# The Fox Project: Advanced Language Technology for Extensible Systems

Robert Harper, Peter Lee, and Frank Pfenning

Carnegie Mellon University
Pittsburgh, PA 15213

January 1998

CMU–CS–98–107

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU-CS-FOX-98-02

This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050.

## Abstract

It has been amply demonstrated in recent years that careful attention to the structure of systems software can lead to greater flexibility, reliability, and ease of implementation, without incurring an undue penalty in performance. It is our contention that advanced programming languages—particularly languages with a mathematically rigorous semantics, and featuring higher-order functions, polymorphic types, and a strong module system—are ideally suited to expressing such structure. Indeed, our previous research has shown that the use of an advanced programming language can have a fundamental effect on system design, leading naturally to system architectures that are highly modular, efficient, and allow re-use of code.
We are thus working to demonstrate the viability and benefits of advanced languages for programming real-world systems, and in particular Active Networks. To achieve this, we have organized our research into the areas of *language technology*, *safety infrastructure*, *compiler technology*, and *applications.* This report describes the current plans for this effort, which we refer to as the Fox project.

Authors' electronic mail addresses:
`Robert.Harper@cs.cmu.edu`
`Peter.Lee@cs.cmu.edu`
`Frank.Pfenning@cs.cmu.edu`

**Keywords:** Standard ML, computer networks, modules, type theory, compilers, programming languages

## 1 Motivation

In a large heterogeneous environment, it is a significant challenge to support the use of mobile code without sacrificing safety, reliability, or performance. Achieving this is fundamentally a question of supporting large-scale modular construction of software. To ensure predictability and reliability of the composite system, the possibilities of interaction between components must be specified and enforced. The forms of interaction include not only familiar notions of interfaces, or API's, to which components must adhere, but also encompass more fine-grained properties such as bounds on resource consumption. These properties must be precisely specified so that they may be enforced; without specification, enforcement is impossible. Conversely, specification without enforcement is at best an exercise in documentation, rather than a means of ensuring integrity. In short, the core enabling technology for achieving dynamic extensibility of software systems is *verified encapsulation* of system components.

The goal of an Active Network is to exploit such dynamic extensibility of software systems to achieve unprecedented degrees of flexibility, robustness, and efficiency in modern high-speed networks [25]. The key idea is to use "active packets", or "capsules", that contain code to be incorporated into a network router or other active component of a network to modify its behavior in an application-specific manner. Such a capability promises to improve flexibility and quality-of-service in the network, particularly in multicast applications, as well as accelerate evolution of the networking infrastructure. The power of this technique is essentially unlimited: in principle, the active node could be completely re-programmed by an active packet! Obviously the behavior of active packets must be limited so that malicious, or otherwise undesirable behavior is precluded. It is the responsibility of the active network component to specify the "rules of engagement" for active packets, and to ensure that these rules are respected in order to protect the integrity of the network.

How might this be achieved? Current methods rely mainly on *dynamic checking* to ensure that the behavior of components is suitably constrained, for example, through the use of *interpreted virtual machines* [13, 26]. Components are written in an abstract machine code that is inherently safe to execute on an active node — the code is executed by a trusted interpreter that executes on behalf of the component. This amounts to an elaborate set of run-time checks to ensure safety. Of course this raises problems of efficiency, since the overhead of interpretation and dynamic enforcement is substantial. Efficiency can be regained, to some extent, by using "just in time (JIT)" compilers [21] to translate virtual machine code to real hardware instructions. However, run-time checks to ensure safety are still required, and, what is worse, the complexity of the trusted code base increases substantially to include the JIT compiler itself, thereby reducing the reliability of the network as a whole.

Another approach to run-time encapsulation is Software Fault Isolation (SFI) [28]. To avoid interpretation overhead, machine binaries are edited at load time to ensure that memory accesses are limited to a specific allowed range so that the component at worst self-destructs, rather than destroys the context in which it executes. SFI is most effective for imposing coarse restrictions on memory usage; its performance degrades rapidly when more fine-grained restrictions are imposed.

Besides the performance problems, run-time encapsulation techniques also suffer from being tied to a specific notion of "safe" execution. The trusted computing base (for example, the virtual machine) must be constructed so as to enforce all of the safety rules, and if new rules are added later then this computing base must be modified accordingly. This can be a nontrivial task. Take, for example, the case of extending a Java Virtual Machine so that it enforces the safety policy that any applet terminates within a bounded number of machine cycles. Making such a change to the JVM (or JIT compiler) so that it does not impose an unreasonable performance penalty may be far from easy.

Rather than attempt to *enforce* restrictions on the behavior of components, some proposals focus on *assigning responsibility* for any such failures. This is achieved by requiring that components be signed by the author so that any malicious or undesirable behavior can be traced to a specific responsible party. Obviously this approach does nothing to ensure the integrity of the running system. Moreover, the overhead of digital signature schemes based on public-key cryptography is substantial.

Fundamentally, we are facing a tension among efficiency, reliability, expressiveness, and enforceability of encapsulation constraints. Dynamic enforcement methods impose considerable overhead at execution time. Many attempts to improve efficiency tend to increase the size of the trusted computing base, limiting reliability and undermining integrity assurances. Increasing the expressiveness of encapsulation constraints leads to greater flexibility in writing components, but also increases the difficulty of checking compliance.

How can these tensions be resolved? It is our contention that these fundamental issues are most profitably addressed from the standpoint of programming language technology, especially type systems. To achieve verified encapsulation of system components requires a formalism for specifying and checking properties of programs. Type systems provide a general framework for specification and checking that has proved to be remarkably effective in a wide variety of settings, ranging from type systems for programming languages such as ML [14] to sophisticated formalisms for specifying and checking the execution behavior of programs, such as Proof-Carrying Code [17, 16]. Type systems have these desirable properties:

1. *Scalability.* Types can be used to express a wide range of program properties, ranging from simple data representations (*e.g.*, int's and float's) to sophisticated data structure invariants (*e.g.*, coloring invariants on red-black trees or restrictions on array accesses).

2. *Static checkability.* Type constraints are statically checkable, usually very efficiently in practice, if not in principle. Type checkers are typically small and easily implemented, reducing significantly the size and complexity of the trusted computing base.

3. *Compatibility.* Static type systems *generalize* dynamic type systems by admitting both statically- and dynamically-enforced constraints. The effectiveness of static checking can often be enhanced by judicious use of dynamic checks; conversely, dynamic checks can often by eliminated in a sufficiently rich static type discipline.

We propose to exploit these properties of type systems in the development and deployment of basic tools and technologies for extensible systems, and in particular for Active Networks. We will do this by developing the necessary theoretical foundations for specifying and managing modularity in large-scale distributed systems, and developing the tools in the form of certifying compilers and PCC infrastructure so that Active Network developers can exploit the theory.

## 2 Approach

Our proposed work goes beyond the application of theory to practice; indeed, there is a synergy between language design, systems building, and compiler technology that benefits both theory and practice together. In our previous work in the Fox Project, this kind of synergy has sparked a considerable amount of follow-on work, particularly in advanced network design, for example in the Cornell Ensemble system [27] and the UPenn Switchware

Project [1]. There have also been significant interactions with the Utah Flux [7] and Washington SPIN Projects [2]. Here, we illustrate this synergy by briefly reviewing the Project's FoxNet reconfigurable network protocol suite [3], and seeing how its development was instrumental in leading to the TIL type-directed compiler [24] and the Proof-Carrying Code [17] approach to safe mobile code.

The FoxNet system is a collection of network protocol modules. The modules are designed to be highly composable so that high-performance network systems can be quickly and reliably constructed and reconfigured. The key idea behind the FoxNet is to use the type system of Standard ML (SML) to define interfaces that mediate the interaction amongst the layers of a network protocol stack. For example, a protocol interface specifies that every protocol module in the FoxNet must contain, in addition to other components, an implementation of host addresses and network connections, as well as a method for establishing a connection on a specified address. This signature specifies only that these components must exist in each protocol layer, but it does not specify their implementation.

In fact, the definitions of addresses and connections vary from protocol to protocol, though the interface itself remains stable. Consequently, when building up a network protocol stack, the upper layers can know very little about the implementation of the lower layers. This information hiding makes it possible, for example, for the FoxNet's IP protocol module to be layered on top of both the Ethernet and ATM modules without any change to the IP module. Such highly reliable "mix-and-match" composability of components is absolutely essential for any secure extensible system, and to make this work it is critical that stable abstract interfaces be definable and enforced by the language.

The standard problem with such high degree of composability is how to get good performance out of the system. The presence of abstract types can make it extremely difficult to optimize the network code, because basic knowledge about data representations (*e.g.*, can addresses and connections be passed in registers?) and calling conventions (*e.g.*, where is the "connect" function, and what is its calling convention?) are not exposed through the protocol interface.

The traditional solution, which is taken by Java, Modula-3, Scheme, Lisp, and previous implementations of ML, is to impose a universal representation on values. This technique hinders good performance, however, because it requires run-time unpackaging of values and imposes general space and time overheads. It also complicates interoperability with C and Fortran programs, with operating systems, and with devices.

Our solution is to use natural representations of all data structures, and then dispatch on type information that is tracked through all phases of the compiler. This allows the compiler to eliminate the dispatch when type information can be inferred statically. The result is a dramatic performance improvement for SML programs, with most benchmarks running 2 to 5 times faster and requiring less than half the memory of previous compilers.

What does this have to do with mobile code? The types that are propagated throughout the TIL compiler provide internal consistency checks via type checking. Each phase of the compiler works with a typed intermediate language (IL) which ensures that the compiler preserves type safety of original source program throughout all optimizations. Thus, TIL intermediate code is "self-certifying". In a mobile code scenario, by shipping typed IL, the recipient is assured of certain safety properties (*e.g.*, no memory faults) simply by type checking.

Of course, this is essentially the same approach taken by the Java system. And by looking at Java, it is clear that there is still a problem: In order for the IL to have an easily checkable type system, it seems necessary to use a relatively high level intermediate language (in the case of Java, the Java Virtual Machine language). But, this requires the recipient of the code to execute this high level language! If this is done by interpretation, there is a substantial interpretive overhead. If a just-in-time compiler is used, there is the increase in latency, as well as the risk that the compiler (which will be complex) might go wrong. Can we avoid these overheads and just ship binaries?

By using ideas from type theory, formal semantics, and logic (yet again!), we can find the answer to this question in Proof-Carrying Code (PCC). The idea here is to equip the code with a proof that it respects the safety policy. By using technology developed within the Fox Project, in particular the Logical Framework (LF) and our implementation of it in the Elf system, we have been able to develop efficient representations of proofs that can be transmitted digitally and quickly checked for validity. Once again, types provide the formal basis for this approach. The proofs are represented in a type system. Proof checking is then reduced to type checking (which is decidable). Thus, by making the code producer provide the proof, the recipient has little work to do. The result is provably safe mobile code, without the performance overheads of previous approaches.

There is one final problem to solve: How can we find these proofs? To-date, we have used standard theorem-proving technology, but since this is only semi-automated, the process can be difficult to carry out. Some properties (*e.g.*, memory safety) for simple programs (*e.g.*, packet filters) seem to work well. But in order to scale up, we are currently building a Certifying Compiler which produces the proofs automatically.

By following this approach, we have been extremely successful in developing innovative and influential core technologies for building reliable, configurable, and extensible system components. Such achievements are made possible by applying programming language theory to the practical problems of building complex systems. Our developments of type-directed compilation in TIL, and of Proof-Carrying Code, have been in direct response to these advanced system-building needs, and we believe that their use will become critical in the Active Networks effort.

## 3 Planned Research

We propose a comprehensive program of research to apply the theoretical foundations of programming languages—including ideas in type theory, formal semantics, and logic—to the development of tools and techniques for Active Networks. The main goal is the development of technologies for achieving *modularity*, *efficiency*, and *safety* in software systems, and then developing tools to support *application* of these technologies specifically to Active Networks. To achieve this, we have organized the proposed work into four areas:

1. **Language Technology.** The development of foundations and design principles for the specification, management, and implementation of modular software systems. Work in this area includes analysis and development of type systems, design of advanced programming languages, and composition of systems in distributed environments.

2. **Safety Infrastructure.** The development of foundations and software support for exploiting Proof-Carrying Code. Work in this area includes basic research into logical frameworks (for proof representation and optimization), abstract machine design, and development of verification-condition generators.

3. **Compiler Technology.** The development of techniques and tools for compiling high-level programs into efficient, certifiable code. Work in this area includes the implementation of type-directed and certifying compilers, for exploitation of type structure and automatic generation of Proof-Carrying Code.

4. **Applications.** Collaboration with other researchers in the Active Networks program (and related programs) to incorporate and demonstrate our language-based tools and technologies. Besides Active Networks, we believe that there are also relevant applications in DARPA's Quorum, Information Survivability, and EDCS programs.

More details on the tasks and activities in each of these four areas is given in the following subsections. It is important to note that these four areas are far from distinct, however. Indeed, the basic strength of the project comes from the synergy that arises from taking a "vertically integrated" view of the software problems posed by Active Networks. For example, the development of the type theory for modular systems has direct application in type-directed compilation, which in turn leads to techniques for extending certifying compilation and thereby improving the utility of proof-carrying code. Our goal is to approach both the foundational and the practical problems in Active Networks with equal weight, so that the theory can be best informed by the practice and that novel practical solutions can be derived from the application of theoretical principles.

## 3.1 Language Technology

Our work on language technology is concerned with the foundations of modular software development. The premise of our work is that fundamental research on type systems is required to ensure the integrity of dynamically extensible software systems. As discussed above, type theory provides a comprehensive framework for the design of mechanisms for expressing and checking the properties of program components, including constraints on the interaction between components. Type theory is also fundamental to our work on proof-carrying code and certified compilation; see the discussion under Compiler Technology and Safety Infrastructure for more details of the use of types in these contexts.

Our proposed work on language technology may be divided into two broad categories, each concerned with the development of type systems for programming lanuages:

1. *Enrichment.* The goal is to allow *more programs* to be expressed naturally within a typed language, in particular those using advanced modular and object-oriented programming techniques, including subtype polymorphism, classes and objects, higher-order modules, and concurrency. The focus of this research thread is the design of ML2000, a proposed successor language to Standard ML.

2. *Refinement.* The goal is to allow *more program properties* to be expressed naturally within a type system. The focus of this research thread are *refinement types* (for recursive data structures) and *dependent types* (for aggregate data structures), both of which are considered in the context of Standard ML.

Standard ML [14] (and its close relative, Caml [12]) represent a high-water mark in the design of flexible, expressive type systems for programming languages. Specific features include polymorphic type inference, a flexible and expressive module and interface language, and support for functional and imperative programming. These features have been effectively deployed in the FoxNet implementation of TCP/IP [3] and in the Ensemble network protocol suite [27]. In the FoxNet a network protocol layer is a functor that is parameterized on the layer beneath it and that implements a generic protocol interface. This design supports the flexible composition of protocol layers and the rapid development of non-standard network protocols (also exemplified in Ensemble). The FoxNet implementation uses higher-order functions to implement "upcalls" [5] or "paths" [10] through a protocol stack, a key to achieving efficient protocol processing in a layered implementation.

The focus of our work on enriched type systems is ML2000, a proposed successor language to Standard ML.[1] The goal of the ML2000 design is to integrate the following features into a unified language extending Standard ML:

1. *Subtyping.* Subtyping is a very powerful and convenient mechanism for achieving code re-use — a value of a subtype may be provided whenever a value of the supertype is required. This is especially important in the context of object-oriented programming, as exemplified by Java.

2. *Objects.* Many programming problems (especially those with intensive interaction requirements) are naturally structured using objects and methods. ML provides only rudimentary support for object-oriented programming.

3. *Classes and inheritance.* Some systems are naturally constructed by accretion of additional mechanism or modification of existing mech-

[1]The design of ML2000 is a joint undertaking with Andrew Appel (Princeton), Luca Cardelli (Microsoft), Carl Gunter (UPenn), Xavier Leroy (INRIA), Dave MacQueen (Bell Labs), John Mitchell (Stanford), John Reppy (Bell Labs), and Jon Riecke (Bell Labs).

anisms through inheritance. ML provides only weak forms of inheritance through the module language.

4. *Concurrency.* Concurrent programming mechanisms are fundamental to modern software development. CML, a dialect of Standard ML supporting concurrency, explored the extension of ML with first-class events, a convenient notation for managing concurrent interaction. Standard ML lacks such support.

Our work on refinement of type systems is based on the idea that types may be viewed as machine-checkable program properties. Under this view, the type system of ML allows only very elementary properties to be expressed. For example, all functions from integers to integers have the same type. This means that many properties cannot be verified statically, requiring either trust in the correctness of code or run-time checking, depending on the application. Thus we have been conducting research to *refine* the type system of ML to allow more program properties to be expressed and checked. We propose to continue this research with a focus on *refinement types* and *dependent types.*

1. Refinement Types. Using initial prototypes developed by the Fox project [8, 6] we propose to investigate how recursive data structure invariants can be checked effectively and how compilers can take advantage of this information to generate more efficient code.

2. Dependent Types. While dependent types in their full generality do not admit practical type-checking algorithms, some common special cases such as array bound checking have been shown to be feasible [31] which allows significantly more efficient proof-carrying code [18]. We propose to further develop and eventually integrate dependent types into the Fox project's TIL compiler [24].

## 3.2 Safety Infrastructure

Our work on safety infrastructure is concerned with both the theoretical foundations and the software necessary to exploit the concept of Proof-Carrying Code (PCC) in Active Networks and other extensible systems. PCC enables a computer system to determine, automatically and with certainty, that program code provided by another system is safe to install and execute. The code may be written in virtually any language that can be given a precise operational semantics. In our previous work, we have developed PCC for DEC Alpha assembly code and demonstrated its utility in safe operating system extensions [17, 16].

The key idea behind PCC is that the external system, which we shall henceforth refer to as the *code producer*, provides an encoding of a formal proof that the code adheres to a *safety policy* defined by the recipient of the code, which we shall call the *code consumer*. The proof is encoded in a form that can be transmitted digitally to the consumer and then quickly validated using a simple, automatic, and reliable proof-checking process.

Safety policies are promulgated by code consumers in an extensible system (*e.g.*, an active network router). The safety policy specifies a sufficient condition for the code consumer to accept a program as a dynamic extension (*e.g.*, an active packet). It is the job of the code consumer to formulate a suitable safety policy to ensure its own integrity. Code producers must comply with the safety policy for the code to be accepted as a dynamic extension.

PCC is directly relevant to Active Networks, and is potentially useful in many other applications as well. It enhances the ability of a collection of software systems to interact flexibly and efficiently by providing the capability to share executable code safely. Besides the nodes in an Active Network, other typical examples of code consumers might include extensible operating system kernels and World-Wide Web browsers, both of which must allow untrusted applications to install and execute code. Indeed, PCC is useful in any situation where the safety in the presence of newly installed code is paramount.

PCC has several key characteristics that, in combination, give it an advantage over previous approaches to safe execution of foreign code:

1. *PCC is general.* The code consumer defines the safety policy, and this policy is not limited to a particular notion of "safety." We have already experimented both with simple safety properties, such as memory and type safety, and with properties that are normally difficult to verify, such as time limits on execution [20].

2. *PCC is low-risk and automatic.* The proof-checking process used by the code consumer to determine code safety is completely automatic and can be implemented by a proof-checking program that is relatively simple and easy to trust. Thus, the *safety-critical infrastructure* that the code consumer must rely upon is reduced to a minimum [19].

3. *PCC is efficient.* In practice, the proof-checking process runs quickly.

Furthermore, in contrast to previous approaches, the code consumer does not modify the code in order to insert costly run-time safety checks, nor does the consumer perform any other checking or interpretation once the proof itself has been validated and the code installed.

4. *PCC does not require trust relationships.* The consumer does not need to trust the producer. In other words, the consumer does not have to know the identity of the producer, nor does it have to know anything about the process by which the code was produced. All of the information needed for determining the safety of the code is included in the code and its proof.

5. *PCC is flexible.* The proof checker does not require that a particular programming language be used. PCC can, in principle, be used for a wide range of languages, ranging from high-level languages such as ML [15], to abstract machine codes, and even to machine languages.

An important design criterion for an implementation of certified code is that the proof checker be simple, concise, and, preferably, universal. The reason is obvious: since the checker is the arbiter of compliance with the safety policy, a faulty checker could accept a non-compliant binary, compromising the integrity of the system as a whole. The checker must therefore be part of the trusted computing base, and should be as simple and concise as possible to maximize confidence in its correctness. Universality—the ability to use a single checker for many different safety policies—is desirable, since then the code for the checker can be re-used, but not essential for this goal. (Universality might be regarded as a disadvantage, for if the checker is faulty, then all code consumers based on it are insecure. On the other hand a universal checker would be scrutinized carefully by each party, thereby increasing our confidence in its correctness.)

In our previous work on PCC, we have shown that the Edinburgh Logical Framework (LF) [9] can be an effective language for encoding proofs. There are several advantages to the use of LF for proof encodings. First, LF is an extremely simple language but still highly expressive. Furthermore, it can be used to encode proofs in a wide range of logics, and in fact the logics themselves can be specified in LF and checked for various kinds of consistency. Second, LF is itself a typed language, and in fact the process of proof checking in LF can be reduced simply to type checking in LF. In other words, the proof checker can be simply a type checker for LF. This connection between proof checking and type checking is critical in proving the soundness of the entire approach, as well as simplifying the trusted computing base. It also opens up many possibilities for further improvements, as much of the work on language design and type theory becomes directly applicable to proof encoding, proof optimization, and proof checking.

By exploiting these features of LF, we have already been able to develop a highly robust, universal proof checker that we believe can be readily incorporated into many Active Network systems. However, there are still several open research problems to be solved before a truly practical PCC infrastructure can be realized.

1. The first research problem concerns the size of the encoded proofs. While we have found that the time required for proof checking is reasonably low, the size of the proofs ranges from one to ten times larger than the code, with typical sizes on the order two to three times larger. While these sizes are not necessarily a critical problem for many applications, they are a clear source of latency if included in every capsule. To date, we have given relatively little thought to the matter of proof size, and so we propose to concentrate effort on the problem of proof optimization. Simple optimizations techniques (such as "common subproof elimination") will be relatively easy to apply. However, more fundamental restructuring of the proof representations may be needed in order to bring the proof sizes down to less than the size of the code, without increasing proof-checking time. To this end, we propose to develop a suite of proof-optimization techniques, including optimizations on encoded proofs as well as improved methods for eliding parts of proofs (and then quickly reconstructing them during proof checking). Especially for complex safety policies which may be required in some Active Networks, more drastic extension to the framework itself might be required, such as provided by the *linear logical framework* (LLF) [4]. A linear framework has more succinct means for expressing mutable state and concurrency than traditional frameworks, and we plan to investigate efficient proof-checking techniques for LLF.

2. A second problem concerns the construction and maintenance of *verification-condition generators*, or VCGens. In a PCC system, there are two components in the trusted computing base, the proof checker and the VCGen. The VCGen program makes a single scan over the program code and extracts a logical predicate whose satisfiability guarantees the code's safety. In a sense, the VCGen is a compiler that translates program code into logic. The proof checker then checks that the proof attached to the code is indeed a valid proof of the predicate. Since the proof checker can concern itself with proofs written in

a pure logic, it is a generic component. On the other hand, a VCGen must be constructed for each language or machine architecture, as well as for each definition of safety.

Fortunately, the VCGen is not a complicated program. Still, there is inherently a greater risk of errors appearing in this component, and hence the matter of its correctness is of greater concern. We propose to investigate techniques for proving the correctness of VCGen programs as well as deriving their implementation from high-level specifications. We plan to once again exploit the expressive power of logical frameworks to describe the operational semantics, logical verification conditions, and their relationship within the same language and apply automated proof tools [22] to verify critical properties of a VCGen. This will also require further research into the efficient implementation of the Twelf meta-theorem prover sketched in [22]. A specific preliminary task along these lines will be to port our existing VCGen component to the Intel x86 architecture (the current VCGen is for the DEC Alpha), both to provide immediate support to other developers who are using this architecture, as well as to gain valuable initial experience in modifying the VCGen.

3. Certifying machine code has the great advantage of efficiency, but restricts code mobility since it applies to only one architecture. We therefore also propose to adapt the idea of PCC to virtual machine code. Proofs attached to virtual machine code can potentially simplify bytecode verification [23] and interpretation, enable multiple safety policies, and allow more efficient just-in-time compilation without sacrificing safety or relying on an unreasonably large trusted computing base.

## 3.3 Compiler Technology

A critical element of our proposed research is the construction of demonstration compilers embodying ideas from our work on language technology and security infrastructure. The construction of working compilers is crucial for validating our work on the principles of safe system extension and for applying these ideas to Active Networks or other extensible systems. It also provides concrete tools that may be distributed to other researchers and developers.

Our work on compiler technology is based on the general notion of *certified code*, of which Proof-Carrying Code is a specific instance. A certified program is a program (written in some language) equipped with a *safety certificate* (in the case of PCC, a formal proof) asserting that the program complies with a pre-determined safety policy. Crucially, not only must the code producer comply with the policy, but it must also produce a machine-checkable representation of the certificate. This certificate is checked by the code consumer (by using the safety infrastructure described above) as a condition of acceptance of the extension. (In the context of Active Networks, a capsule, or active packet, would be required to contain a certificate of compliance with an active router's safety policy.)

One advantage of certified code is that the code consumer need not rely on any assumptions about the origin of the component — there is no need for any form of authentication, nor any reliance on the integrity of a given compiler. This rules out "trojan horse" attacks based on spoofing a compiler and avoids the need for complicated authentication protocols. Another advantage is that there is no fundamental limitation on the efficiency of the generated code — if a safety certificate can be produced, any code, whether a machine binary, byte code for an abstract machine, or abstract syntax for a high-level language, is acceptable. The only limitation is the feasibility of producing safety certificates for programs of practical interest, and the checkability of proofs of those certificates.

It is important to emphasize that compliance with a safety policy does not in itself ensure the integrity of an extensible system for the simple reason that the policy might not in fact be sufficient to ensure that the system remains well-behaved under extension. This limitation is fundamental to any approach to extensibility—the code consumer may enforce limitations that are not enough to avoid compromise of functionality. We must be satisfied with the observation that the safety policy is a contractual arrangement between the code producer and code consumer whose terms are determined by the consumer itself.

The purpose of a *certifying compiler* is to generate certified code from a high-level language relative to a given safety policy. We are experimenting with two approaches to building a certifying compiler:

1. Typed Intermediate Languages and Type-Directed Compilation. A type-directed compiler produces abstract machine code in an intrinsically safe statically typed language. Type safety of the abstract machine code is sufficient for a range of safety properties — including memory safety and adherence to specified API's — that are often sufficient for acceptance by a code recipient. The abstract machine code could be as high level as ML or as low level as the Java Virtual

Machine, equipped with a suitable type system (for example, Abadi and Stata's type system for JVM [23]).

2. Proof-Carrying Code and Certifying Compilers. In the interest of efficiency the code generated by a compiler need not be intrinsically safe (*i.e.*, there might be unsafe instruction sequences), so the compiler must augment the binary with a safety certificate, in the form of a proof, asserting that the target program is safe. As explained in the previous section, the safety proof, if valid, guarantees compliance with a safety policy, and is checkable by the code recipient (given also the binary to be executed and the safety policy itself). From the user's point of view, the compiler is a tool for automatic generation of optimized PCC binaries.

More specifically, we propose the following work on these two approaches to certifying compilers:

1. The basis for our experiments with Typed Intermediate Languages and Type-Directed Compilation is the TILT compiler for Standard ML. TILT is an evolution of the TIL compiler [24] to encompass the complete Standard ML language, including the module system. TILT is based on the idea of *type-directed translation*, a generalization of syntax-directed translation in which not only is a program transformed during compilation, but so too is it type in such a way that the translated program has the translated type. Thus the TILT compiler is based on the idea of *typed intermediate languages*, which are statically-typed programming languages that serve as the target languages for compiler transformations. Type-directed translation ensures that the compiler is able to track the typing properties of the translated code through the compiler. At any stage the generated code may be checked against its type to ensure integrity. Moreover, this code may be used as a distribution format in an extensible system — the type is a safety certificate that may be checked by a type checker for that intermediate language.

   We propose to build the TILT compiler for Standard ML to validate the feasibility of type-directed compilation for a full-scale programming language. This work includes the design of typed intermediate languages, the development of type checkers for them (a surprisingly difficult and subtle task), the development of a compiler and run-time system, and comparison of its performance with other compilers for

Standard ML. The TILT compiler will be based on the TIL compiler developed by the Fox Project for the "core" language of Standard ML (without modules). We also plan to extend the TILT compiler with support for refinement types and dependent types, as outlined in the Language Technology section of this proposal.

2. For Proof-Carrying Code and Certifying Compilers, we plan to develop an optimizing compiler for a safe C-based programming language suitable for some kinds of systems-level (especially Active Network) programming. This compiler development will start from our current early prototype system called Touchstone. The Touchstone compiler translates a small type-safe C-based language into highly optimized DEC Alpha assembly code. The assembly code is annotated in such a way that a theorem prover (a prototype of which we have built as part of the PCC infrastructure) is always able to find a proof of the type safety of the code. Since our theorem prover generates checkable encodings of its proofs, it is then straightforward to generate PCC binaries for the target programs produced by the compiler. Our early experiments with the Touchstone prototype have been extremely encouraging, with excellent results shown for several realistic C programs [18].

   We propose to continue the development of Touchstone, with the goal of eventually making it robust enough to distribute to other researchers. There are several aspects to this task. First, an appropriate "safe C-like language" has to be designed and formally specified. Our current prototype implements only a fragment of C, with many important features missing, such as structs. Second, although our current prototype performs a number of code optimizations, it does this without the benefit of any global dataflow analysis and hence is inherently limited in what it can do. Exactly how dataflow optimizations will interact with the generation of proofs is uncertain at this point, but we believe that it should not pose any fundamental difficulties. Third, there is the matter of the run time system, and in particular, garbage collection. Our current Touchstone prototype does not have any kind of support for automatic garbage collection, and hence in this sense it is not really a practical system. Arranging for the compiled code to invoke a garbage collector at the appropriate times again should not pose any serious problems, though as we pointed out in our earlier work on PCC extensions of ML programs [15], some care must be taken in the case that copying garbage collection is used. Finally, as progress in type systems and safety infrastructure are made, particularly in the area of specification of resource constraints, we plan to incorporate these extensions in the certifying compiler, so as to be able to experiment with them in the context of Active Networks and other extensible systems.

## 3.4 Applications

A major part of the success of our research is derived from experiences gained in the application of principles from language design, safety infrastructure, and compiler design. As we explained earlier, the basic strength of the project comes from the synergy that arises from taking a "vertically integrated" view of systems problems, such as those posed by Active Networks. Our goal is to approach both the foundational and the practical problems with equal weight, so that the theory can be best informed by the practice and that novel practical solutions can be derived from the application of theoretical principles. Active Networks promise to be an especially rich source of relevant practical problems, since the requirements for modularity, safety, and efficiency are so severe. We believe that the language technologies that we are proposing to investigate will be absolutely critical for successful development of Active Networks.

Our approach to applications will be to build tools, including new language designs, compilers, and basic infrastructure components, that can be distributed to other researchers for incorporation into their Active Networking systems. We plan the following activities:

1. We plan to develop robust prototypes of the TILT and Touchstone certifying compilers and make them available to other researchers. The TILT compiler will provide a high-performance alternative to existing ML compilers, as well as a valuable platform for further experimentation in type-directed compilation and typed intermediate languages. The Touchstone compiler will provide an easy way for users to generate PCC binaries, for use in their own experiments with PCC.

2. In conjunction with the University of Utah, we propose to develop PCC components that can be incorporated into the Flux OSKit. This will involve the integration of the PCC proof checker into the OSKit as an additional system service, as well as the development of an appropriate VCGen module. Once integrated, this will provide users of the OSKit to use native code in capsules, for performance-critical applications, as well as applications that require more detailed safety policies than

can be easily provided by a Java Virtual Machine. It will also allow Utah's proposed Janos system to be extended with native code, again for performance-critical and safety-critical applications.

3. In addition to the incorporation of PCC into the OSKit, we also propose to extend the Kaffe JVM system with PCC. This will provide users of the Kaffe Java infrastructure with a safe way to extend Java applications with highly optimized native code. Since several research groups are using Kaffe (including the Utah group as well as the CMU Darwin project), this will provide yet another way for other researchers to experiment with PCC and other advanced language technologies.

The principles and technologies that we will develop in the areas of language design, safety infrastructure, and compiler technology will have direct application in any system that depends on safety, modularity, and efficiency. This includes Active Network, but also systems being developed under the DARPA/ITO Quorum, Information Survivability, and EDCS programs.

## 4 Related Work

**Active Networks.** The idea of an Active Network was proposed by Wetherall, Guttag, and Tennenhouse [30, 25, 29] as a means of improving the flexibility and performance of network protocols. The idea of Active Networks raises significant problems that are addressed in this proposal. As outlined above, we propose to focus on the fundamental problem of how to achieve safe, reliable, and efficient integration of "foreign" code into a safety and performance-critical software system such as a network router. Ongoing research in Active Networks will draw on and inspire our research on the underpinnings of safe composition of code from untrusted components.

**PLAN.** The Switchware Project at the University of Pennsylvania is developing an Active Network in conjunction with Bellcore [1]. A critical part of their work is the development of a programming language, called PLAN, for programming Active Networks [11]. Their work builds on ours in a number of respects, including the reliance on a typed, functional language based on ML as the basis for their design, and the use of a variant of our Proof-Carrying Code technology to provide assurances about the run-time behavior of active network software. Their efforts are complementary to ours in that they are working actively with industrial partners to transfer fundamental language technology to the development of a “real world” active network system. We expect to interact closely with the UPenn group in the course of our research.

**Joust.** John Hartman, Larry Peterson, *et. al.* [10] at the University of Arizona have coined the term *liquid software* for a general approach to achieving flexible and reliable code mobility in a network environment, with application to the construction of Active Networks and, more broadly, to the general problem of dynamic integration of mobile code. Their approach is philosophically and technically similar to ours in that they emphasize the importance of *modularity*, including the critical notion of *enforced abstraction boundaries*, for achieving safe integration of components. However, they rely on dynamic checking techniques, rather than static analysis, to achieve these ends. In particular they employ a variant of the Java Virtual Machine [13] (extended with some Joust-specific constructs, and omitting some of the JVM functionality) to ensure safety of mobile code, and employ a “just in time” compiler to enhance performance. But, as we discussed earlier, this increases very substantially the trusted code base on which their approach must rely to guarantee integrity. Moreover, their reliance on C as an implementation vehicle complicates, or even precludes, the kind of static analysis that we maintain is essential for achieving efficiency without sacrificing safety.

**Ensemble.** Ken Birman and Robbert van Rennesse at Cornell have demonstrated convincingly that advanced programming languages with strong type systems, rich modularity mechanisms, and higher-order functions can be deployed to achieve an unprecedented degree of flexibility, reliability, and efficiency in the construction of networking software [27]. Moreover, efficiency is achieved, in part, by using theorem-proving techniques to carry out the verified transformation of well-structured programs into more efficient, but functionally equivalent, versions of these protocols. Their work is inspired by our earlier work on the Fox Project, in which we demonstrated the use of ML to achieve efficient, composable implementations of network protocol stacks.

**Flux.** We have an ongoing collaboration with Jay Lepreau at the University of Utah who is developing the Flux Operating System Toolkit [7] to provide the infrastructure for experimentation with advanced system structuring techniques. We are at present engaged in a cooperative effort to extend the Kaffe implementation of the JVM and the Flux Toolkit to include support for safe extension based on proof-carry code. This ties in with the work of Peterson, *et. al.*as well as CMU's own Darwin group on the use of Kaffe, and provides a further point of contact with the PLAN work at UPenn.

## 5 Key Personnel

**Robert Harper** is an Associate Professor of Computer Science at Carnegie Mellon University. He is well-known for his work on the design and semantics of Standard ML, for introducing (together with Honsell and Plotkin) the idea of a logical framework for representing proofs in formal systems, and for exploring the use of typed intermediate languages in a compiler.

**Peter Lee** is an Associate Professor of Computer Science at Carnegie Mellon University. Since 1991, he has been leading the *Fox Project* with Robert Harper. He has made significant research contributions in many areas related to the implementation and use of advanced programming languages, especially semantics-based analysis and optimization techniques for languages such as Standard ML. Most recently, he has been involved in the development of Proof-Carrying Code and its application to extensible operating systems.

**Frank Pfenning** is a Senior Research Computer Scientist at Carnegie Mellon University. His research focus is on the development of advanced type systems for programming languages and mechanized reasoning about properties of programming languages in logical frameworks. Among his accomplishments, he designed and implemented the Elf language which was used in the first prototype implementation of Proof-Carrying Code as well as many other logic-based systems.

## References

[1] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active bridging. In *Proceedings of the ACM SIGCOMM'97 Conference*, Cannes, France, September 1997.

[2] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance

in the SPIN operating system. In *Symposium on Operating System Principles*, pages 267–284, December 1995.

[3] Edoardo Biagioni, Nicholas Haines, Robert Harper, Peter Lee, and Brian G. Milnes. Standard ML signatures for a protocol stack. Fox Memorandum CMU–CS–93–170, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1993. (Also published as CMU–CS–FOX–93–01).

[4] Iliano Cervesato and Frank Pfenning. A linear logical framework. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 264–275, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[5] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180. ACM, December 1985.

[6] Rowan Davies and Frank Pfenning. Practical refinement-type checking. Draft paper, July 1997.

[7] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for OS and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint-Malo, France, October 1997.

[8] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, March 1994. Available as Technical Report CMU-CS-94-110.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Symposium on Logic in Computer Science*, pages 194–204, Ithaca, New York, June 1987.

[10] John H. Hartman, Larry L. Peterson, Andy Bavier, Peter A. Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd A. Proebsting, and Oliver Spatscheck. Joust: A platform for communcation-oriented liquid software. Technical Report TR 97-16, Department of Computer Science, The University of Arizona, December 1997.

[11] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. Submitted, November 1997.

[12] Xavier Leroy. The Objective Caml system: Documentation and user's guide. Available at `http://pauillac.inria.fr/ocaml/htmlman/`., 1996.

[13] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[14] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[15] George C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris*, January 1997.

[16] George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, September 1996.

[17] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96), Seattle*, pages 229–243, October 1996.

[18] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. Submitted to the ACM SIGPLAN Conference on Programming Language Design and Implementation, November 1997.

[19] George C. Necula and Peter Lee. Efficient representation and validation of proofs. Submitted to the ACM Symposium on Logic in Computer Science, December 1997.

[20] George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. Submitted to the Special Volume on Mobile Agents, Lecture Notes in Computer Science, December 1997.

[21] Michael P. Plezbert and Ron K. Cytron. Is "just in time" = "better late than never"? In *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 120–131, Paris, January 1997.

[22] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for lf. Submitted, January 1998.

[23] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proceedings of the 25th Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Diego, CA, January 1998. To appear.

[24] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, May 1996.

[25] David L. Tennenhouse and David J. Wetherall. Towards and active network architecture. *Computer Communication Review*, 26(2), April 1996.

[26] USENIX Association. *The BSD Packet Filter: A New Architecture for User-Level Packet Capture*, January 1993.

[27] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptive systems using ensemble. Technical Report TR97-1638, Department of Computer Science, Cornell University, July 1997.

[28] R. Wahbe, S. Lucco, T. .E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating System Principles*, pages 203–216, December 1993.

[29] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In Aurel A. Lazar, editor, *Proceedings of the First IEEE Conference on Open Architectures and Network Programming (OPENARCH'98)*, San Francisco, CA, April 1998. To appear.

[30] David J. Wetherall and David L. Tennenhouse. The ACTIVE IP option. In *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[31] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. Submitted, November 1997.