Enforcing and Validating User-Defined Programming Disciplines

Brian Chin Daniel Marino Shane Markstrum Todd Millstein

Computer Science Department
University of California, Los Angeles
{naerbnic,dlmarino,smarkstr,todd}@cs.ucla.edu

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Reliability, Validation; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Invariants, Mechanical verification, Specification techniques

General Terms Languages, Reliability, Verification

Keywords programming disciplines, type qualifiers, pluggable type systems, type-and-effect systems

1. Motivation

One way that programmers manage the complexity of building and maintaining software systems is by adhering to *programming disciplines* of various sorts. Informally, a programming discipline is a set of rules governing the ways in which certain program entities may be manipulated. For example, a common synchronization discipline associates a lock with some data structure and requires the lock to be acquired before the data structure may be accessed, thereby preventing race conditions.

The static type systems used in mainstream programming languages can be seen as enforcing some commonly useful programming disciplines. Unfortunately, many disciplines are not able to be enforced, let alone specified, in today's programming languages. At best they are described informally in documentation like comments, which are useful but can easily become out of date.

Programming language researchers have shown how to extend traditional type systems to enforce many kinds of disciplines. For example, the synchronization discipline described above has been cast as an extension to Java's type system [5]. However, language designers cannot anticipate all the programming disciplines that programmers will want to enforce.

2. User-Definable Programming Disciplines

To remedy this problem we are developing frameworks for *user-definable programming disciplines*. Our approach involves three main tasks (Figure 1):

Discipline Specification Users of the framework define new program annotations. The framework provides a language in which an annotation's associated programming discipline can be declaratively specified as a set of rules about program entities.

PASTE'07 June 13–14, 2007, San Diego, California, USA.
ACM 978-1-59593-595-3/07/0006.

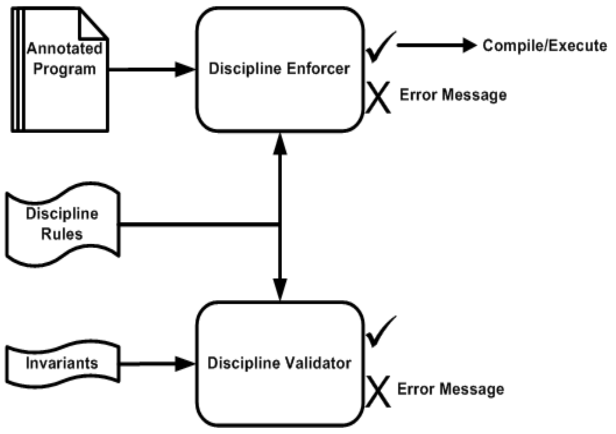

Figure 1. Our approach to user-defined programming disciplines.

```
qualifier nonzero(int Expr E)
  case E of
      C, where C != 0
    | E1, where positive(E1)
    | E1 * E2, where nonzero(E1) && nonzero(E2)
  restrict E1 / E2, where nonzero(E2)
  invariant value(E) != 0
```

Figure 2. A Clarity type qualifier for nonzero integers.

Discipline Enforcement The framework automatically ensures at compile time that programs respect their annotations, based on the user-defined discipline rules.

Discipline Validation Users may provide a set of run-time invariants that a discipline is intended to ensure. The framework then validates, possibly with user interaction, that a discipline in fact establishes its intended invariants. This validation is performed *once* when the discipline is specified, independent of any particular program that uses the discipline.

Because static type systems are a natural technique for enforcing programming disciplines, we have pursued our approach as a framework for programmer-definable type system extensions. We are exploring three instantiations of the approach, which are discussed in the rest of this section.

Clarity: Type Qualifiers for C Our Clarity framework [3] allows C programmers to easily define new *type qualifiers* [6], which are atomic tags that refine existing types. For example, Figure 2 presents a programmer-defined specification for a `nonzero` quali-

```
rule ConfinedClass(ClassDef c){
  where(confined(c)){
    require(!c.isPublic()):
      error(c, "Confined class may not be public");
    require(c.pkg() != globals.emptyPackage):
      error(c, "Confined class may not be " +
               "in the default package");
}}

rule ConfinedSubtype(a <: b @ pos){
  where(confined(a)){
    require(confined(b)):
      error(pos, "confined type "+a+
                 " may not be cast to "+
                 "unconfined type "+b);
}}
```

Figure 3. Two JavaCOP rules for confined classes.

fier. The first line introduces the qualifier and declares it to apply to expressions of type `int`.

The `case` and `restrict` blocks define the programming discipline associated with the `nonzero` qualifier. Each `case` clause specifies a subset of expressions that can be given the qualifier `nonzero`: integer constants other than zero, expressions that satisfy the discipline for another qualifier `positive` (definition not shown), and multiplication expressions whose operands recursively have the qualifier `nonzero`. A `restrict` clause specifies a requirement on certain kinds of expressions. The clause in Figure 2 indicates that the denominator of every division expression in a program must have the qualifier `nonzero`. The programmer-specified rules are combined with standard rules for typechecking expressions like variables and function calls in order to typecheck programs at compile time. Clarity also supports qualifier inference [4].

The last line of Figure 2 specifies the run-time invariant associated with `nonzero`: an expression having this qualifier should always evaluate to an integer other than zero. Clarity's *qualifier validator* component employs this invariant to automatically validate the programmer-defined rules: each rule is proven to establish this invariant, for all possible programs.

Clarity has been used to specify, enforce, and validate a variety of type qualifiers, including `positive`, `negative`, and `nonzero` for integers, `nonnull` for pointers, and `tainted` for strings.

JavaCOP: Pluggable Type Systems for Java JavaCOP [1] adapts our approach to support "pluggable type systems" for Java. JavaCOP employs Java 1.5 attributes to represent user-definable type annotations. In addition to programming disciplines on expressions as in Clarity, JavaCOP supports larger-scale disciplines on methods, classes, and packages.

Figure 3 shows two rules that are part of the programming discipline for *confined* classes, whose objects are guaranteed not to escape the current package [9]. Each rule is defined for a particular kind of abstract syntax tree (AST) node. For example, the first rule applies to all class definitions. The rule requires that if the class satisfies the user-defined `confined` predicate (definition not shown), which simply checks that the class was annotated as `@Confined`, then the class cannot be public and cannot be in the default package. User-defined `error` clauses allow for meaningful compile-time error messages. The second rule applies to any AST node that performs an explicit or implicit cast from one type to another (e.g., a cast, an assignment, parameter passing). The rule ensures that an expression of confined type is never cast to an unconfined type.

JavaCOP has been used to define a variety of pluggable type systems for Java, including forms of object confinement, static race detection, nonnull types, and types for immutability. Our current work is focusing on discipline validation for JavaCOP. We are developing tools to allow users to specify run-time properties of interest and to gain confidence that their JavaCOP rules ensure these properties.

User-Definable Type-and-Effect Systems Type-and-effect systems [7] are a natural way to extend the power of traditional type systems, by capturing information about the *evaluation* of expressions in addition to their values. For example, the locking discipline described earlier can be formalized as a type-and-effect system, where an expression's effects are the set of locks needed during its evaluation. Another example of a type-and-effect system is a type system for static checking of exception handling.

We are developing a framework for programmer-defined type-and-effect systems. Users will define new effects, provide rules that specify how effects are generated dynamically, and provide rules that statically ensure that a program's effects obey a desired programming discipline. We are currently using the Twelf proof assistant [8] as a back end for both performing type-and-effect checking on programs and establishing the soundness of the programmer-defined static rules with respect to the dynamic rules [2].

3. Acknowledgments

Thanks to our wonderful collaborators on the described projects: Chris Andreae, James Noble, Jens Palsberg, Rob Simmons, Gang Tan, and David Walker. This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0427202 and CCF-0545850, as well as by a generous gift from Microsoft Research.

References

[1] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 57–74, New York, NY, USA, 2006. ACM Press.

[2] B. Chin, D. Marino, T. Millstein, G. Tan, R. J. Simmons, and D. Walker. Mechanized metatheory for user-defined type extensions. In *1st Informal ACM SIGPLAN Workshop on Mechanizing Metatheory*, 2006.

[3] 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.

[4] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *European Symposium on Programming*, 2006.

[5] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 219–232. ACM Press, 2000.

[6] J. S. Foster, M. Fähndrich, and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[7] J. M. Licassen and D. K. Gifford. Polymorphic effect systems. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, CA, Jan. 1988.

[8] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999.

[9] J. Vitek and B. Bokowski. Confined types. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–96. ACM Press, 1999.