Program Verification through Soft Typing

Robert Cartwright & Matthias Felleisen
Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

During the past 25 years, programming language researchers have developed increasingly sophisticated techniques for statically analyzing program behavior. These techniques have been widely used in optimizing compilers to infer program properties supporting code optimization, but they have had little impact on program development and maintenance. This paper explains how these techniques can be be used to perform a restricted form of program verification, namely the automatic proof of the preconditions of primitive operations.

Computations in high-level programming languages are expressed in terms of operations on abstract values such as integers, arrays, lists, and functions. Most of these operations are defined only for selected inputs such as the set of all integers or the set of non-empty lists. Nevertheless, the machine code for an operation always does *something* regardless of the form of the input. If a language fails to prevent the application of an operation's machine code to illegal inputs (which are bitstrings), it is *unsafe* because subsequent program behavior is unpredictable.

To ensure safety, some high-level languages prevent the misapplication of operations through a combination of static and dynamic checks. Some safe languages, such as ML and Modula III, rely primarily on static checks, while others, such as Scheme and Smalltalk, rely exclusively on dynamic checks. From the perspective of program reliability, static checking is preferable to dynamic checking because it proves preconditions for program operations instead of merely testing them on individual inputs. In essence, static checking is a restricted form of program verification where the invariant assertions are the preconditions of primitive operations. If all the preconditions in a program are valid invariants, then it can never abort with a run-time error.[1]

Statically typed languages enforce the preconditions for some operations by including inference rules in the syntax of the language. But the preconditions for many operations, such as extracting the head of a list or an element of an array, cannot be proven by simple syntactic arguments. To support a more aggressive approach to proving operation preconditions, we have developed a generalization of static typing called *soft type analysis* [1] that is performed by the programming environment rather than the language compiler. A soft type analyzer generates and proves program invariants corresponding to the preconditions of primitive operations. Since these preconditions are undecidable in general, no soft type analyzer can prove every valid precondition. The analyzer flags each primitive operation

[1] Ignoring finite limits that are installation dependent such as stack overflow and heap exhaustion.

```
(define sum
  (lambda (tree)
    (if (number? tree)
        tree
        (+ (sum (car tree))
           (sum (cdr tree))))))

(sum (cons (cons '() 1) 2))
```

FIGURE 1: A simple Scheme program ...

```
(define sum
  (lambda (tree)
    (if (number? tree)
        tree
        (+ (sum (CHECK-car tree))
           (sum (CHECK-cdr tree))))))

(sum (cons (cons '() 1) 2))

; sum              2  (2 prim)
; TOTAL CHECKS     2  (of 10 is 20.0%)
```

FIGURE 2: ...and its analysis

with an unproven precondition and ensures that the condition is checked at run-time to preserve safety. Since soft typing decouples static analysis from syntax checking, it is applicable to both statically and dynamically typed languages.

Over the past five years, we have designed and implemented two different soft type analyzers for Scheme. One is based on a generalization of the familiar Hindley-Milner type discipline [1, 6]. Instead of the usual type algebra where basic types are constants and type constructors are operators, the soft type analyzer relies on an augmented type algebra including a restricted union operator. Our second soft type analyzer relies on Heintze's [5] formulation of flow analysis as set-based analysis [3]. Set-based analysis interprets [2] program constructs as operations on sets of values, capturing the intuitive idea of generalized program evaluation on the set of "all-possible" values for each expression.

Both soft type analyzers prove 90% of the preconditions (excluding the analysis of array references) in typical Scheme code. In practice, they both run in time comparable with conventional Hindley-Milner type checking. Neither system is uniformly superior to the other. Hindley-Milner soft typing works well for programs that exploit polymorphism; it proves a superset of the preconditions provable by conventional Hindley-Milner type-checking. Set-based analysis works particularly well for programs where type inference is sensitive to control flow.

Figure 2 shows the analysis performed by the SBA soft type analyzer on a simple program given in Figure 1. The Hindley-Milner analyzer produces exactly the same results. The sample program intentionally includes erroneous applications that demonstrate the capabilities of soft type analysis. The analyzer flags the applications of `car` and `cdr` in the body of `sum` because the argument to `sum` in the final line of the program is an ill-formed tree. The analyzer supports a graphical interface that marks up the text of the original program with hyperlinks giving the type of each expression and the flow constraints that produced it.

In extensive tests of both analyzers, we have made two important observations [4]. First, programmers embrace soft type analysis if it is readily available as an editor command applicable to any compilable program unit. Second, programmers can exploit the results of soft type analysis more effectively if the analyzer clearly explains why it cannot prove a given precondition. The programmer can use this information to determine whether the precondition is invalid or the analysis is too weak to prove it. In addition, the information enables the programmer to locate the bug responsible for an invalid precondition or, alternatively, to tune the code to improve the accuracy of the analysis.

Our experience with soft type analysis indicates that aggressive static analysis can help programmers write and maintain reliable code, but much research still remains to be done. In particular, we need to extend the analysis in two directions. First, we need to analyze array operations and arithmetic operations such as division. Second, we need to develop analysis methods to support important language constructs not present in Scheme such as modules and classes.

References

1. Cartwright, R. and M. Fagan. Soft typing. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991, 278–292.
2. Cousot, P. and R. Cousot. Formal language, grammar, and set-constraint-based program analysis by abstract interpretation. In *Proc. 1995 Conference on Functional Programming and Computer Architecture*, 170-181.
3. Flanagan, C. and M Felleisen. Set based analysis for full Scheme and its use in soft-typing. Manuscript. Submitted for journal publication. 1995.
   URL: `http://www.cs.rice.edu/ cormac/`.
4. Flanagan, C., M. Flatt, Shriram K., S. Weirich, and M. Felleisen. Static debugging or browsing the web of program invariants. In *Proc. Sigplan 1996 Conference on Programming Language Design and Implementation*, to appear.
5. Heintze, N. Set based analysis of ML programs. Technical Report CMU CS 93-193. Carnegie Mellon University, July 1993. In *1994 ACM Conference on LISP and Functional Programming*, 1994.
6. Wright, A.K. and R. Cartwright. A practical soft type system for Scheme. In *1994 ACM Conference on LISP and Functional Programming*, 1994, 250–262.