Mechanized Metatheory for User-Defined Type Extensions [*]

Daniel Marino Brian Chin
Todd Millstein
University of California, Los Angeles
{dlmarino,naerbnic,todd}@cs.ucla.edu

Gang Tan
Boston College
gtan@cs.bc.edu

Robert J. Simmons
David Walker
Princeton University
{rsimmons,dpw}@cs.princeton.edu

Motivation Type systems are a natural discipline for ensuring that programs maintain certain runtime invariants. Of course, language designers cannot anticipate all the invariants that programmers will want to enforce. Therefore, it is desirable to allow programmers to specify and statically check invariants of interest for their applications.

Researchers have designed expressive type systems that allow programmer-defined invariants to be directly encoded as types. However, there is a tradeoff between type-system expressiveness and ease of use for programmers. First, the more expressive the type system, the more annotation burden there is on the programmer. For example, expressive type systems often require programmers to manually discharge proof obligations to ensure that a program fragment meets its specified type. Second, the more expressive the type system, the more difficult it is for programmers to understand. In traditional type systems, each type has a relatively simple set of syntax-directed rules, which constitutes a *programming discipline* for programmers to obey. As type systems become more expressive, it becomes more difficult for programmers to understand what programming idioms can and cannot be typechecked and why. As a result, languages with expressive type systems may be challenging for programmers to use effectively.

Our Approach We are pursuing an approach to enriching traditional type systems with support for user-defined invariants while maintaining ease of use. The key idea is to partition the task of proving invariants about programs into two distinct roles: the *type-system extender* (TSE) and the programmer. A TSE can define a new type annotation along with a set of typing rules that constitutes the associated programming discipline. The TSE manually proves once and for all that an annotation's programming discipline is sufficient to ensure a desired runtime invariant. Separately, any number of programmers may employ the new annotation in programs, which are statically checked to obey the annotation's typing rules.

As a nontrivial example, consider ensuring that programs have no runtime race conditions. A TSE can define a new annotation `guardedBy(l)`, which allows programmers to associate a lock with each variable, along with a set of rules that conservatively ensure a variable's lock is held whenever the variable is accessed. For a program to typecheck, the programmer need only obey this simple locking discipline, with no manual proof obligations required. Offline, the TSE proves once and for all that every program obeying this discipline avoids race conditions on "guarded" variables.

In our approach, programmers need not be provers; all proving is done by the TSE, and this proving is done once per annotation rather than once per program. Further, programmers are provided with an explicit programming discipline to obey. While this discipline likely rules out some useful ways to satisfy a particular invariant, we expect this loss of flexibility to be worth the gain in simplicity and understandability for programmers. Essentially, our approach corresponds to proving invariants of a set of *programming idioms*, which can then be safely used in *any* program, rather than proving invariants about each program individually.

A useful analogy exists with the idea of domain-specific programming languages (DSLs). Although DSLs are often less expressive than general-purpose languages, the intent is that a DSL will be expressive enough for its target domain and significantly easier for programmers to use and understand. Our approach to proving invariants about programs amounts to allowing TSEs to define domain-specific typing rules that conservatively ensure an invariant. The hope is that these rules are expressive enough to admit the desired programming idioms and significantly easier for programmers to use than a single general-purpose type system.

Current Status We previously built the CLARITY framework for user-defined type qualifiers in C based on this approach [1]. User-defined typing rules in CLARITY were restricted to a very simple form, and user-defined invariants were restricted to a decidable logical theory. These restrictions allowed a TSE's proofs to be discharged automatically but severely limited expressiveness.

We are building a much more flexible framework for user-defined type extensions using the Twelf proof assistant [2]. Our framework extends a Twelf formalization of an imperative language that includes a standard type system and type soundness proof [3]. A TSE will define new type annotations and associated typing rules in a stylized language that desugars into Twelf syntax. The TSE will also define each annotation's associated invariants. Finally, the TSE will define the relevant cases of the lemmas and theorems that make up an extended type soundness proof, which guarantees that each annotation's rules ensure the intended invariants.

The LF type theory that underlies Twelf provides a powerful platform, allowing TSEs to conveniently encode the syntax, typing rules, and metatheory for a type system extension in a unified representation. Twelf's support for totality checking will be used to verify the metatheory, and Twelf's logic programming engine will allow Twelf itself to be used as the typechecker for programs.

References

[1] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–95, New York, NY, USA, 2005. ACM Press.

[2] F. Pfenning and C. 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, 1999. Springer-Verlag LNAI 1632.

[*] This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0238328, CCF-0427202, CCF-0545850, and CNS-0627650, as well as by a gift from Microsoft Research.

[3] R. Simmons. Twelf as a unified framework for language formalization and implementation. Undergraduate honors thesis, Princeton University, 2005.