Effect Capabilities for Haskell

Ismael Figueroa\(^1,2,\ast\), Nicolas Tabareau\(^2\), and Éric Tanter\(^1,\ast\ast\)

1 PLEIAD Laboratory, Computer Science Department, University of Chile, Santiago, Chile
2 ASCOLA Group, Inria, Nantes, France

Abstract. Computational effects complicate the tasks of reasoning about and maintaining software, due to the many kinds of interferences that can occur. While different proposals have been formulated to alleviate the fragility and burden of dealing with specific effects, such as state or exceptions, there is no prevalent robust mechanism that addresses the general interference issue. Building upon the idea of capability-based security, we propose effect capabilities as an effective and flexible manner to control monadic effects and their interferences. Capabilities can be selectively shared between modules to establish secure effect-centric coordination. We further refine capabilities with type-based permission lattices to allow fine-grained decomposition of authority. We provide an implementation of effect capabilities in Haskell, using type classes to establish a way to statically share capabilities between modules, as well as to check proper access permissions to effects at compile time. We exemplify how to tame effect interferences using effect capabilities, by treating state and exceptions.

1 Introduction

Computational effects (e.g. state, I/O, and exceptions) complicate reasoning about, maintaining, and evolving software. Even though imperative languages embrace side effects, they generally provide linguistic means to control the potential for effect interference by enforcing some forms of encapsulation. For instance, the private attributes of a mutable object are only accessible to the object itself or its closely-related peers. Similarly, the stack discipline of exception handling makes it possible for a procedure to hide exceptions raised by internal computation, and thereby protect it from unwanted interference from parties that are not directly involved in the computation.

We observe that all these approaches are hierarchical, using module/package nesting, class/object nesting, inheritance, or the call stack as the basis for confining the overall scope of effects. This hierarchical discipline is sometimes inappropriate, either too loose or too rigid. Consequently, a number of mechanisms that make it possible to either cut across or refine hierarchical boundaries have been devised. A typical example mechanism for loosening the hierarchical constraints is friendship declarations in C++. Exception handling in Standard ML—with the use of dynamic classification [7] to prevent unintended access to exception values—is an example of a mechanism that strengthens the protection offered by the hierarchical stack discipline.

\* Funded by a CONICYT-Chile Doctoral Scholarship.
\*\* Partially funded by FONDECYT project 1110051.
Exploiting the intuitive affinity between encapsulation mechanisms and access control security, we can see classical approaches to side effect encapsulation as corresponding to hierarchical protection domains. The effective alternative in the security community to transcend hierarchical barriers is capability-based security, in which authority is granted selectively by communicating unforgeable tokens named capabilities \cite{11,13}. Seen in this light, the destructor of an exception value type in Standard ML is a capability that grants authority to inspect the internals of values of this type \cite{6}. The destructor, as a first-class value itself, can be flexibly passed around to the intended parties. Friendship declarations in C++ can also be seen as a static capability-passing mechanism.

Following this intuition we propose effect capabilities, in the context of Haskell\footnote{Implementation on the GHC compiler available online at: \url{http://pleiad.cl/effectcaps}}, for flexibly and securely handling computational effects. Effect capabilities are first-class unforgeable values that can be passed around in order to establish secure effect-related interaction channels. The prime focus of effect capabilities is to guarantee, through the type system, that there is no unauthorized access to a given effectful operation. Authorization is initially granted through static channel sharing at the module level, allowing detection of violations at compile time. We do not focus on dynamic sharing of capabilities, because this can only be done by modules that were already trusted at compile time.

We start illustrating the main problem addressed by effect capabilities in Haskell: the issue of effect interference in the monad stack (Section 2). Then we present the main technical development: a generic framework for capabilities and permissions, which can be statically shared between modules (Section 3). In this framework we combine several existing techniques, along with two novel technical contributions. First, a user-definable lattice-based permission mechanism that checks access at compile time using type class resolution (Section 3.2). And second, a static secret sharing mechanism implemented using type classes and mutually recursive modules (Section 3.3). Finally, effect capabilities are implemented using this framework in the particular case of monadic operations (Section 4), and we illustrate how to implement private and shared state (Section 4.1 and Section 4.2) as well as protected exceptions (Section 4.3).

## 2 Effect Interference in Monadic Programming

In this section we illustrate the problem of effect interference in monadic programming. We start with a brief description of monadic programming in Haskell (Section 2.1). Then we illustrate the particular issue of state interference (Section 2.2), also showing that the currently accepted workaround is not scalable (Section 2.3). Finally, we illustrate the issue of exception interference (Section 2.4).

### 2.1 Monadic Programming in a Nutshell

Monads\cite{15,25} are the mechanism of choice to embed and reason about computational effects such as state, I/O or exception handling, in purely functional languages like