Abstract

Side effects are an essential part of many programs. However, side effects make it harder to understand program behavior, because expressions no longer can be treated as mere values. Pure expressions have no side effects, and research in different areas has demonstrated that purity aids program specification, optimization, testing, debugging, and maintenance.

Determining side effects is useful for program comprehension and optimization, but it is a difficult problem and particularly so in the presence of objects and higher-order procedures that may flow freely through a program. In practice it is not sufficient to detect the presence of side-effecting expressions in a program. The dynamic extent of the side effect must also be accurately established to obtain useful information. A side effect in one particular context, e.g. a procedure application, may not be observable outside that context. In this regard, manual or crude approaches to establish the extent of side effects are too imprecise and error-prone.

This dissertation explores and evaluates techniques for statically computing the side-effecting behavior of higher-order imperative programs to determine procedure purity.

We base our approach on an abstract state machine that is an interpreter for a core Scheme language, instrumented to register read and write effects on resources. Resources in our semantics are variables and objects, which are allocated in a store at a specific address. Following the AAM approach, the machine is parameterized to be able to express both concrete and abstract semantics. The result of program interpretation is a flow graph that is consumed by client analyses interested in program properties involving control flow, value flow, and effects.

Our first contribution is a procedure side-effect analysis that computes for each procedure application the side effects that are observable by direct and indirect callers. Applications and associated callers are found by traversing all reachable application contexts on the call stack at the point where an effect occurs. Observability of effects is based on freshness of resources. A resource

is fresh in a particular context if it was created in that context. Effects on fresh resources can never be observed outside that context. For example, if during procedure application an object is allocated, then that object is fresh in that application context, and any effects on it can never be observed by callers. We discuss three characterizations of freshness. The first, address freshness, is attractive because it is a close fit with the store-semantics of the abstract machine. However, this is not always ideal because termination of static analysis is primarily guaranteed by allocating resources at addresses that are already in use, and therefore not fresh. For example, a variable is typically allocated at the same address throughout an entire abstract interpretation. To improve the precision of side-effect analysis, we introduce two additional scope-based characterizations of freshness. Variable freshness is based on locality of variables, i.e. whether they are local or free with respect to a procedure's scope, and object freshness keeps track of the flow of objects in and out of scopes through object references. Their formalization forms the second contribution of this work.

Our third and final contribution is the design of a purity analysis on top of procedure side-effect analysis. Purity analysis classifies procedures as either pure, observer, or procedure. A procedure is pure if none of its applications generate or depend upon externally observable side effects. A procedure is an observer as soon as one of its applications depends on an external side effect, but none of its applications generate observable side effects. Otherwise, a procedure is classified as a procedure.

We apply the analyses presented in this work to a set of programs, and discuss the outcome in terms of four key aspects: correctness, soundness, precision, and performance. We find that our purity analysis is capable of uncovering purity in a variety of programs. Also in this setting our experiments show that our analysis is capable of correctly classifying functions in terms of the side effects they generate and depend upon.