Type Refinements

Robert Harper and Frank Pfenning
Project Proposal
NSF Grant CCR-0204248

November 2001

Project Description

1 Background and Motivation

Despite many concentrated research efforts in various areas such as software engineering, programming languages, and logic, software today is not fundamentally more reliable than it was a decade ago. Software is becoming increasingly complex and inter-reliant and the techniques and tools provided by the academic community are used only sparsely. In part, this can be attributed to the many barriers to technology transfer. However, one can also recognize that in a number of ways the methods provided by the research community fail to be applicable to the problems faced by developers or maintainers of large-scale, long-lived systems.

One important aspect of software development and maintenance is to understand properties of a complete system, its individual components, and how they interact. There is a wide range of properties of interest, some concerned only with the input/output behavior of functions, others concerned with concurrency or real-time requirements of processes. Upon examining the techniques for formally specifying, understanding, and verifying program behavior available today, one notices that they are almost bi-polar. On the one extreme we find work on proving the correctness of programs, on the other we find type systems for programming languages. Both of these have clear shortcomings: program proving is very expensive, time-consuming, and often infeasible, while present type systems support only minimal consistency properties of programs.

The proposed research is intended to help bridge this gap by designing and implementing more refined type systems that allow rich classes of program properties to be expressed, yet still be automatically verified. Through careful, logically motivated design we hope to combine the best ideas from abstract interpretation, automated program analysis, type theory, and verification. In the remainder of this section we explain and justify our approach in somewhat more detail, before giving a research plan in the next section.

Types and Complete Specifications. Complete specifications of a program's behavior are generally not feasible for complex software systems. For some smaller programs or components where specification may be possible, the effort required to formally prove adherence to the specification can be tremendous. Finally, even if both specification and proof are undertaken for a given module, it is exceedingly burdensome to maintain such a proof as the

program evolves in response to changing requirements. A combination of these factors means that complete specification and verification are rarely undertaken in practice.

Type systems as they exist in current programming languages, on the other hand, have none of these drawbacks. Types are generally small compared to a program, types can be checked automatically by a compiler, and types evolve easily together with the program. However, there remain two principal difficulties with types as a software engineering tool: only a limited number of program properties can be expressed, and depending on the underlying language, types may or may not actually provide a guarantee for the absence of certain kinds of errors.

We therefore propose to follow the basic design principles underlying type systems, increasing their expressive power without sacrificing their desirable qualities: practical, automatic verification and maintainability throughout the software life cycle.

Types and Program Analysis. Program analysis tools have been developed for two major purposes: for improving the reliability of software by providing help in identifying bugs, and for improving program efficiency through safe optimizations. Generally, program analysis tools are designed to be automatic and some of the more recent ones scale to large systems. However, program analysis tools are inherently handicapped because they must deal with the bare software artifact as it exists, without the benefit of the programmer's knowledge about the program's behavior or invariants. This is particularly detrimental at module boundaries, especially when those boundaries provide interfaces to external modules written by others and may thus not be available for analysis.

Type systems on the other hand are designed to work well at module boundaries. Properties of implementations can be checked against an interface, and external code has license to assume such properties. Moreover, these properties reflect the programmer's understanding of the code's behavior in a formal way.

We therefore propose to adapt some of the algorithms used in program analysis to explicit type systems in order to combine the benefits of both approaches. This idea is not entirely new (see, for example, [NS95, PP98]), but generally the type systems that have been proposed as specifications for analysis algorithms were biased towards inference. In contrast, our emphasis is on checking implementations against type specifications.

Types and Extended Static Checking. Recently, extended static checking [Lei01] has been proposed as a method for program property verification that is integrated into the development process. The goal of extended static checking is to use machine assistance to verify relatively simple, yet informative, properties of programs. For example, an extended static checker for Java can check a programmer's assertion that an object is non-null at a designated program point. If the verification succeeds, the compiler may suppress run-time null checks on this object. Should the verification fail, a warning is issued, but the program nevertheless compiles and executes, albeit with run-time null checks included.

Extended static checking is a useful tool for stating and verifying simple properties of programs. Although in most cases the verification is sound, in the sense of affirming only correct operational behavior, in some instances soundness is sacrificed in favor of a more efficient checker [DLNS98]. In our view this is an unfortunate compromise since it forces the programmer to consider the validity of error messages, and encourages a natural tendency to ignore a warning in the belief that the checker is, once again, incorrect.

Our proposed research can be seen as a form of extended static checking. However, we insist that our type system be a sound approximation to actual program properties. This

may impose a somewhat greater burden on the programmer in the form of more verbose type information. By casting extended static checking as a type system, however, we hope to attain increased portability across implementations and predictability to the programmer.

Our goals are similar to those of soft typing [CF91], but our technical approach is rather different. As with soft typing, we insist on a sound logic, and we require that the programmer state assertions that are to be verified. In contrast to soft typing, however, we do not start with an untyped (better: unityped) language, but rather refine a static type system to capture more precise properties. Our work is therefore not an alternative to static typing, but rather complements and enriches it.

Type Refinements. It is our observation that, by and large, present type systems are general enough to allow practical program development with types. However, when viewed as a system for program properties, they are too limited in their expressive power. The emphasis of our proposed work is therefore on refining known type systems in order to reflect more program properties—and thereby expose more bugs—rather than extending them to admit more programs.

We think of a *type refinement* as a decomposition of the values of a given type into subsets. Such subsets cannot be arbitrary if we want to maintain decidability and practicality of type checking. Instead, each system of refinement imposes a restriction on the subsets that can be formed and analyzed. In each case, this raises several problems that determine the viability of the type system:

1. How verbose are the type specifications the programmer must supply?
2. How efficient are the algorithms for checking the refinement properties?
3. How accurately does the type system track the properties the program actually has?
4. How do the refinement properties interact with other advanced constructs of a language such as higher-order functions, objects, polymorphism, or effects?

An Example: Red/Black Trees. As an example for type refinements throughout this proposal we use red/black trees [Oka99]. They are binary trees with red and black nodes satisfying three invariants: (1) a *color invariant* which states that the children of a red node must always be black (where leaves are black), (2) a *balance invariant* which states the number of black nodes on each path from the root to all leaves is equal, and (3) a *order invariant* which states that the data entries in the tree must be ordered from left to right. With present technology we can easily state and enforce the color and balance invariant for implementations. Combining them is possible, yet inelegant at present. The order invariant cannot be presently enforced.

We show here only the types, constructors, and a few declarations. We use the following pseudo-code notation to concentrate on the essential issues.

`datatype` t	introduces a new data type t
`con` $c : \tau$	declares the type of a data constructor c
`val` $f : \tau$	declares the type of a function f
`refine` $p <: t$	introduces a new refinement p of t
`prop` $d : \rho$	declares the property ρ of constructor or function d

Actual declarations would be slightly different, but convey essentially the same information. Type constructors are written in postfix notation as in ML, where α denotes a type variable.

For our example, we assume a type τ `entry`, whose elements consists of a key and a data element of type τ. Then the underlying types for red/black trees and their constructors are:

```
datatype color
con red : color
con black : color
datatype α tree
con leaf : α tree
con node : color * α tree * α entry * α tree -> α tree
```

Prior Work on Type Refinements. Through our prior work we have obtained significant evidence that type refinements can be expressive, practical, and applicable in a wide variety of languages.

The first of these are the so-called *refinement types* [FP91, Fre94]. In order to avoid confusion in this proposal we will refer to this narrower class of refinements as *datasort refinements*. A datasort specification names subsets of a data type that can be recognized by a finite tree automaton, that is, regular tree languages. Questions such as membership of values, subtyping, or intersections between languages can then be computed by standard algorithms on regular tree languages. The datasort refinements are extended through the (ordinary) type hierarchy by using intersection types. Type inference is then decidable via abstract interpretation over a programmer-specified lattice of datasort refinements. The implementation of refinements types for ML has shown good practical efficiency [Dav97].

Recently, we have simplified this type system in two respects. First, in order to guarantee soundness in the presence of effects we eliminated the law of distributivity for subtyping and introduced a form of the so-called value restriction [DP00a]. Second, in order to work well at module boundaries we have required more programmer annotations in a system of bi-directional type checking, simultaneously making the system more efficient and modular, at a small cost in additional type annotations. Datasort refinements have been applied to functional languages and logical frameworks [Pfe93].

As a first example, we consider two singleton refinements of the type `color`. The two constructors have the obvious properties.

```
refine red <: color
refine black <: color
prop red : red
prop black : black
```

As a second example, we can now state the color invariant for red/black trees. We need two properties, α `rbt` which are valid red/black trees (visible externally) and α `bt` which are trees with a black root (not exported at the module boundary). Note the use of conjunction `&` to associate more than one property with a constructor.

```
refine α bt  <: α tree  (* valid tree, black root *)
refine α rbt <: α tree  (* valid tree *)
prop leaf : α bt
prop node : black * α rbt * α entry * α rbt -> α bt
          & red * α bt * α entry * α bt -> α rbt
```

Now we can state and enforce properties of the constant `empty` and functions `insert` and `lookup` via refinement checking.

```
prop empty  : α rbt
prop insert : α rbt * α entry -> α rbt
prop lookup : α rbt * α entry -> α option
```

Note that it is illegal to apply `insert` to a tree that is not a valid red/black tree. Internally to the implementation of `insert`, further properties of `insert` are required in order to check the property given above. This is because `insert` locally violates the invariant and then restores it. To illustrate this point, here are the refinements required internally, where α `rt` is a valid tree with a red root and α `bad` is a tree where the invariant may be violated at a red root when one of the children could also be red.

```
refine α rt <: α tree      (* red root *)
prop node : red * α bt * α entry * α bt -> α rt
refine α bad <: α tree     (* possibly red/red at root *)
prop leaf : α bad
prop node : black * α rbt * α entry * α rbt -> α bad
          & red * α bt * α entry * α bt -> α bad
          & red * α rt * α entry * α bt -> α bad
          & red * α bt * α entry * α rt -> α bad
```

The second major approach has been the design of dependent type systems over decidable index domains [Xi98]. Instead of finite lattices and algorithms on tree automata, type-checking here requires decision procedures for the index domain, such as Presburger arithmetic. The major applications have been for static array bounds checking [XP98] and similar verification tasks. Again the type system is robust in the sense that it applies to several languages. Practical dependent type systems for functional [XP99], imperative [Xi00], object-oriented [XX99] and even assembly language [XH01] have by now been developed.

To illustrate this, we can refine red/black trees further by their black height, that is, the number of black interior nodes on each path from a leaf to the root.

```
refine α bt(n) <: α tree
refine α rbt(n) <: α tree
prop leaf : α bt(0)
prop node : black * α rbt(n) * α entry * α rbt(n) -> α bt(n+1)
          & red * α rbt(n) * α entry * α rbt(n) -> α rbt(n)
```

Note that the type of `node` enforces that both subtrees have the same black height n. Externally, the type of `insert` and `lookup` would not change, internally we need an existential type to capture that `insert` may or may not increase the black height of a tree.

With the present dependent type system, the example above has to be written in a signficantly less transparent manner, coding the color information by several integers, because we do not have datasort and dependent refinements in the same system. One of the important points of proposed work will be to combine datasort and dependent refinements without sacrificing decidability or practicality.

Note also that present refinement systems are too weak to capture the order invariant on elements stored in the tree. This would require an index domain of transitive relations and an appropriate decision procedure. One of the items of proposed work will be to devise a more flexible mechanism to describe and implement index domains so that invariants such as the order invariant can be expressed and checked.

Summary. We have argued above that type refinements offer a practical way to specify and check important program properties beyond traditional type systems. By design, they are conservative over existing languages so that present programs can still check and run as before. We have practical implementations of type refinement systems that exploit algorithms such as abstract interpretation or decision procedures for linear equalities and inequalities over integers. Type refinements provide a formally specified language to state assumptions and check compliance at module boundaries. They are robust in that they have been shown to apply to high-level and low-level languages of various forms, including logic, functional, imperative, and object-oriented programming languages. We believe that they can make a significant contribution to the problem of developing and maintaining reliable software by verifying critical program properties in a manner that is both controlled and understood by the programmer, yet automatically checked and easily evolves with the program.

2 Results from Prior NSF Awards

2.1 Type Theory and Operational Semantics, R. Harper

The objective of this CAREER Award (9502674, June 1995-June 1998) was to investigate the use of type theory and operational semantics in the design and implementation of programming languages and in teaching introductory programming to first- and second-year undergraduate students.

This grant supported my research on the type-theoretic interpretation of Standard ML and its application to the design and implementation of TIL, a type-based certifying compiler for the Standard ML core language. This grant also supported my work on the design of a new curriculum for Computer Science 15-212: Principles of Programming.

The role of type theory in language design and implementation is exemplified by my work with Chris Stone on the type-theoretic interpretation of Standard ML. This interpretation serves as a formal definition of the static and dynamic semantics of Standard ML in terms of a type theory that accounts for modularity, polymorphism, data abstraction, higher-order functions, and control and store effects. The interpretation takes the form of an elaboration into this type theory in a manner similar to that used in the formal definition of the Standard ML language. This interpretation avoids the need for implicit evaluation rules that inhibit formal verification, and clarifies a number of crucial issues in the design of the language (especially its module system and the treatment of polymorphism in the presence of effects). This interpretation is the starting point for the TIL compiler, which explored the use of typed intermediate languages and type-base translation to improve the efficiency of generated code

even in the presence of unknown types (such as arise in modular programming). The TIL compiler also demonstrated the use of types to improve the reliability and maintainability of the compiler by type checking the results of each intermediate translation step.

This grant also supported the design of a new curriculum for the second-semester undergraduate course in computer science at Carnegie Mellon. The new curriculum makes use of techniques from type theory and operational semantics in teaching introductory programming using Standard ML. The overall goal of the course is to teach the students to exploit techniques for reasoning about program behavior during program development. The curriculum relies on the use of structured operational semantics to define the semantics of the language and to support reasoning about programs using such techniques as structural induction and data type representation invariants.

The following papers report results obtained with the support of this grant: [HS00, BH97, Har99, TMC+96].

2.2 Logical Frameworks, R. Harper and F. Pfenning

Robert Harper and Frank Pfenning were co-principal investigator on NSF Grant CCR 9303383, *Design, Implementation, and Application of a Framework for the Formalization of Deductive Systems*, September 1, 1993–August 31, 1996 and then Frank Pfenning sole principal investigator on grant CCR-9619584, August 1, 1997–July 31, 1999 with the same title. Presently, Frank Pfenning is principal investigator on NSF grant CCR-9988281, September 1, 2000–August 31, 2003. The description below summarizes the main accomplishments of the first two grants. All papers, a bibliography, and further material on this and related work on the LF logical framework and its implementation in Elf are accessible at `http://www.cs.cmu.edu/~twelf/`. A structured surveys of work on logical frameworks can be found in [Pfe96] and [Pfe01].

This research is directly relevant to the proposed work, because the technology of logical frameworks relies heavily on dependent types, which also constitute a cornerstone of type refinements.

A *logical framework* is a meta-language for the formalization of deductive systems. The use of such systems in the description of programming languages and logics is wide-spread current practice, for example, type systems or operational semantics are mostly specified via inference rules. Programming languages thus constitute a major application area for logical frameworks. In some cases (like polymorphic type reconstruction or properties of continuations), new algorithms or proofs have been discovered with the help of the framework and its implementation we developed during this grant.

Case Studies. We have carried out a number of case studies of the use of logical frameworks, including cut-elimination in intuitionistic, classical, and linear logics, the latter leading to the discovery of new algorithms [Pfe00, CP98]. In another study we investigated properties of terms in continuation-passing style [DDP99]. In this instance we were able to prove an open conjecture with the help of Elf.

Framework Refinements and Extensions. We designed and implemented a calculus for the modular presentation of deductive systems, loosely structured after the module system of ML [HP98]. We have also made significant progress in the design and implementation of a system for checking properties of specifications (expressed as signatures) in the logical framework.

We have further designed and analyzed the theory of various framework refinements in order to allow more immediate and natural encodings for a larger class of object languages and theories. These include subtyping [Pfe93], linearity [CP96], and definitional equality [Vir99]. A cornerstone for further techniques is a recent robust proof for properties of logical frameworks [HP00].

Meta-Logical Frameworks. Most recently, we have been working toward the automation of meta-logical proofs, generalizing the previous paradigm of proof implementation and checking. This work is described in [SP98]. The current implementation of the Twelf system [PS99] (the successor to Elf) can automatically prove a number of interesting theorems previously only implemented and partially checked, including various properties of compiler correctness, type preservation, and cut elimination. We have also used a version of the system for experiments in proof-carrying code [PP99]. Twelf is also in active use at Stanford and Princeton for experiments in proof-carrying code [AF99, AF00] and certifying decision procedures [SD99].

2.3 Staged Computation, F. Pfenning and P. Lee

Frank Pfenning and Peter Lee were co-principal investigators on CCR-9619832, supported under the *Software Engineering and Languages* program at NSF, May 1997-October 2000. Under this grant, we developed the theoretical foundations (based on type theory) and language design (using ML as a starting point) for exploiting staged computation, leading us to the language PML. The appropriate type-theoretic foundations turns out to be based on modal logic, as proposed in [DP96, Dav96, DP00b] and applied to run-time code generation in [WLP98, WLPD98].

3 Proposed Research

3.1 A Type-Theoretic Framework

As the name suggests, we view type refinements as an extension of, rather than an alternative to, conventional static type systems. In our view the role of a static type system is as a form of context-sensitive grammar that defines the set of well-formed programs. Each type comes equipped with expressions for creating values of that type and for manipulating values of that type. The totality of the types define the language; the richer the type system, the more expressive the language.

The dynamic semantics of a language determines the computational meaning of a well-formed program in that language by defining how to execute it. Once the dynamic semantics is fixed, we may then discuss properties of the execution behavior of programs. Foremost among such properties is type safety, which associates a canonical property with each type describing the execution behavior of expressions of that type. For example, if e is an expression with static integer type, then type safety assures us that if e evaluates to a value v, then that value must be an integer. An execution property may be systematically associated with each type in such a way that any well-typed program is ensured to satisfy this property. This is, in essence, the theory of logical relations [Sta85].

Not all interesting properties arise in this manner. For example, we may consider the property `even` of the type `int` stating that if e : `int` has the property `even` and evaluates to v, then v is divisible by two. It is then a fact that the doubling function, if applied to an integer yielding a value v, will ensure that v has the property `even`. Similarly, we may associate a

property with the type `object` in an object-oriented language that states that an expression e of type `object` is not null, or that it inhabits a particular class in the class hierarchy. This information may be used to suppress null checks, or to optimize dynamic dispatch. Finally, value range properties, such as the requirement that in integer lie within the range $[0, 10]$, are of particular interest, since they lie at the heart of many compiler optimizations and are particularly suitable for mechanical verification.

We propose to develop a core theory of type refinements with these attributes:

- Each property refines a given type by isolating those expressions of that type satisfying specified run-time behavior.
- Each type constructor induces a satisfaction-preserving action on properties defining safety for that type; in particular, every well-typed program satisfies the canonical safety property associated with it.
- The class of properties is rich enough to include both refinement types and dependent refinements.

Initial work in this direction was carried out by Ewen Denney [Den98] and Susumu Hayashi [Hay93] for purely functional languages. We propose to extend this framework to languages with computational effects in two, distinct senses. First, we propose to account for *persistent* properties in the presence of effects. Second, we propose to investigate the theory of *ephemeral* properties of programs with effects.

By a *persistent* property we mean one that holds of an immutable value (such as a number, or a list of numbers); satisfaction of such a property does not change during execution. The purely functional framework of Hayashi and Denney (cited above) considers persistent properties in an effect-free setting. It is important to extend the theory of persistent properties to languages with effects, including non-termination and performance of I/O. For even in the presence such effects, it is important to be able to state and verify persistent properties of values. We expect to build on the work of Davies and Pfenning on modal type systems [PD01] as a foundation for this work, which in turn is a refined analysis of Moggi's monadic metalanguage [Mog89].

By an *ephemeral* property we mean one that may hold at a given moment of execution, but may fail to hold at some later stage. For example, a given assignable variable may hold an even integer at a particular moment, but hold an odd integer at some other moment. To take another example, a program may hold a lock for concurrency control at one point in a program, and fail to hold it at another. Indeed, we may wish to insist that a lock *not* be held at a given point as a way of ensuring that locking protocols are obeyed.

We propose to investigate the integration of these techniques into the framework of type refinements. The main idea is to treat ephemeral properties as properties of the "world", the (implicit) context in which programs are executed. The world comprises the input/output devices, the mutable storage, and any other expendable resource used by a program. The framework of type refinements may be extended to describe not only the input/output behavior of a function, but also its effect on the world as a whole by simply thinking of the world as an additional, implicit argument. For example, suppose the world consists of one piece of integer state. We may state explicitly that a given function, which modifies the world, preserves the property that the state contains an even number.

However, this global approach does not scale to more realistic situations in which the world has a more complex structure such as a heap for dynamically allocated data. For in that case the specification of the behavior of a function on the entire world becomes unwieldy.

What we seek, instead, are methods for specifying only the "local" behavior of a function, its effect on those aspects of the world that it actually uses. Recent work by Reynolds [ORY01], O'Hearn [IO01] and Walker [WCM00, WW01], among others, has stressed the importance of sub-structural logics to support such local reasoning about the world. We propose to investigate how to integrate these methods into the framework of type refinements.

The motivation here is similar to that of types-and-effects systems [JG91], but the technical development we propose is rather different. In particular, by maintaining a clear separation between properties and types we avoid the problem of always having to simultaneously specify the type and effect of an expression. By separating concerns we allow for selective specification of effect properties in situations where it is appropriate to check and enforce them, without requiring that they be fully specified as part of the type system.

3.2 Feasible Fragments

A unifying, type-theoretic framework provides the central conceptual foundation for the design of elegant, uniform, and modular type refinement systems. However, by itself a framework is not sufficient to guarantee that we obtain a usable type system with a feasible type checking problem. Here, some delicate balancing is required to obtain the right blend of programmer-supplied information, inference, practical efficiency, and accuracy of the refined type information.

We propose to investigate this design space using both theoretical analysis and prototype implementations. We highlight some of the possible choices by isolating extreme points and discussing their positive and negative properties.

Verbosity *vs.* Checking Time *vs.* Expressiveness. At first thought this may seem impossible, but there is one extreme that admits full behavioral specification, yet has a decidable checking problem. We can achieve this simply by insisting that the programmer write out a formal proof that the program satisfies its specification. With technology such as the LF logical framework [HHP93, PS99] or oracle-based checking [NR01], such proofs can be checked efficiently in practice. However, the correctness proofs would outweigh the size of the program and would be much more difficult to construct, even if we make the somewhat unrealistic assumption that we have sound proof systems for practical programming languages. If we omitted the proof, the corresponding inference problem would become undecidable.

On the other extreme we have languages such as ML that admit type inference with almost no programmer-supplied information. In contrast to the previous extreme, we have exceedingly concise programs, but we pay for this with a very limited set of program properties that can be verified. And type inference, though theoretically intractable [Mai92], is practically feasible in this case. In part this can be attributed to the fact that its complexity is pseudo-linear in the size of the largest required type [McA96]. Similar results are likely for some type refinements, although there also some negative results [Rey96].

Several recurring themes of type checking algorithms provide some theoretically motivated intermediate points between these extremes with many practical merits. One of the major items of proposed work will be to consider how these techniques apply in the context of uniform type-theoretic framework described in the previous subsection.

The Refinement Restriction. Type systems that are undecidable when considered in isolation can become decidable when superimposed on a simpler type system, that is, if pressed

into service for refining an underlying type system. For example, full inference for datasort refinements with intersections is decidable via abstract interpretation over a user-defined lattice, while full inference for unrestricted intersections is undecidable. We believe the practical value of limiting oneself to refinements (even at the cost of generality) is easily underestimated. But there can be subtle interactions between the refinements and the underlying types, especially in the presence of polymorphism. We propose to investigate such interactions.

Conservative Extension. One practically very important property of type refinement is that, by design, it is conservative over the underlying type system. If no refinements are introduced by the programmer, then type checking will behave exactly as before. That is, precisely the same errors will be flagged and with essentially the same efficiency. The more refinements one introduces, the more errors will be caught while type-checking slows down. One of the goal in the design of refinement type system is to make this trade-off attractive to the programmer. One technique we propose to investigate is to support checking of refinements only in certain, specified regions of a program, under programmer control. Not only does this allow selective deployment of refinement checking, it also makes it more efficient by restricting the regions of program over which checking must be performed.

Bi-Directional Checking. One critical tool to control verbosity of types is bi-directional type checking [Boe89, Rey96, PT98, DP00a] where type information flows through a term combining bottom-up and top-down propagation in a strictly prescribed manner. Interestingly, this is directly related to the notion of focusing proofs used to justify logic programming languages and efficient theorem provers [And92]. Bi-directional checking is robust in that it applies in many situations such as those involving polymorphism and subtyping, fully dependent types, dependent refinements, and datasort refinements. It is one of the primary tools to reduce verbosity (when compared to systems where types are always synthesized bottom-up) and reduces complexity of checking (when compared to full inference).

Constraint-Based Inference. A second recurring theme in type-checking is constraint-based inference (see, for example, [SOW97]). If we can arrange that type refinement checking reduces to constraint satisfaction over some domain, we can interleave type-checking proper and incremental constraint simplification to obtain a feasible algorithm for program property verification. The main precondition is that the constraint domain have an incremental constraint solving algorithm that is practically efficient on the kind of constraints generated by type-checking. Fortunately, there are many useful domains where this is the case, such as the integers with equality and inequalites. This has been the basis for DML (Dependent ML) [Xi98], one of the prior experiments with type refinement.

Combining Decision Procedures. The interactions of bi-directional checking and constraint-based inference are non-trivial and have to be reconsidered in each case. Even for the relatively simple case where we try to combine two decidable and practical refinement systems (datasort and dependent refinements), one can quickly slide into undecidable problems. General techniques for combining decision procedures [NO79] might help in some cases, but often they do not apply because the underlying theories interact too strongly. One possible solution is to erect boundaries in the very *definition* of the type refinement system to forestall such complications. For example, we might layer dependent refinements on top of datasort

refinements without further interactions between the layers. At present it is unclear if such a design would be accurate enough and if checking would be practical.

One of the important items of proposed work will be to study both bi-directional and constraint-based inference in our uniform framework to develop meta-theoretic results that allow a more modular construction of complex refinement system than is possible today.

3.3 Modularity

A crucial advantage of type refinements is that they extend the programming language with a formal means of stating (persistent and ephemeral) properties of programs. This is in sharp contrast to tool-based approaches in which the information gleaned by the tool is confined to the internal data structures of that tool. Such approaches afford no means of explicitly stating assertions about the behavior of programs. In particular, they provide no means of stating assumptions about unknown (separately compiled) portions of a program. In contrast, a prime virtue of type refinements is precisely that they provide a formal means of stating such assumptions at module boundaries.

We propose to develop the theory of type refinement for modular programming languages, chiefly ML [MTHM97, O'C]. The ML module system remains the most highly developed and richly expressive language for modular programming. It includes a rich formalism for specifying modules ("signatures"), expressive constructs for building hierarchical, parameterized modules ("structures", "sub-strucutres", and "functors"), and a flexible compilation and linking mechanism [BA99, O'C].

An important direction for research is to extend the expressiveness of the ML signature language to include declarations of refinement properties. By stating properties of the operations of the abstract type, we can track these properties in client code. For example, consider the following signature of "optional" values of a type:

```
signature OPTIONAL = sig
  type α option
  exception Null
  val null : α option
  val just : α -> α option
  val check : α option -> α option
  val the_value : α option -> α
end
```

A value of type τ `option` is either the "null" object, or is just a value of type τ, marked as being non-null. The value `null` is the null object of any type; the function `just` marks a value of type α as being a non-null value of type α `option`. The function `check` checks whether its argument is null, raising the exception `Null` if it is, and otherwise returning the object itself. The function `the_value` simply returns the underlying value of a non-null object, raising `Null` if it is null.

A natural use of refinements in this setting is to introduce a persistent property stating that a value of option type is non-null. This may be achieved by the following declarations augmenting the above signature:

```
refine α null <: α option
refine α non_null <: α option
prop null : α null
prop just : α -> α non_null
prop check : α option -> α non_null
prop the_value : α non_null -> α
```

Here we choose to track two refinements, α `null` and α `non_null` of the type α `option`. We may think of these as defining "subsets" of the type α `option`. The specifications for `null` and `just` assert that the former yields a null object, the latter a non-null object. The property for `check` asserts that the result is non-null, regardless of the argument. The property for `the_value` asserts that the argument must be non-null.

Using these property declarations we can derive properties of user code. For example, consider the following case analysis function.

```
fun compose (x : α option, f : α -> β) : β option =
  just (f (the_value (check x)))
  handle Null => null
```

A refinement checker may deduce that the function `compose` has the conjunctive property

```
prop compose : α null * (α -> β) -> β null
             & α non_null * (α -> β) -> α non_null
```

stating that `compose` yields a null object if called with a null object, and yields a non-null object if called with one. The checker may also deduce that the call to `the_value` may suppress a null check, because the result of `check x` will have the property α `non_null`, as required by `the_value`.

3.4 Implementation and Evaluation

As discussed in the preceding sections, devising a system of type refinements is a challenging task and involves balancing various criteria such as parsimony of refinement specifications, practical efficiency of checking, and expressive power of the language for refinements. It can therefore only be successful if theoretical groundwork is supported with implementation and experimentation.

We already have some experience with refinement type implementations through our work on datasort refinements and dependent refinements. Our preliminary conclusions about the important aspects of such an implementation may be somewhat surprising. We therefore detail them here together with a discussion of proposed work.

Efficiency of Refinement Checking. The first system we devised, namely datasort refinements, had the property that full refinement inference was decidable via a combination of algorithms from regular tree automata and abstract interpretation. The theoretical worst-case complexity results were not promising: the tree automata algorithms require exponential time, type inference itself was super-exponential. Essentially, it requires one further exponent for every increase in the order of the type of the defined function. Nonetheless, practical

efficiency was not a problem with our first implementation with only relatively simple memoization techniques. We attribute this practical efficiency primarily to three factors. Most importantly, inference is performed incrementally for each function rather than for the whole program. Typically, each individual function is small. Secondly large tree automata never arose because the major distinctions were made in the underlying type system and refinements were relatively straightforward. Finally, functions of order higher than one or two are rare in practical programming.

For the system of dependent refinements over integers, efficiency could be controlled effectively through simple pre-processing and straightforward implementations of standard variable elimination techniques. Again, the fact that checking is modular and incremental plays an important role in the practicality of the system.

In neither case has efficiency be a major issue. As we push the boundaries of what can be expressed this might change; for now we do not expect this to become a major focus of the proposed work.

Inference *vs.* Checking. Despite its practicality, we decided to abandon full inference, since full inference captures and subsequently exploits too many accidental properties of programs. This led to problems in that some errors were not caught, or caught at the wrong program points. For example, assume we are implementing red/black trees. This data structure must satisfy several strong invariants, one of them stating that no red node has a red child node. The ordinary type α `tree` does not capture this property, while its refinement α `rbt` does. Now, if we have a function

```
val insert : α entry * α tree -> α tree
```

we can in fact check that, when given an entry and a valid red/black tree is returns another valid red/black tree, that is

```
prop insert : α entry * α rbt -> α rbt
```

However, the same code also has some well-defined, though entirely accidental behavior when given an *invalid* tree. If we use full inference, this behavior will be captured and allowed, preventing an appropriate error to occur where `insert` is applied to an invalid tree.

It turns out that the additional required annotations to make the expected properties explicit is not a significant burden, because it only formalizes the intuition and design knowledge that the programmer had in the first place. In fact, we found it helps in producing correct code from the start if the programmer formulates explicitly, yet concisely, which invariants he or she expected the code to satisfy.

We plan to investigate how checking may be combined with *local* inference of refinement properties in order to avoid excessively verbose annotations that might arise in more expressive refinement languages. This is related to a similar approach by Pierce and Turner [PT98] for polymorphism and subtyping. Even in datasort refinements such local inference might be exploited when there are additional internal properties of functions that are not exported. For example, in some implementations the color invariant of red/black trees is temporarily violated and then restored, locally. This should not be visible to client code, something that usually enforced at module boundaries. However, it might be beneficial to perform some inference to detect such local invariants without further information from the programmer.

Error Diagnosis. The error diagnosis improved greatly when we switched from full inference to bi-directional checking. In fact, subjectively, it has been more accurate than error diagnosis for ML types, since no variables are created and unified as is the case in type inference. The principal remaining problem with error diagnosis is that it is sometimes difficult to decide if the program is incorrect, the refinement specification is incorrect, or the invariant is too weak and needs to be generalized. We propose to work on tools in the programming environment to help correctly diagnose the kind of error as reported by refinement checking.

Host Language. One critical property of type refinements is that, as type systems, they are to some extent independent of the underlying host language, or at least can be adapted to various languages. For an implementation and experimentation, we would like to choose a language which has a strong type system to begin with and at the same time has a well-developed module system, since many of the practical benefits derive from modular programming and are most palpable during program maintenance and evolution.

Since ML has the most advanced type and module system available at present, we propose to use ML as our main experimental testbed. In addition, we already have significant implementations of both datasort refinements and dependent refinements in ML, and we have reasonably large code bases for evaluation.

We may also consider Java as a second testbed for versions of type refinements. Already, Xi has designed a version of dependent refinements for Java [XX99], and datasort refinements would also seem to apply and interact well with the class hierarchy.

Evaluation. Evaluation proceeds in two ways. One is to take existing, large code bases such as the Twelf implementation [PS99], or the TIL compiler [TMC+96] that were developed here at Carnegie Mellon University and instrument them with refinement specifications that the current ML compiler will view as comments and our refinement type checker will see as obligations. This will give us information about both efficiency, verbosity, expressiveness and practical utility of specific type refinement systems. Many of these can be expressed in clear measurements.

The second is to use refinement types during our own code development. The results will be much more difficult to quantify (controlled studies would seem to be next to impossible), but will nonetheless provide important feedback about limitations of expressive power, precision of error messages, and other pragmatic aspects of our systems.

3.5 Applications

As discussed in the introduction, we believe that type refinements offer a practical way to specify and check many more program properties than is possible with current type systems. Refinement and property specifications serve as formally verified documentation and easily evolve with the program. We believe that type refinements can therefore make a significant contribution towards development and maintenance of reliable, modular software.

Below we give some more specific examples of the kinds of refinements we plan to support. It should be clear that these properties are indeed of practical interest and that their verification will catch bugs that might otherwise escape detection.

Data Structure Invariants. Refinement types may be used to state and check simple properties of data structures such as the color conditions on red/black trees using datasort refinements. We anticipate that these ideas will extend to properties of similar complexity for

a wider variety of data structures. One of the most critical items of future work is to reconsider the issue of datasort refinements and data abstraction, especially at module boundaries. We expect the work on a unifying type-theoretic framework to point the way towards the most effective way to communicate refinements and properties at module boundaries.

Value Range Invariants. Dependent types may be used to track ranges of values such as the bounds on indices used to access arrays. They can also be used for more complex data structure invariants such as balance invariants on red/black trees. The strength of these properties depends on the strength of the formalism used to express constraints. We propose to experiment with a variety of systems of constraints and their combinations to assess the practicality of dependent types.

Dimensional Analysis. Type refinements may be used to track dimensions of data. For example, one may regard `kg`, `m`, and `s` to be refinements of the type `float` for values that represent masses, lengths, and time intervals, respectively. Dimensions combine to form compound dimensions such as `kgm` and $\texttt{kgm}/\texttt{s}^2$. Flexible treatment of exponents requires a constraint domain similar to that required for dependent types. We propose to investigate the combination of dimensional analysis [Ken94] with integral constraint domains to track dimensionality of expressions in a program.

Resources. Sub-structural logics support local reasoning about ephemeral properties of the "world" [WCM00, WW01]. For example, we may wish to track whether a particular lock is held at given program point, and enforce the requirement that it be free on entry to and exit from a given function. We propose to investigate the use of type refinements to ensure that a program complies with such a locking protocol. Note that this is a fundamentally different problem from protocol verification, which seeks to ensure that protocols have certain semantic properties of interest. It is a combination of the two which would be desirable in practice, since a protocol property is only useful if the implementation actually adheres to the protocol.

References

[AF99] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th Conference on Computer and Communications Security*, pages 52–62, Singapore, November 1999. ACM Press.

[AF00] Andrew W. Appel and Amy P. Felty. A semantic model of types and machine instructions for proof-carrying code. In Thomas Reps, editor, *Conference Record of the 27th Annual Symposium on Principles of Programming Languages (POPL'00)*, pages 243–253, Boston, Massachusetts, January 2000. ACM Press.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[BA99] Mathias Blume and Andrew W. Appel. Hierarchical modularity. *ACM TOPLAS*, 21(4):813–847, 1999.

[BH97] Lars Birkedal and Robert Harper. Relational interpretations of recursive types in an operational setting (summary). In *Theoretical Aspects of Computer Science*, pages 458–490, Sendai, Japan, September 1997.

[Boe89] Hans-J. Boehm. Type inference in the presence of type abstraction. In *Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, Portland, Oregon*, pages 192–206. ACM Press, June 1989.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, pages 278–292, 1991.

[CP96] 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.

[CP98] Iliano Cervesato and Frank Pfenning. A linear logical framework. *Information and Computation*, 1998. To appear in a special issue with invited papers from LICS'96, E. Clarke, editor.

[Dav96] Rowan Davies. A temporal logic approach to binding-time analysis. In E. Clarke, editor, *Proceedings of the Eleventh Annual Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[Dav97] Rowan Davies. A practical refinement-type checker for Standard ML. In Michael Johnson, editor, *Algebraic Methodology and Software Technology Sixth International Conference (AMAST'97)*, pages 565–566, Sydney, Australia, December 1997. Springer-Verlag LNCS 1349.

[DDP99] Olivier Danvy, Belmina Dzafic, and Frank Pfenning. On proving syntactic properties of CPS programs. In Andrew Gordon and Andrew Pitts, editors, *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)*, Paris, September 1999. Electronic Notes in Theoretical Computer Science, Volume 26.

[Den98] Ewen Denney. Refinement Types for Specification. In David Gries and Willem-Paul de Roever, editors, *IFIP Working Conference on Programming Concepts and Methods (PROCOMET '98), Shelter Island, New York, USA*, pages 148–166. Chapman and Hall, 1998.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq SRC, December 1998.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Guy Steele, Jr., editor, *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg Beach, Florida, January 1996. ACM Press.

[DP00a] Rowan Davies and Frank Pfenning. Intersection types and computational effects. In P. Wadler, editor, *Proceedings of the Fifth International Conference on Functional Programming (ICFP'00)*, pages 198–208, Montreal, Canada, September 2000. ACM Press.

[DP00b] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 2000. To appear. Preliminary version available as Technical Report CMU-CS-99-153, August 1999.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation*, pages 268–277, Toronto, Ontario, June 1991. ACM Press.

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, Carnegie Mellon University, March 1994. Available as Technical Report CMU-CS-94-110.

[Har99] Robert Harper. Proof-directed debugging. *Journal of Functional Programming*, 9(4):463–470, July 1999.

[Hay93] S. Hayashi. Logic of refinement types. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 108–126. Springer, Berlin, Heidelberg, 1993.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HP98] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.

[HP00] Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. Technical Report CMU-CS-00-148, Department of Computer Science, Carnegie Mellon University, July 2000.

[HS00] Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 2000. Extended version published as CMU technical report CMU-CS-97-147.

[IO01] Samin Ishtiaq and Peter O'Hearn. BI as an assertion language for mutable data structures. In *Conference Record of POPL 2001*, pages 14–26, London, UK, January 2001.

[JG91] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *Eighteenth ACM Symposium on Principles of Programming Languages*, pages 303–310, 1991.

[Ken94] Andrew Kennedy. Dimension types. In Donald Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, pages 348–362, Edinburgh, U.K., 1994. Springer Verlag LNCS 788.

[Lei01] K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*, pages 157–175. Springer-Verlag, 2001.

[Mai92] H.G. Mairson. Quantifier elimination and parametric polymorphism in programming languages. *Journal of Functional Programming*, 2(2):213–226, April 1992.

[McA96] David McAllester. Inferring recursive data types. Draft Manuscript, 1996.

[Mog89] Eugenio Moggi. Computational lambda calculus and monads. In *Fourth Symposium on Logic in Computer Science*, Asilomar, California, June 1989.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

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

[NR01] George C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 142–154, London, UK, January 2001.

[NS95] H.R. Nielson and K.L. Solberg, editors. *Types for Program Analysis*, University of Aarhus, Denmark, May 1995. Informal Workshop Proceedings.

[O'C] The O'Caml language. URL: `http://www.ocaml.org`.

[Oka99] Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, July 1999.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *CSL 2001*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[Pfe93] Frank Pfenning. Refinement types for logical frameworks. In Herman Geuvers, editor, *Informal Proceedings of the Workshop on Types for Proofs and Programs*, pages 285–299, Nijmegen, The Netherlands, May 1993.

[Pfe96] Frank Pfenning. The practice of logical frameworks. In Hélène Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1059. Invited talk.

[Pfe00] Frank Pfenning. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation*, 157(1/2):84–141, March 2000.

[Pfe01] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 977–1061. Elsevier Science and MIT Press, 2001.

[PP98] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. In L. Cardelli, editor, *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, pages 209–221, San Diego, California, January 1998. ACM Press.

[PP99] Mark Plesko and Frank Pfenning. A formalization of the proof-carrying code architecture in a linear logical framework. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of the 25th Symposium on Principles of Programming Languages (POPL'98)*, 1998. Full version in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1), January 2000, pp. 1–44.

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[SD99] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[SOW97] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.

[SP98] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, pages 286–300, Lindau, Germany, July 1998. Springer-Verlag LNCS 1421.

[Sta85] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[TMC+96] 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.

[Vir99] Roberto Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, September 1999. Available as Technical Report CMU-CS-99-167.

[WCM00] David Walker, Karl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.

[WLP98] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and modal-ML. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 224–235, Montreal, Canada, June 1998. ACM Press.

[WLPD98] Philip Wickline, Peter Lee, Frank Pfenning, and Rowan Davies. Modal types as staging specifications for run-time code generation. *ACM Computing Surveys*, 30(3es), September 1998.

[WW01] David Walker and Kevin Watkins. On linear types and regions. In *ACM SIGPLAN International Conference on Functional Programming*, pages 181–192, September 2001.

[XH01] Hongwei Xi and Robert Harper. Dependently typed assembly language. In *Proceedings of the 6th International Conference on Functional Programming (ICFP'01)*, pages 169–180, Florence, Italy, September 2001. ACM Press.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.

[Xi00] Hongwei Xi. Imperative programming with dependent types. In *Proceedings of 15th Symposium on Logic in Computer Science (LICS'00)*, pages 375–387, Santa Barbara, June 2000.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.

[XX99] Hongwei Xi and Songtao Xia. Towards array bound check elimination in java virtual machine language. In *Proceedings of CASCOON '99*, pages 110–125, Mississauga, Ontario, November 1999.