspondence. The operator \uparrow in the imported theorem coerces a boolean into a Nuprl proposition. The imported list type is defined to be Nuprl’s list type, and the imported tail function is defined to be Nuprl’s tail function. Note however that `htl` is *applied*, as a function, to its argument, while the Nuprl `tl` is a defined operator with a single operand (Nuprl also has an operator for function application, of course). We have used a notational device to suppress type arguments in the (pre-rewrite) imported theorem. Each of the imported constants in the theorem actually has at least one type argument. In the rewritten theorem, there are no hidden type arguments (the Nuprl operations are “implicitly polymorphic”).

The most interesting point in this translation is the function for head of a list. In HOL, this is a *total* function on lists. When we import it into Nuprl, we must prove that the interpretation returns a value on every list, empty or not. Since `hhd` is polymorphic, given an arbitrary type and the empty list as an argument, it must choose some arbitrary member of the type as output. Thus we must give `hhd` a nonconstructive definition in Nuprl. However, we can prove that this function is the same as Nuprl’s `hd` when the list is non-empty. This gives us a conditional rewrite which goes through for this example theorem.

In general, the rewriting of imported theorems is completely automatic, except for cases like `hhd`, where we may occasionally be left with a subgoal to verify the condition of some conditional rewrite. Usually such conditions are proven automatically.

One obvious limitation of the implemented connection between Nuprl and HOL is that it is one-way. It should be possible to reverse the rewriting described above, so that many Nuprl theorems could be written into a form appropriate for HOL, but there would remain the

problem that the theorem’s constants are interpreted, i.e. are defined as particular Nuprl terms, whereas in a counterpart in HOL the constants would be abstract, i.e. declared in a theory signature and possibly constrained by axioms. Although the constructive interpretation of HOL mathematics is ultimately useful in Nuprl, there is no fundamental reason why much of the mathematics built in Nuprl on top of imported math could not be done at the same level of abstraction as in HOL, delaying interpretation until necessary.

We plan to expand the Nuprl/HOL prototype link in several ways. First, we will use the mechanisms described elsewhere in the proposal for structuring and organizing Nuprl libraries to account for HOL’s notion of theory. Building on this, we will make the connection 2-way, allowing certain chunks of mathematics developed in Nuprl to be exported to HOL. Also, there are a number of practical problems that need to be addressed. These include: general robustness and ease of use; incrementally exchanging growing theories; system support to help a user ensure that their mathematics is exportable, when desired; more automation for determining interpretations of constants from HOL theories within Nuprl; and more automation and organization for specifying the correspondences used in rewriting, e.g., imported HOL facts into their Nuprl-friendly form.

We see no fundamental problem to implementing a connection between PVS and Nuprl along the Nuprl/HOL lines discussed above. Other provers, for example Coq [6], should be possible too. This gives the possibility of constructing shared libraries of basic mathematics. Some libraries, like list theory, will be basic enough that they can be shared between most systems that can be interpreted in the LPE. Of course, this will often require rewriting to put the theorems in the form most appropriate for the particular prover. Other libraries may rely on features, such as higher-order logic, present in only some systems.

One hindrance to this idea of sharing mathematics is that currently, most theories are simply not designed for reuse. Users tend to do demand-driven development of basic theories, only doing enough to complete the proof of the ultimate theorem of interest, and investing little thought in abstractions that would make the theories more reusable. This is a problem not unique to theorem-proving, of course, and the solution should be mostly a matter of motivation and will.

B.4.2 Mathematics of logical library structure

The Nuprl Library Manager is the common foundation for sharing mathematics; it provides clients with utilities for building, storing, and sharing, collections of “session objects”. Session objects are things such as expressions, rules, definitions, programs, proofs, and informal items such as documentation or explanations.

The Nuprl system includes a “native” inference method, which comprises execution of proof tactics written in ML, and with a specifically defined method of interpreting inference rules and definitions, but the Nuprl Library Manager is not dependent on those details. Indeed, one may define alternative inference methods, implemented as processes called “refiners.” An example is Nuprl-Light. It has its own concepts of how rules and definitions are to be specified. It is possible for a proof in a Nuprl library to call different kinds of refiners for different inference steps within it, because each inference step may be expressed as an object content, thus one way to interface to a new or foreign system is to call it as a new kind of

refiner.

Now, let us consider how Nuprl relates session objects to each other. Objects used in a client session are named, and there is a method for including object names inside expressions that are the content of objects, thus implementing pointers between objects.

A key service provided by the Library Manager is to enforce a discipline for building named collections so that collision can be easily avoided in selecting new identifiers and object addresses, without explicit interaction between users. This is effected by making such identifiers and object addresses *abstract*, and we have assimilated abstract identifiers to abstract object addresses to avoid duplicating name management; so allocating a new abstract identifier is allocating a new object in the session collection. There are infinitely many abstract identifiers, so one can always find new ones, and one can always determine identity between identifiers; no further structure is supported.

The Library Manager includes a repository of such object collections, and one may store a client session's collection, or a subcollection, under an externally significant name, such as a string, into the library. When one retrieves a stored collection, however, one is not guaranteed to get a collection back with the same addresses – one is only assured of getting a “similar” collection, which is like the original except for uniform renaming of the objects throughout the collection, including adjustments of the object addresses mentioned within object contents.

For various purposes, one may define notions of “Coherency” for collections of session objects, and the Library Manager helps maintain and combine coherent maps. The basic coherency condition is that if some object contains an object address, then the addressed object is also in the collection. Another typical coherency condition would be that there are no two objects that define the same operator, and that definition chains are well-founded. Obviously one would expand the coherency conditions to assist in maintaining variously “meaningful” collections of interdependent objects.

The structure of session collections described here is essentially “flat.” When one wishes to impose more structure on a collection of objects, such structure may be expressed as contents of other objects created for that purpose, such as the Module System described elsewhere in this proposal. Yet one may always ask after the “internal” relations between objects determined by their contents, and may choose to impose different external structures on the collection by description.

Here are some examples of operations we wish to implement on session collections:

- The smallest coherent subcollection that includes specified objects.
- The largest coherent subcollection omitting specified objects.
- Add some new objects to a collection, or utilize some new identifiers.
- Duplicate part of a collection, making up new names and adjusting pointers within the new object contents.
- Merge two collections, maintaining a correspondence between common parts.
- Collapse a collection by identifying some of the objects.
- Change the content of an object.

B.4.3 Formal module systems

The logical environment Nuprl-Light organizes the contents of the logical library using theory modularity, as we discussed in Section ???. The formal module system we propose provides the features of standard programming modules, promoted to the formal domain based on the mathematical practice of the last century.

Within this module system, the root of a mathematical theory is defined as a module containing the fundamental definitions of the theory, and formal features are defined in submodules that are based on the formal definitions. For instance, the module structure of the Nuprl type theory is shown in Figure 2.

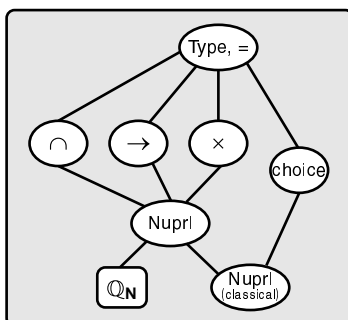


Figure 2: Modular design of the Nuprl type theory

The root theory in the Figure is a module that defines only the most fundamental features of Nuprl: the typing and equality judgments. The modules below the root define the properties of each of its type constructors: function and product spaces, disjoint unions, etc. Finally, the constructive Nuprl theory joins the common constructors together to form the computational Nuprl type theory.

Most of the current knowledge in Nuprl is expressed in the standard Nuprl theory. The classical Nuprl theory is obtained by adding the choice operator. The framework allows the classical theory to inherit all the results of the constructive theory.

B.4.3.1 Relations between mathematical domains To illustrate the problems in relating logics as theories, we return to the problem of relating results in HOL to Nuprl. The HOL theory construction is similar to the Nuprl construction, except that HOL has a simpler classical type theory, and the construction has correspondingly fewer modules. The goal of relating HOL to Nuprl is shown by the dotted arrow between the HOL and Nuprl-HOL theories in Figure 3. Any such relation must justify the rules for each of the modules making up HOL from the rules in Nuprl.

Theory relations to justify one theory from another. In the $HOL \rightarrow Nuprl$ example, the relation is stated so that results from HOL can be used in Nuprl. The *justification* is in the opposite direction: the HOL theory is justified in terms on the Nuprl theory, providing a Nuprl interpretation of the HOL results.

Once this relation is made, the Nuprl-Light framework automatically translates the HOL results into Nuprl. This provides a raw interpretation of the HOL results, but it is also

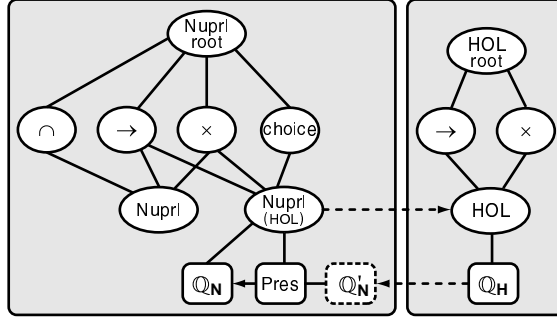


Figure 3: Relating HOL to Nuprl

necessary to adapt the presentation because the existing Nuprl formalization of the rational numbers differs from the account in HOL. A common presentation must be chosen before the two theories can be combined. The first step in finding the relation is to provide a transformation between the basic representations used in the two theories. This provides a means for the logical environment to transform the remaining knowledge automatically, although intervention may be required to summarize the relation intuitively.

B.4.3.2 Inheritance of results One of the key features of the logical environment is automation. One way in which this gets exhibited in Nuprl-Light is in *inheritance* of results. As we described, type theories are typically constructed by refinement. For instance, the classical Nuprl theory is the constructive theory refined with the axiom of choice. Mathematically, the constructive results are valid in the classical theory, and Nuprl-Light automates that task of importing all the theorems and proof tools to the classical subtheory.

The same mechanism is used to transfer results between theories. Once the relation is justified (or asserted) between Nuprl and HOL, the results from HOL, including the theorems and proof tools are made available in Nuprl.

B.4.3.3 Reflection with the implementation language In any formal framework, the language used for proof automation provides a key link between logic and knowledge. Proof automation is used to rephrase the fundamental mathematics into proof intuitive procedures specific to mathematical domains.

Our approach in Nuprl-Light integrates the formal module system with the OCaml programming language modules, providing a form of reflection in the system. We can use the formal system to speak about its implementation. The formal effect of this reflection is to add a new semantics of ML to the module structure. For our work in Nuprl, a formal interpretation of the OCaml semantics is given in Nuprl, allowing Nuprl to be used to reason about OCaml programs. At the programming level, the effect is to augment OCaml modules with formal statements.

In practice, there are two types of modules—those that define the type theory, and those that define ML programs. For instance, the module in Figure 4 defines a partial description of the Nuprl logical implication.

This module is derived from the root Nuprl module `NuprlRootSig`, and it defines a new

```

module type NuprlFunSig = sig
  include NuprlRootSig
  declare  $A \Rightarrow B$ 
  axiom modus_ponens :  $\frac{\Gamma \vdash A \Rightarrow B}{\Gamma, A \vdash B} \quad \Gamma \vdash A : Prop$ 
end

```

Figure 4: Implication module fragment

syntax for a type constructor $A \Rightarrow B$. The final `axiom` statement defines the propositional behavior of implication: $A \Rightarrow B$ is true if whenever A is true, then B is true.

The module construction for reasoning about programs has a similar layout. The module in Figure 5 is used to define nonempty sets of numbers. This module uses the Nuprl interpretation of OCaml (defined in `NuprlSig`) and it defines a type `t` to represent non-empty sets with three operators: `singleton` creates a set with one element, `union` combines two sets, and `mem` tests for set membership. Finally, the remaining two statements give the formal verification properties for the set: the singleton set really contains one element, and the union contains exactly the elements in the sets being combined.

```

module type SetSig = sig
  type t
  val singleton : int -> t
  val union : t -> t -> t
  val mem : int -> t -> bool
  axiom sing_mem :  $\forall a, b: \text{int}. (a = b) \Leftrightarrow (\text{mem } a \text{ (singleton } b))$ 
  axiom union_mem :  $\forall a: \text{int}. \forall s_1, s_2: t. (\text{mem } a \text{ (union } s_1 \text{ } s_2)) \Leftrightarrow ((\text{mem } a \text{ } s_1) \vee (\text{mem } a \text{ } s_2))$ 
end

```

Figure 5: Set module

B.5 New capabilities

Doing the work we propose will result in significant and interesting new capabilities. First, it will be possible to share results. It will be possible to use the appropriate logic engine for the task at hand and the appropriate logical theory. For example, a computational theory like Nuprl can talk about complexity constraints in its specifications. No other logic can do this, but in the LPE it will be possible to use this rich language. An indirect consequence of this accomplishment will be an incentive to share basic results that will motivate and enable other forms of cooperation.

In this section we comment on how the LPE will be useful in the on-going verifications at Cornell and Bell Labs. We also discuss how it will enable a broader use of the method of programming by extracting code from proofs. As Howe demonstrated, this capability can now benefit from theorems proved in noncomputational logics.

B.5.1 Bell Labs verifications

Along with Amy Felty and Frank Stomp at Bell Labs, Howe has been using Nuprl to verify the portion related to cache coherence of the SCI (Scalable Coherent Interface) protocol [17]. This protocol is an IEEE standard for specifying communication between shared memory multiprocessors, and is called scalable because it is intended for systems which may consist of up to 64,000 processors. An attempt to validate the program verified in the current paper has been made by the model checking community. Holzmann using SPIN [12], Kurshan using COSPAN [19], and Long and McMillan using SMV [21] report to have validated the program for up to five processes (at which point the state explosion problem takes its toll). Stern and Dill [32] have encountered similar state-explosion problems using Mur ϕ , though they were nevertheless able to find several errors in the C-code for the protocol.

The reason this particular protocol was chosen for a verification project was because of its complexity, because of its general similarity to communication protocols of interest to Lucent, and also because of local expertise. The current Nuprl effort directly formalizes the proof described in the paper [8], and does not involve any external tools. We plan to test the LPE by redoing the verification, but using model-checkers to speed up the process, for example by verifying some of the properties using formally justified abstractions. The connection to model checkers could be direct, or could be imported from, e.g., HOL or PVS, via the planned sound link with these systems.

B.5.2 I/O automata and verification of group communications software

We expect to use the LPE in the on-going verification of Ensemble. We will explore using PVS formalizations of protocols in terms of I/O automata of Lynch [?] being done at the Naval Research Laboratory. Below we illustrate this method on a simple example.

One of the protocols of the Ensemble group communication systems is implemented by the Elect layer. Its purpose is to elect a coordinator for the group from among the processes which are not suspected to have failed. Initially and on specific request, one coordinator is selected externally while at runtime each process has to decide whether it has to elect himself coordinator if other processes have failed. To make sure, that the subgroup of correct processes will always have exactly one coordinator, each process maintains a list of suspected processes, which will be updated in each cycle, and its own rank in the group. The following extract from the original code of Ensemble's Elect layer shows how the election algorithm is realized.

```
let hdlrs s (ls,vs) {up_out=up;upnm_out=upnm;dn_out=dn;dnlm_out=dnlm;dnnm_out=dnnm} =
.
  let upnm_hdlr ev = match getType ev with
  | EInit -> upnm ev ;
          if ls.rank = vs.coord then (
            Once.set s.elect ;
            dnnm (Event.create EElect[])
          )
  | EElect -> Once.set s.elect ;
            upnm ev
  | ESuspect -> s.suspects <- Arrayf.map2 (||) s.suspects (getSuspects ev) ;
```

```

        if Arrayf.min_false s.suspects >= ls.rank then (
            Once.set s.elect ;
            dnm (Event.create Elect[])
        )
    | _ -> upnm ev
    .

```

This code corresponds to a finite I/O automaton whose states are encoded in the (parameterized) variable s , whose outputs are described by calls to outgoing event handlers ($upnm\ ev$, $dnm (Event.create \dots)$). The automation distinguishes the direction of an input event (Up, Dn), its type ($Init, Elect, Suspect, \dots$), and a list of suspected processes which is empty if the type is not $Suspect$. Its transition table begins as follows

$s_{coord, sus, e}$	$Up, Init, S$	$sr, sus, true$	$[Up, Init, S; Dn, Elect, []]$
sr, sus, e	$Up, Init, S$	sr, sus, e	$[Up, Init, S]$
sr, sus, e	$Up, Elect, S$	$sr, sus, true$	$[Up, Elect, S]$
$s_{\mu_i[(susS)[i]=false], sus, e}$	$Up, Suspect, S$	$sr, susS, true$	$[Dn, Elect, S]$
sr, sus, e	$Up, Suspect, S$	$sr, susS, e$	$[]$
sr, sus, e	$Up, *, S$	sr, sus, e	$[Up, *, S]$
	\vdots		\vdots

Since we have developed tools for importing Ensemble code into the Nuprl system, we can treat the actual implementation of Ensemble as Nuprl terms and reason about them. This makes it possible to relate the running code to the I/O automaton and prove by partial evaluation (i.e. Nuprl’s computation rules) that the code computes the correct state and output events for each line in the automaton’s transition table.

Furthermore we can now state the above mentioned properties of the layer and prove formally that the code makes sure that there is always exactly one coordinator.

B.5.3 Program Extraction

The ability to extract “correct-by-construction” programs directly from proofs is the most notable feature of constructive type theories as implemented in Nuprl [5] and related systems [20, 7, 23]. As Howe demonstrated [], it is possible to use these methods even with classically proved equations from HOL. Since the LPE will widen the applicability of this methodology, we propose to improve its practical effectiveness.

Despite substantial experience with this extraction methodology since 1984 when we introduced it, there still is no established software engineering practice based on the extraction of programs from constructive proofs. We have an opportunity to improve the technology for this as we build the LPE since it will include the Nuprl logic engine and extractor.

A key obstacle to the transition of program extraction methods into practice is the lack of a methodology for extracting clear and readable programs from formal proofs. Often, the extracted programs contain unnecessary structure, artifacts of the constructive proof, that do not contribute to the computational content of the extracted term. Researchers

have addressed this problem in various constructive systems in an attempt to improve the efficiency, readability, and understanding of extracted programs [25, 15, 26].

Nuprl is unique among the existing systems in its ability to formalize and verify induction principles as lemmas within the type theory that yield extracts that are recursion schemes optimized for both readability and efficiency.

The proposed investigation under this item is the development of libraries of induction principles having efficient recursion schemes as their extracts. This extends work in [3].

B.5.3.1 Extraction of non-local control operators from Nuprl proofs. The use of certain non-constructive logical constructs such as excluded middle $P \vee \neg P$ and Peirce's law $((P \Rightarrow Q) \Rightarrow P) \Rightarrow P$ can be computationally interpreted as non-local control operators such as call-with-continuation (call/cc).

In [4] a heuristic search algorithm, conflict-directed back-jump, is developed in a Nuprl like type theory extended to extract call/cc from occurrences of Peirce's law. The computationally significant portions of the proof we formalized in an extended Nuprl system and the extracted term was translated into Scheme.

The goal of this part of the research is to build on [4] and establish methodology for the routine (but limited) use of non-constructive constructs in Nuprl proofs to yield efficient programs that utilize non-local control.

C Deliverables (1 page)

HOL/Nuprl linked system
Nuprl 5 system with logical library
Applications to Ensemble plus

{Need more}

D Statement of work (3 page expansion of E)

We will work to accomplish four main goals and one optional goal if funding is provided for it.

1. Release an open LPE system containing these key components: HOL/Nuprl linked logic, Common Logical Library, Module system
2. Demonstrate uses of the LPE on parts of on-going software verification (Ensemble, SCI) and on a verified decision procedure and a synthesized procedurs.
3. Explore integration of PVS and possibly other verifiers and checkers to produce on HOL/Nuprl/PVS logic engine for the LPE. Demonstrate whatever integration is achievable.
4. Explore exporting verified decision procedures and possible general reflection capabilities for the hybrid logic engine.
5. Optional. Provide standard editor interface for the LPE and its Common Logical Library; demonstrate integrated proof capabilities in Nuprl and PVS which will be components of LPE logic and will be integrated into the GrammaTech Formal Methods Environment (FME).

D.1 Goal 1

To achieve Goal 1 we need to move proof of concept into a standard system:

1. Create a module directory corresponding to the Nuprl theory that links to HOL;
2. Need to automate and enforce the connections done by Howe, establish sentinel structure;
3. Extend scope of convection to other types and explore PVS theory.

This requires: implement abstract addresses; upgrade Nuprl library to Common Logical Library (CLL); define proof sentinels; augment editors to produce standard displays; and modularize tactic library. This will be a two-year effort running concurrently with Goals 3, 4 and 5.

This work is based on building a Common Logical Library that supports the features we mentioned in Section ???. Some of the operations that we propose to implement on collections of mathematical knowledge are the following.

- The smallest coherent subcollection that includes specified objects. For example, one might indicate a few theorems one is interested in, and select out just the things needed to make sense of them, such as definitions, lemmas, and rules needed to prove them. There is no reason to make a person load a huge library when only a slice of it is desired.
- Merge two collections by specifying an initial correspondence between some of the object addresses, extending the correspondence if need be as the corresponding object contents are matched (this can fail of course). While one might on occasion wish actually to specify the initial correspondence, it will be far more common to want to merge independently developed extensions to some common collection.

The scenario is, two people load the same collection from the library, then make some additions, and want to merge the results back together, identifying the original objects taken from the library. Of course, since the abstract identifiers (object addresses) attached to the original objects are not guaranteed to be the same for each client, one must somehow specify the initial correspondence for the merge to cause the obvious identifications.

Rather than requiring the users to develop such an initial correspondence preparatory to the merge, the Library Manager will provide such a correspondence heuristically based on its remembering that it generated the two original collections from the same description, even though it may have generated different names.

D.2 Goal 2

To achieve Goal 2 we need to import Nuprl 4 libraries to the Common Logical Library describe Jason's verification ideas

direct ocaml link

Doug's work -- speeding rewrites using modules,...apply to SCI.

D.3 Goal 3

To achieve Goal 3 we need to study the CVS semantics in more detail and examine the proof structure to see what can be done beyond simple sharing of facts. Simultaneously we will work on reflected term structure to help on Goal 4:

1. Establish connection with Math Bus terms.
2. Use HRL proofs in Ensemble verification.

E Schedule of milestones (1 page summary of D)

{Basically for the first year we want to talk about doing what we have well underway, Nuprl 5, Nuprl_light, links to HOL, library design, better extracts, things we can deliver and talk about soon. Then we ramp up to one or two new things that we think we can get done in the first year and tail off with papers and demos and public release.}

Need milestones for what we will accomplish over the 3 years. Who does what. Prioritize and spell out costs.

Year 0 July 98 -- Sept 98

start theoretical work on linking to other type theories
continue work on Common Logical Library

Year 1 Oct 98 -- 99

deploy HOL/Nuprl link more widely --- programming, student
continue design of logical library, plan Web reading as well
experiment with Nuprl-Light for linking theories
continue verification of SupInf
begin implementation of parts of reflection in Nuprl 5 (Term and Eval)
foundations for links to PVS logic
improve quality of theorem extracts, produce ML code, other code?
experiment with logic variables ???

Year 2 Oct 99-00

import some NRL work in PVS on Ensemble-like protocols using I/O automata
export our SupInf to others, link in verified version to Nuprl 5
Web presentation of Common Library released
synthesize and Ensemble protocol
release Nuprl 5/NL/HOL open system
use reflected system to specify timeliness properties based on comp complexity
integrate reflection fully into editor (color coding and all that)
work on another decision procedure or small model checker ???

Year 3 Oct 00-01

demonstrate combined system on Ensemble and Bell Labs applications
incorporate all PVS libraries into Common Logical Library
add further reflection rules and integrate more verified procedures

F Technology transfer (2 pages)

F.1 Supporting a key DARPA role

One of the major barriers to wider use of formal methods in industry is the “catch 22 problem”. Since the market is small, there is little incentive to industry to invest in good tools. But the market will remain small unless there are better tools available to industry. DARPA can help solve this problem by investing in better tools. Our work is aimed at providing better tools.

By its very nature, the work we propose to do significantly impacts the creation of better tools through cooperation among the tool providers.

F.2 Role of universities

Universities also play a key role in technology transfer by training the engineers and scientists, creating the educational material, and making advanced technology accessible through applied research. This role is critical in the formal methods area where the subject is difficult, interdisciplinary and new.

Cornell has been effective in carrying out the university mission in the area of formal methods. We have pioneered some of the best research, produced some of the leading researchers (Ed Clarke, Bob Harper, Rance Cleaveland, Doug Howe, David Basin and Paul Jackson). We have supplied staff for several research labs (Intel, Bell Labs, IBM, and Microsoft). At Intel 3 of the PhD’s in the hardware verification group come from Cornell.

F.3 Industrial partners

Our ties to Bell Labs will help bring the technology to points where it is used. The GrammaTech option is concerned with integrating our tools with standard industrial environments. The SBIR contract under which they receive ONR funding is to build a Formal Methods Environment (FME). Its main goal is

“to speed the adoption of formal methods in industry.”

The overall goal of the company is to transfer university and industrial research into polished products. GrammaTech in preparing its SBIR proposal identified 5 reasons for the fact that formal methods are not yet widely used in industry. Two key reasons that we address in this proposal are:

Tool isolation: valuable multipurpose formal methods nuggets are locked inaccessibly inside monolithic systems, and coupling formal methods tools is currently a very difficult task.

User-Interface inadequacies: existing formal methods are hobbled by weak user-interfaces, and the cost of making better ones often dominates the cost of implementing the core method.

Our proposal directly attacks these problems. We have worked with GrammaTech before to build a standard user interface for Nuprl. The prototype they built for us is the basis for their plans to build one for PVS with SBIR funding. We are proposing that they expand their plans to include an interface to the LPE we are building.

Our previous work with GrammaTech also helped us test our design of an open theorem proving architecture. We wrote an editor interface that will support foreign as well as resident editors.

G Related work (3 pages)

Compare to Jose, Goguen, Talcott, etc.

Kestrel work --- Christoph summaries and compares, especially on program synthesis and transformations. Perhaps on integration of tools as well.

Talcott and Messeguer --- Doug and I try to decipher this.

Harper and Lee --- get Karl Crary to help with this.

HOL ---- Doug can discuss

PVS ---- Jim and Doug might comment, possible help from Jackson
Elf --- Jason

Nuprl-Light addresses the issues of diversity and sharing by providing a modular, object-oriented framework for specifying, relating, and developing type theories and mathematical domains. The framework itself assumes (and provides) no type theory or logic, as in LF [11], and we sometimes call it an *implementation* framework. Instead, Nuprl-Light provides a meta-framework where logical frameworks such as LF, *Nuprl*, set theory, and other theories can be defined and developed. Since proof automation is such a critical part of theorem proving in these logics, the implementation framework is tied closely to a programming language (in this case *OCaml*) and the formal module system is tied closely to the programming language modules.

Like the Isabelle [28] generic theorem prover, Nuprl-Light uses generalized Horn clauses for logical specification. Indeed, specifications in Nuprl-Light appear quite similar to those in Isabelle. However, where Isabelle uses higher order unification and resolution, Nuprl-Light retains a tactic-tree [10] of LCF [27] style reasoning based on tactics and tacticals, and Nuprl-Light also allows theories to contain specifications of rewrites, using the computational congruence of Howe [13]. Like LF, the Nuprl-Light meta-logic also relies on the judgments-as-types principle (an extension of propositions-as-type), where proofs are terms that inhabit the clauses.

The main departure from Isabelle and LF is in the module system. In Nuprl-Light modules have signatures and implementations, providing the ability not only to specify multiple logics, but to relate them (using functors). In addition, modules in Nuprl-Light are object-oriented, providing the ability to extend type theories and their reasoning strategies incrementally. Taken together, these abilities provide a view of “theories-as-objects,” encouraging incremental and modular specification of theories.

The SPECWARE system of Kestrel Institute [30, 31] already provides a visual interface for managing the links between formal theories as soon as they are established semantically. Although it uses category theory as formal foundation and focuses mostly on refinements of formal specifications instead of links between general mathematical domains, we believe that the some of the techniques developed for Specware could also be used in our approach and vice versa.

The KIDS system of Kestrel Institute [29] is a tool for automatically synthesizing executable code from formal specifications which have been very successful in practical applications [?, ?]. Its algorithm design tactics are based on *algorithm theories*, which describe the axioms for the correctness of a particular algorithmic structure, on program transformations, which rely on a knowledge base describing the properties of fundamental operations, and on techniques for *data type refinement* [2].

An even more integrated approach to specification and program development is Kestrel's SPECWARE system [30, 31], which will eventually be used to reimplement the techniques behind KIDS on a higher level. SPECWARE supports the modular construction of formal specifications and their refinement into executable code, using concepts from category theory for establishing relations between algebraic theories or constructing new theories. It provides a highly a visual interface for managing the links between formal theories as soon as they are established semantically.

Despite fundamental differences in the formalizations we believe that there are many similarities between the SPECWARE / KIDS approach and the work done at Cornell and that both approaches could profit from each other.

- We could use some of the techniques developed for SPECWARE (maybe even an adapted version of the system itself) as tools for *implementing* the morphisms between Nuprl, HOL, and PVS which we are currently investigating. This would result in a proof system that can easily import results from others.
We could also use the same techniques as mechanism for establishing a theory structure in Nuprl's library.
- On the other hand we could use higher-order reasoning capabilities of the Nuprl system as a mechanism for checking the correctness of the formal knowledge base and used by SPECWARE and KIDS. In particular we could implement theory morphisms as *derived inference rules* which were created from a more abstract description. This would establish a relation between the theoretical work on paper and their implementation in SPECWARE and KIDS and, by improve the efficiency and clarity of the implemented mechanisms (see [18] for an exposition).

By establishing a link between theorem proving and algebraic refinement techniques we would on the one hand improve the automated synthesis and verification capabilities of proof-based systems. On the other hand we would harden the security of program synthesis systems and the software generated by them.

H Key personnel

Vitas for Doug, Bob, Christoph

I Facilities

List Cornell equipment and Bell Labs equipment.

J Cost of tasks and budget

Year 0 Year 1 Year 2 Year 3

References

- [1] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Symposium on Logic in Computer Science*, pages 95–197. IEEE, June 1990.
- [2] Lee Blaine and Allen Goldberg. DTRE—a semi-automatic transformation system. In B. Möller, editor, *Proceedings of the IFIP TC2 working conference on Constructing Programs from Specifications*. ELSEVIER, 1991.
- [3] James Caldwell. Moving proofs-as-programs into practice. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 1997.
- [4] James Caldwell, Ian Gent, and Judith Underwood. Search algorithms in type theory. Manuscript available at <http://www.cs.cornell.edu/Info/People/caldwell/index.html>, September 1997.
- [5] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
- [6] Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Gérard Huet, Pascal Manoury, Christine Paulin-Mohring, César Muñoz, Chetan Murthy, Catherine Parent, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant reference manual. Technical report, INRIA, 1995.
- [7] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide*. INRIA, Version 5.8, 1993.
- [8] Amy Felty and Frank Stomp. A correctness proof of a cache coherence protocol. In *Proceedings of the 11th Annual Conference on Computer Assurance*, pages 128–141, 1996.
- [9] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In *Fourteenth International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Computer Science*, pages 351–365, Berlin, 1997. Springer-Verlag.
- [10] T. G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems*. PhD thesis, Cornell University, 1988.

- [11] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS '87.
- [12] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall Software Series, 1991.
- [13] Douglas J. Howe. Equality in lazy computation systems. In *Proceedings of Fourth Symposium on Logic in Computer Science*, pages 198–203. IEEE Computer Society, June 1989.
- [14] Douglas J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings of Sixth Symposium on Logic in Computer Science*, pages 162–172. IEEE Computer Society, 1991.
- [15] Douglas J. Howe. Reasoning about functional programs in Nuprl. *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Lecture Notes in Computer Science, Vol. 693, Springer, Berlin, 1993.
- [16] 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 *LNCS*, pages 85–101. Springer-Verlag, Berlin, 1996.
- [17] IEEE-P1596-05Nov90-doc197-iii. *Part IIIA: SCI Coherence Overview*, 1990. Unapproved Draft. Approved standard is described in IEEE Std. 1596-1992 “The Scalable Coherent Interface”.
- [18] Christoph Kreitz. Formal mathematics for verifiably correct program synthesis. *Journal of the Interest Group in Pure and Applied Logics*, 4(1):75–94, February 1996.
- [19] R. P. Kurshan. Analysis of discrete event coordination. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness (REX Workshop)*, pages 414–453. Springer Verlag Lecture Notes in Computer Science 430, 1989.
- [20] Martin Löf. Constructive mathematics and computer programming. In *Proceedings of the 6th International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.
- [21] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [22] P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.
- [23] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.
- [24] Sam Owre and Natarajan Shankar. The formal semantics of pvs. Technical report, SRI, August 1997.

- [25] Christine Paulin-Mohring. Extracting F'_w s programs from proofs in the calculus of constructions. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 89–104, 1989.
- [26] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ml programs in the system coq. *Journal of Symbolic Computations*, 15:607–640, 1993.
- [27] L. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, NY, 1987.
- [28] L. Paulson and T. Nipkow. Isabelle: a generic theorem prover. *Lecture Notes in Computer Science*, Vol. 825, 1994.
- [29] D.R. Smith. KIDS—a knowledge-based software development system. In M.R. Lowry and R.D. McCartney, editors, *Automating Software Design*, pages 483–514. AAAI Press / MIT Press, Menlo Park, CA, 1991.
- [30] Y. V. Srinivas and Richard Jüllig. Specware: Formal support for composing software. In *Proceedings of the International Conference on the Mathematics of Program Construction*, 1995.
- [31] Y. V. Srinivas and James L. McDonald. The architecture of SPECWARE, a formal software development system. Technical Report KES.U.96.7, KESTREL, 1996.
- [32] Ulrich Stern and David L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.
- [33] Carolyn Talcott. A theory for program and data type specification. *Theoretical Computer Science*, 104(1):129–159, 1992.
- [34] Benjamin Werner. Sets in types, types in sets. In *International Symposium on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

Section III References and Additional Information

We can include 3 relevant papers. I think that we should include at least these two:

1. Importing Mathematics from HOL into Nuprl
(perhaps delete any negative comments about Nuprl semantics)
2. CADE 98 paper on Ensemble, Christoph, Jason and Mark.
3. An Implementation Framework for Programming Logics with Reflection
Jason, Karl and Bob