Effect Capabilities For Haskell

Ismael Figueroa, Nicolas Tabareau, Éric Tanter

Effect Capabilities For Haskell

Ismael Figueroa[⋆1,2], Nicolas Tabareau[1], and Éric Tanter[⋆⋆2]

[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.

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

⋆ Funded by a CONICYT-Chile Doctoral Scholarship.
⋆⋆ Partially funded by FONDECYT project 1110051.

granted selectively by communicating unforgeable tokens named capabilities [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 [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[3], 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 [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 Haskell. Using monad transformers [12] it is possible to modularly create a monad that combines several effects. A monad transformer is a type constructor used to create a *monad stack* where each layer represents an effect. Monadic programming in Haskell revolves around the standard MTL library, which provides a set of monad transformers that can flexibly be composed together. Typically a monad stack has either the $Identity$, or the IO monad at its bottom. When using monad transformers it is necessary to establish a mechanism to access the effects of each layer. We now briefly describe current mechanisms; for a detailed description see [21].

[3] Implementation on the GHC compiler available online at: http://pleiad.cl/effectcaps

Explicit Lifting. A monad transformer t must define the *lift* operation, which takes a computation from the underlying monad m, with type $m\ a$, into a computation in the transformed monad, with type $t\ \ m\ \ a$. Explicit uses of *lift* directly determine which layer of the stack is being used.

Implicit Lifting. To avoid explicit uses of *lift*, one can associate a type class with each particular effect, defining a public interface for effect-related operations. Using the type class resolution mechanism, the monadic operations are routed to the first layer of the monad stack that satisfies a given class constraint. This is the mechanism used in the transformers from MTL, where the implicit liftings between them are predefined.

Tagged Monads. In this mechanism the layers of the monad stack are marked using type-level tags. The tags are used to improve implicit lifting, in order to route operations to specifically-tagged layers, rather than the first layer that satisfies a constraint [16,23,21]. In this work we focus only on the standard lifting mechanisms, which underlie the implementations of tagged monads, and leave for future work the integration of type-level tags and effect capabilities.

2.2 State Interference

As a running example to illustrate the issue of effect interference as well as its solution using effect capabilities, we consider the implementation of two monadic abstract data types (ADTs). These are a queue of integer values, with operations $enqueue$ and $dequeue$; and a stack, also of integer values, with operations $push$ and pop.

Regarding state, ideally each ADT should have a private state that cannot be modified by components external to the module. Before we describe the implementation, let us recall the standard state transformer and its associated type class:

$$\begin{aligned}
&\mathbf{newtype}\ StateT\ s\ m\ a = StateT\ (s \to m\ (a, s)) \\
&\mathbf{class}\ Monad\ m \Rightarrow MonadState\ s\ m \mid m \to s\ \mathbf{where} \\
&\quad get :: m\ s \\
&\quad put :: s \to m\ ()
\end{aligned}$$

A typical and reusable implementation of these ADTs is defined using implicit lifting. A straightforward implementation of the structures' operations is as follows:

$$\begin{aligned}
&enqueue_1 :: MonadState\ [Int]\ m \Rightarrow Int \to m\ () \\
&enqueue_1\ n = \mathbf{do}\ \{\, queue \leftarrow get; put\ \$\ queue +\!\!+ [n] \,\} \\
&dequeue_1 :: MonadState\ [Int]\ m \Rightarrow m\ Int \\
&dequeue_1 \quad = \mathbf{do}\ \{\, queue \leftarrow get; put\ \$\ tail\ queue; return\ \$\ head\ queue \,\} \\
&push_1 :: MonadState\ [Int]\ m \Rightarrow Int \to m\ () \\
&push_1\ n = \mathbf{do}\ \{\, stack \leftarrow get; put\ (n : stack) \,\} \\
&pop_1 :: MonadState\ [Int]\ m \Rightarrow m\ Int \\
&pop_1 \quad = \mathbf{do}\ \{\, stack \leftarrow get; put\ \$\ tail\ stack; return\ \$\ head\ stack \,\}
\end{aligned}$$

Thanks to implicit lifting, the functions can be evaluated in any monad stack m that fulfills the $MonadState\ [Int]$ constraint. With the intent of giving each ADT its own private state, we define a monad stack M with two state layers.

$$\mathbf{type}\ M = StateT\ [Int]\ (StateT\ [Int]\ Identity)$$

However, using both ADTs in the same program leads to state interference. The problem is that implicit lifting will route both $enqueue$ and $push$ operations to the first layer of M. For example, evaluating:

$$
\begin{aligned}
client_1 = \mathbf{do}\ & push_1\ 1 \\
& enqueue_1\ 2 \quad \text{-- value is put into the state layer used by the stack} \\
& x \leftarrow pop_1 \\
& y \leftarrow pop_1 \quad \text{-- should raise error because stack should be empty} \\
& return\ (x + y)
\end{aligned}
$$

yields 3 instead of throwing an error when attempting to pop_1 the empty stack. To address this issue, one of the ADTs must use explicit lifting to use the second state layer, for instance we can modify the queue operations:

$$
\begin{aligned}
&enqueue'_1\ n = \mathbf{do}\ \{\, queue \leftarrow lift\ get; (lift \circ put)\ (queue +\!\!+ [n])\,\} \\
&dequeue'_1\ n = \mathbf{do}\ \{\, queue \leftarrow lift\ get; (lift \circ put)\ (tail\ queue); return\ (head\ queue)\,\}
\end{aligned}
$$

However, as discussed by Schrijvers and Oliveira [21], this solution is still unsatisfactory. First, the approach is *fragile* because the number of *lift* operations is tightly coupled to the particular monad stack used, thus hampering modularity and reusability. And second, because the monad stack is *transparent*, meaning that nothing prevents $enqueue'_1$ or $dequeue'_1$ to use get and put operations that are performed on the first state layer. Conversely, nothing prevents $push_1$ or pop_1 from accessing the second state layer. In fact, any monadic component can modify the internal state of these structures.

2.3 State Encapsulation Pattern

To the best of our knowledge, the current practice to implement private state in Haskell—in order to avoid issues like the one above—is to define a custom state-like monad transformer and hide its data constructor. For instance, a polymorphic queue ADT can be implemented based on a new $QueueT$ monad transformer, which reuses the implementation of $StateT$ to represent the queue as a list of values:[4]

$$\mathbf{newtype}\ QueueT\ s\ m\ a = QueueT\ (StateT\ [s]\ m\ a)\ \mathbf{deriving}\ ...$$

The definitions of $enqueue$ and $dequeue$ are similar to those already presented, but let us consider their types:

$$
\begin{aligned}
&enqueue_2 :: s \rightarrow QueueT\ s\ m\ () \\
&dequeue_2 :: QueueT\ s\ m\ s
\end{aligned}
$$

Because these definitions are tied specifically to a monad stack where $QueueT$ (resp. $StackT$) is on top, another requirement to integrate with implicit lifting is to declare a new type class $MonadQueue$ (resp. $MonadStack$), whose canonical instance is given by $QueueT$ (resp. $StackT$).

In short, a $Queue$ module that encapsulates its state can be defined as:

[4] We do not show it here, but we use the $GeneralizedNewtypeDeriving$ extension of GHC to derive the necessary instances of the $Monad$ and $MonadTrans$ type classes.

```
module Queue (QueueT (), MonadQueue (..), enqueue, dequeue) where
newtype QueueT s m a = QueueT (StateT [s] m a) deriving ...
class Monad m ⇒ MonadQueue s m where
  enq :: s → m ()
  deq :: m s
instance Monad m ⇒ MonadQueue s (QueueT s m) where
  enq s = QueueT $ StateT $ λq → return ((), q ++ [s])
  deq   = QueueT $ StateT $ λq → return (head q, tail q)
enqueue :: (MonadQueue [Int] m) ⇒ Int → m ()
enqueue = enq
dequeue :: (MonadQueue [Int] m) ⇒ m Int
dequeue = deq
```

Declaring $QueueT$ as instance of $MonadQueue$ requires the implementation of enq and deq. As $QueueT$ relies on the standard state transformer $StateT$, the implementation is straightforward. The crucial point to ensure proper encapsulation is that *the module does not export the $QueueT$ data constructor*. This is explicit in the module signature as $QueueT$ (), which means that only the type $QueueT$ is exported, but its data constructors remain private.

Avoiding interference. Using the $QueueT$ and $StackT$ transformers, as well as the $MonadQueue$ and $MonadStack$ type classes defined using this pattern,we can rephrase our previous example in order to avoid state interference:

```
import Queue
type M = QueueT Int (StackT Int Identity)
client₂ = do push 1
             enqueue 2
             x ← pop
             y ← pop    -- error: popping from empty stack
             return (x + y)
```

Scalability Issues. The main issue of the state encapsulation pattern is that it is not scalable. To properly integrate $MonadQueue$ and $MonadStack$ with implicit lifting, we would need to declare $QueueT$ and $StackT$ as an instance of every other effect-related type class, and to make every other monad transformer an instance of $MonadQueue$ and $MonadStack$ as well. If we consider only the 7 standard transformers in the MTL, this effort amounts to 28 instance declarations! (14 instances for each encapsulated state) Moreover, when using non-standard transformers, it may not be possible to anticipate all the required combinations; therefore the burden lies on the user of such libraries to fill in the gaps. We are not the first to note the quadratic growth of instance declarations with this approach, *e.g.* Hughes dismisses monads as an option to implement global variables in Haskell for this very reason [8].

2.4 Exception Interference

Another form of effect interference can occurr in a program that uses exceptions and exception handlers. The problem is that due to the dynamic nature of exceptions and handlers, it is possible for exceptions to be inadvertently caught by unintended handlers—

for instance, by "catch-all" handlers. As an illustration, consider an application where the queue is used by a $consume$ function.

$$
\begin{aligned}
&consume :: (MonadQueue\ Int\ m, MonadError\ String\ m) \Rightarrow m\ Int \\
&consume = \mathbf{do}\ x \leftarrow dequeue \\
&\qquad\qquad \mathbf{if}\ (x < 0)\ \mathbf{then}\ throwError\ \text{“}Process\ error\text{”} \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \ return\ x
\end{aligned}
$$

This function checks an invariant that values should be positive, and throws an exception otherwise. Further assume that another $process$ function relies on $consume$.

$$
\begin{aligned}
&process :: (MonadQueue\ Int\ m, MonadError\ String\ m) \Rightarrow Int \rightarrow m\ Int \\
&process\ val = consume\ \text{`}catchError\text{`}\ (\lambda e \rightarrow return\ val)
\end{aligned}
$$

Here $process$ uses an exception handler, $catchError$, to get a default value val whenever $consume$'s invariant does not hold. Consider now a variant of $dequeue$ that raises an exception when trying to retrieve a value from an empty queue. Its type is

$$dequeue :: (MonadQueue\ s\ m, MonadError\ String\ m) \Rightarrow m\ s$$

In this scenario, exception interference will occur because the same exception effect is used to signal two different issues. Consider the following program:

$$
\begin{aligned}
&program_1 = \mathbf{do}\ enqueue\ (-10) \\
&\qquad\qquad process\ 23
\end{aligned}
$$

When evaluated, $program_1$ yields 23 because the value in the queue breaks the invariant of $consume$, triggering the handler of $process$. Now, consider a second program:

$$program_2 = process\ 23$$

which will also yield 23, but because the queue was empty—not because the invariant of $consume$ was broken. In this setting, it is not possible to assert non-emptiness of the queue, because exceptions get "swallowed" by another handler. Similar to state interference, current solutions rely on custom exception transformers and explicit lifting.

As argued by Harper [6], the standard semantics of exceptions difficult the modular composition of programs because of the potentially modified exception flows. Indeed, issues like this has been identified in the context of aspect-oriented programming [3]. In Section 4.3 we show how effect capabilities allows us to define exceptions that, like in Standard ML, can be protected from unwanted interception.

3 A Generic Static Framework for Capabilities and Permissions

This section presents the main technical development of this work: a generic framework for capabilities, upon which effect capabilities are built, in the next section. First, we define capability-based access as a computational effect (Section 3.1). Then, we refine simple capabilities with type-based and user-definable permission lattices (Section 3.2); and show how capabilities can be shared between modules (Section 3.3). Finally we describe how the framework supports two key features of capabilities-based mechanisms: delegation and attenuability (Section 3.4).

3.1 Private Capabilities as a Computational Effect

A private capability is a singleton type whose type is public but whose constructor is private. For instance, consider the capability for read/write access to some state:

$$\textbf{data}\ RWCap = RWCap$$

We turn this capability into a notion of *protected computations* by using a specific reader monad transformer for capabilities, $CapT$. Using the reader transformer allows us to embed the actual capability used to run a computation into the read-only environment bound to a reader monad. Similar to state encapsulation, $CapT$ is defined in terms of the canonical reader monad transformer $ReaderT$.

$$\textbf{newtype}\ CapT\ c\ m\ a = CapT\ (ReaderT\ c\ m\ a)\ \textbf{deriving}\ ...$$
$$fromCapT :: c \to CapT\ c\ m\ a \to m\ a$$
$$fromCapT\ !\ c\ (CapT\ ma) = runReaderT\ ma\ c$$

A capability has a public type but a private value, but as Haskell is lazy, a malicious module can always forge a capability for which it has no access by passing $\perp$ as the capability argument to evaluate $fromCapT$.[5] To avoid this situation, we use a strictness annotation ! in the implementation of $fromCapT$. Note this issue would not be present in a strict setting. As an example, consider a module A that uses $RWCap$ to restrict access to a state monad holding a value of type s.

$$\textbf{module}\ A\ (getp, putp, RWCap\ ())\ \textbf{where}$$
$$\textbf{data}\ RWCap = RWCap$$
$$getp :: CapT\ RWCap\ (State\ s)\ s$$
$$putp :: s \to CapT\ RWCap\ (State\ s)\ ()$$

A module B that imports A will get access to both operations, but will not be able to perform any of them because it will lack the $RWCap$ value, which can only be constructed in the context of module A.

3.2 Private Lattice of Permissions

Capabilities are unforgeable authority tokens that unlock specific monadic operations. Ideally, a system should follow the *principle of least privilege* [18], which in our context means that it should not be necessary to have write permissions just to read the value of a state monad; and conversely, reading access is not necessary to update such a state.

We now refine the model of capabilities with the possibility to attach *permissions* to a capability, in order to allow a finer-grained decomposition of authority. A permission denotes the subset of operations that the capability permits. Now capabilities are defined as type constructors with a single argument, the permission; and permissions are defined as singleton types.

Permission Lattices. Permissions can be organized in a lattice specified by a $\sqsupset$ type class. $\sqsupset$ is a simple reflexive and transitive relation on types defined as:[6]

$$\textbf{class}\quad a \sqsupset b$$
$$\textbf{instance}\ a \sqsupset a\quad \text{-- generic instance for reflexivity}$$

The type class $\sqsupset$ must not be public because it would allow a malicious user to add a new undesirable relation in the lattice to effectively bypass permission checking

[5] $\perp$ is an expression that pertains to all types, and directly fails with an error if evaluated. Hence, it can pass as any capability, fulfilling the expected type of $fromCapT$.

[6] Unlike logic programming, transitivity cannot be deduced from a generic instance, due to an ambiguity issue during type class resolution; hence all pairs of the relation must be explicit.

altogether. Still, we want to be able to impose constraints based on the private lattice in other modules. To do that, we define a public lattice $\supset$ that exports the private lattice without being updatable from the outside of the module, because extending it always requires to define an instance on the private lattice.[7]

```
class    a ⊐ b ⇒ a ⊃ b
instance a ⊐ b ⇒ a ⊃ b
```

Permission Lattices in Practice. Going back to the previous example, we now define the $RWCap$ capability, as well as the $ReadPerm$, $WritePerm$ and $RWPerm$ permissions, denoting read-only, write-only and read-write access, respectively. We also define the private and public permission lattices $\sqsupset_{RW}$ and $\supset_{RW}$, for state access permissions:

```
module RWLattice (⊃_RW, RWCap (), ReadPerm, WritePerm, RWPerm) where
data RWCap p   = RWCap p
data ReadPerm  = ReadPerm
data WritePerm = WritePerm
data RWPerm    = RWPerm

  -- private lattice
class    a ⊐_RW b
instance a ⊐_RW b

  -- private instances, not updateable externally
instance RWPerm ⊐_RW ReadPerm
instance RWPerm ⊐_RW WritePerm

  -- public lattice
class    a ⊐_RW b ⇒ a ⊃_RW b
instance a ⊐_RW b ⇒ a ⊃_RW b
```

Crucially, we use module encapsulation to hide the private lattice $\sqsupset_{RW}$. Thus we only export the public lattice $\supset_{RW}$, which can be used as a regular class constraint. Otherwise, obtaining write-access from a read-only permission is as simple as:

```
module BypassRW where
import RWLattice
instance ReadPerm ⊐_RW WritePerm
```

Using the public permission lattice allows developers to impose fine-grained access constraints using the public type class $\supset_{RW}$. For instance, the functions $getp$ and $putp$ can be refined as:

```
getp :: perm ⊃_RW ReadPerm  ⇒ CapT (RWCap perm) (State s) s
putp :: perm ⊃_RW WritePerm ⇒ s → CapT (RWCap perm) (State s) ()
```

As a final remark, recall from Section 3.1 that type class resolution statically checks for proper permissions when a computation is evaluated using $fromCapT$.

Capabilities as namespaces for permissions. Capability constructors, such as $RWCap$, may appear superfluous, because we are interested in the permissions for protected operations. However, such constructors serve the crucial role of serving as namespaces

[7] Haskell type classes are *open*, that is, instances of publicly exported type classes can be added in any part of the system. Private type classes are confined to the module that defines them.

for permissions. This allows a module to have restricted read-only access to some state, while still having full read-write access to another state.

3.3 Static Sharing of Capabilities

We now describe how to go beyond private capabilities and support the ability to allow specific modules to have access to capabilities. The issue addressed here is that most module systems, including that of Haskell, do not make it possible to expose bindings to explicitly-designated modules. For example, as we illustrate in Section 4.2, for efficiency reasons a $Queue$ module can provide read-only access to its internal state to a $PriorityQueue$ module, which simply acts as another interface on top of the queue.

Conceptually, the idea of static sharing is to use public accessors to selectively share capabilities. However this requires a trusted mechanism by which modules can be identified properly by the accessors. The development of this idea yields a mechanism for *static* message passing, using type classes, loosely inspired by the π-calculus notion of messages and channels [19].

Message sending as type class instances. In analogy with capabilities, a channel is just a singleton type whose type is public, but whose (unique) value is private. Channels are governed by the $Channel$ monad reader which prevents from the use of $\bot$:

$$\textbf{newtype}\ Channel\ ch\ a = Channel\ (Reader\ ch\ a)\ \textbf{deriving}\ ...$$
$$fromChannel :: ch \to Channel\ ch\ a \to a$$
$$fromChannel\,!\ ch\ (Channel\ ma) = runReader\ ma\ ch$$

We define a type class $Send$ for message sending:

$$\textbf{class}\ Send\ ch\ c\ p\ \textbf{where}$$
$$receive :: p \to Channel\ ch\ (c\ p)$$

This type class requires three types: a channel ch, a capability c, and a permission p; and it provides the $receive$ method. Sending a message of type $c\ p$ on ch amounts to declaring an instance $Send\ ch\ c\ p$. Conversely, receiving a message of type $c\ p$ on ch amounts to applying the function $receive$ to p and getting the value back using $fromChannel$.[8] Observe that the messaging protocol is rather asymmetric, because capabilities are sent statically by declaring type classes instances, but are received dynamically by calling $receive$. This is not problematic because type class resolution will check that all calls to $receive$ are backed up by an instance of $Send$, or else typechecking will fail. Therefore, the protocol ensures that one module can only receive a message that has been sent to it. [9]

For instance, following the motivation example, the $Queue$ module can send the $RWCap$ capability with read permission to the $PQChan$ channel provided by the $PriorityQueue$ module (full example in Section 4.2):

$$\textbf{instance}\ Send\ PQChan\ RWCap\ ReadPerm$$
$$\textbf{where}\ receive\ ReadPerm = return\ \$\ RWCap\ ReadPerm$$

[8] The expected result type $c\ p$ has to be provided explicitly, because messages of different types can be sent on the same channel.

[9] We rely on GHC support for mutually recursive modules for inter-module communication. See http://www.haskell.org/ghc/docs/latest/html/users_guide/separate-compilation.html

```
module A where
import B   -- to send capability to BChannel
instance Send BChannel RWCap ReadPerm where
  receive ReadPerm = return $ RWCap ReadPerm
module B where
import A   -- to get the capability sent by A
import C   -- to send capability to CChannel
data BChannel = BChannel
instance Send CChannel RWCap ReadPerm where
  receive ReadPerm = return $ (fromChannel BChannel $ receive ReadPerm)
module C where
import B   -- to get the capability sent by B
data CChannel = CChannel
cap :: RWCap ReadPerm
cap = fromChannel CChannel $ receive ReadPerm
```

Fig. 1: Static delegation of capabilities

Of course, this mechanism is less expressive than message passing in process calculi—only one message of type $c\ p$ can be sent on a specific channel—but it is sufficient for our purposes since permissions are singleton types.

3.4 Delegation and Attenuability

Capabilities-based mechanisms feature two characteristics called *delegation* and *attenuability* [11]. In combination, these characteristics allow an entity to transmit (a restricted version of) its capabilities to another entity in the system. We describe how these characteristics are supported in the framework.

Delegation. The sharing mechanism allows for static delegation of capabilities. A module B that receives a capability from other module A, can in turn transmit the capability to another module C. This is sound because B cannot transmit more capabilities than those it receives from A. Figure 1 shows static delegation of the $RWCap\ \ ReadPerm$ capability.

Attenuability. A capability with a high permission in a permission lattice can be attenuated into another capability with a lower permission implied by the former. To support attenuability, we force capabilities to define a function $attenuate$ using the type class:

```
class Capability c ⊃ | c → ⊃ where
  attenuate :: p1 ⊃ p2 ⇒ c p1 → p2 → c p2
```

Here $attenuate$ degrades the permission if it respects the lattice structure of $\supset$. If module A needs to provide a limited version of a capability to module B it can provide a sub-permission based on the existing permission lattice using the function $attenuate$. Note that $\supset$ is a parameter of the class because different capabilities may be defined on different lattices, but the functional dependency $c \rightarrow \supset$ imposes that only one lattice is attached to a capability. If the required permission is not already provided by the existing lattice, one can always define a refined lattice (and redefine associated functions).

```
class Monad m ⇒ MonadStateP c s m | m → s where
  getp :: (Capability c ⊃RW, p ⊃RW ReadPerm)  ⇒ CapT (c p) m s
  putp :: (Capability c ⊃RW, p ⊃RW WritePerm) ⇒ s → CapT (c p) m ()
newtype StateTP c s m a = StateTP (StateT s m a) deriving ...
instance Monad m ⇒ MonadStateP c s (StateTP (c ()) s m) where
  getp   = lift ∘ StateTP $ get
  putp   = lift ∘ StateTP ∘ put
```

Fig. 2: Protected versions of the monad state type class and state monad transformer.

4 Effect Capabilities: Upgrading Monads with Capabilities

We now delve into the main subject of this work: how to use capabilities to control monadic effects and their interferences in an effective and flexible manner. Building upon the generic capabilities framework, which can be used to restrict access to arbitrary monadic operations, the essential idea of effect capabilities is to secure the operations of the layers in the monad stack using capabilities.

Concretely, this means that we define protected versions of monad transformers, and of the type classes associated to their effects, in which all the monadic operations are wrapped by the $CapT$ monad transformer. This way, while an external component can still access any layer of the monad stack using explicit lifting, it will not be able to perform operations on them unless it can present the required capability.

In particular, we define protected versions of the state and exception MTL transformers and their associated type classes. As a naming convention we append the P suffix to the name of the protected monad transformers and type classes. We now illustrate how to implement private and shared state (Section 4.1 and Section 4.2) and protected exceptions (Section 4.3).

4.1 Private Persistent State

Based on the state permission lattice (Section 3.2), we define the protected versions of the state monad transformer and corresponding type class (Figure 2). To use the $getp$ function, one needs to have a capability c that implies the $ReadPerm$ read permission; and dually to use the $putp$ function, one needs the capability that implies the $WritePerm$ write permission.

To illustrate, consider the following polymorphic $Queue$ using private state:

```
module Queue (enqueue, dequeue, QState ()) where
data QState p = QState p
instance Capability QState ⊃RW where
  attenuate (QState _) perm = QState perm
enqueue :: MonadStateP QState [s] m ⇒ s → m ()
enqueue s = do queue ← fromCapT (QState ReadPerm) getp
               fromCapT (QState WritePerm) $ putp (queue ++ [s])
dequeue :: MonadStateP QState [s] m ⇒ m s
dequeue   = do queue ← fromCapT (QState ReadPerm) getp
               fromCapT (QState WritePerm) $ putp (tail queue)
               return (head queue)
```

Thanks to the use of the $QState$ capability and the secure $MonadStateP$ class, the internal state of the queue is private to the $Queue$ module. Since the $QState$ data constructor is not exported, external access is prevented—even if explicit lifting can be used to access the respective instance of $MonadStateP$, it cannot be used to perform any monadic operation on it because the proper capability is required. We still require to export $QState$ as a type, in order to create a suitable monad stack, *e.g.* to instantiate an integer queue:

$$\textbf{type}\ M = StateTP\ (QState\ ())\ [Int]\ Identity$$

To construct a monad stack we are only interested on the capability type, but not in any particular permission—however permissions will still be checked statically as required for each operation—hence we use $()$ as the permission type in the definition of M.

4.2 Shared Persistent State

We now illustrate capability sharing with shared persistent state. We define a module $PriorityQueue$ that adds a notion of priority on top of $Queue$. In a priority queue one can access directly the most recent element having a high priority, using the $peekBy$ function. For efficiency, the $PriorityQueue$ module needs direct access to the internal state of the queue. As we do not want to do this by publicly exposing the capability $QState$, we send the capability on the channel $PQChan$ provided by $PriorityQueue$.

```
module Queue (enqueue, dequeue, QState ()) where
import PriorityQueue   -- to get PQChan channel
newtype QState p = QState p
instance Capability QState ⊃RW where
  attenuate (QState _) perm = QState perm
instance Send PQChan QState ReadPerm where
  receive perm = return $ QState perm
  -- enqueue and dequeue operations as before
```

The implementation of $PriorityQueue$ is as follows:

```
module PriorityQueue (PQChan (), peekBy) where
import Queue
data PQChan = PQChan
queueState :: QState ReadPerm
queueState = fromChannel PQChan $ receive ReadPerm
peekBy :: (Ord s, MonadStateP QState [s] m) ⇒ (s → s → Ordering) → m (Maybe s)
peekBy comp = do queue ← fromCapT queueState getp
                 if null queue then return Nothing
                               else return (Just $ maximumBy comp queue)
```

To use the internal state of the queue, the $PriorityQueue$ module imports $Queue$, defines and exports its channel $PQChan$, and retrieves the capability $QState$ with the read-only permission, as prescribed by $Queue$. $peekBy$ can access the internal state of the queue by using the $queueState$ capability and $fromCapT$.

```
class (Monad m, Error e) ⇒ MonadErrorP c e m | c m → e where
  throwErrorp :: perm ⊃Ex ThrowPerm ⇒ e → CapT (c perm) m a
  catchErrorp :: perm ⊃Ex CatchPerm ⇒ m a → (e → m a) → CapT (c perm) m a
newtype ErrorTP c e m a = ErrorTP {runETP :: ErrorT e m a} deriving ...
instance (Monad m, Error e) ⇒ MonadErrorP c e (ErrorTP (c ()) e m) where
  throwErrorp       = lift ∘ ErrorTP ∘ throwError
  catchErrorp m h = lift ∘ ErrorTP $ catchError (runETP m) (runETP ∘ h)
```

Fig. 3: Protected versions of the monad error type class and error monad transformer

4.3 Protected Exceptions

Exception handling may be seen as a communication between two modules, one that raises an exception, and one that handles it. For correctness or security reasons, we may wish to ensure that a raised exception can only be handled by specific modules. Protecting exception handling can be achieved using exception capabilities, similar to how state capabilities control shared state. First, we define the private and public lattices, $\sqsupset_{Ex}$ and $\supset_{Ex}$, as permissions for throwing and catching exceptions:

```
module ExLattice (⊃Ex, ThrowPerm, CatchPerm, TCPerm) where
data ThrowPerm = ThrowPerm
data CatchPerm = CatchPerm
data TCPerm    = TCPerm
  -- private lattice
class      a ⊐Ex b
instance a ⊐Ex a
  -- private instances
instance TCPerm ⊐Ex ThrowPerm
instance TCPerm ⊐Ex CatchPerm
  -- public lattice
class      a ⊐Ex b ⇒ a ⊃Ex b
instance a ⊐Ex b ⇒ a ⊃Ex b
```

Then, in Figure 3 we define the protected versions of the standard $ErrorT$ monad transformer and $MonadError$ type class, following the same approach as for protected state. Going back to our running example, we can make $dequeue$ raise an exception when accessing an empty queue, in order to allow for recovery. Using a $QError$ exception capability we can control which modules are allowed to define their own handlers:

```
module Queue (enqueue, dequeue_Ex, QState (), QError ()) where
  -- QState definition, type class instances and enqueue as before ...
data QError p = QError p

instance Capability QError ⊃_Ex where
  attenuate (QError _) perm = QError perm

dequeue_Ex :: (MonadStateP QState [s] m, MonadErrorP QError String m) ⇒ m s
dequeue_Ex = do queue ← fromCapT (QState ReadPerm) getp
  if null queue then fromCapT (QError ThrowPerm) $ throwErrorp "Empty..."
                else do fromCapT (QState WritePerm) $ putp (tail queue)
                        return (head queue)
```

Recall from Section 2.4 the example of exception interference. Now the *consume* function can catch the exceptions it is interested in, while exceptions thrown by $dequeue_{Ex}$ will simply pass-through. Actually, it is not possible for *process* to catch those exceptions unless the *QError* capability is shared from the *Queue* module.

```
consume :: (MonadStateP QState [Int] m, MonadError String m,
            MonadErrorP QError String m) ⇒ m Int
consume = do x ← dequeue_Ex
             if (x < 0) then throwError "Process error"
                        else return x

process :: (MonadStateP QState [Int] m, MonadErrorP QError String m,
            MonadError String m) ⇒ Int → m Int
process val = consume `catchError` (λe → return val)
```

Consider a *debug* function in a module that has access to the *QError* capability with permission implying *CatchPerm*. Then, *debug* can define a custom handler:

```
debug val = fromCapT (QError CatchPerm) $
            process val `catchErrorp` (λe → error "...")
```

Finally, for cases where the client is not trusted and cannot catch the exception, we can export a function $dequeue_{Err}$, that reraises the error using the *QError* capability:

```
dequeue_Err :: (MonadStateP QState [s] m, MonadErrorP QError String m) ⇒ m s
dequeue_Err = fromCapT (QError CatchPerm) $ dequeue_Ex `catchErrorp` error
```

5 Related Work

Extensible effects (EE) [9] proposes an alternative representation of effects that is not based on monads or monad transformers, that still subsumes the MTL library by providing a similar API. EE presents a client-server architecture where an effectful operation is requested by client code and is then performed by a corresponding *handler*. The internal implementation of EE uses a continuation monad, *Eff*, to implement coroutines, along with a novel mechanism for extensible union types. An effectul value has type *Eff* r where r is a type-level representation, based on the novel union types, of the effects currently available; thus defining a type-and-effect system for Haskell. EE does not describes any mechanism for restricting access to effects. Any effect available in the type-level tracking of effects is available to any component. To add two copies of the same effect, the user is required to define a wrapper using a **newtype** declaration.

The *Effects* [1] library is a effect system implemented in the dependently-typed language Idris, based on algebraic effect handlers, also as an alternative to monads and monad transformers. Similar to EE, Effects keeps track of the available effects that can be used in a heterogeneous list. Performing an effectful operation requires a proof that the given effect is indeed available, but such proof is automatically generated if the effect is available. As with EE, Effects do not address the issue of controlling the access to effects. Any available effect can be used by any part of the system; and references to copies of a same effect are available (*e.g.*two integer states) are disambiguated using labels in the effect-tracking list.

Effect capabilities are orthogonal to the mechanism used to implement effects. We have shown how to apply them in the context of monad transformers as a solution to known interference issues, and indeed the same approach should be applicable to other approaches like EE and Effects.

6 Future Work

We identify several venus for future work. A first one, regarding safety, arises from the fact that we have ignored a number of Haskell features that defeat the integrity of the type system. For instance, module boundaries can be violated using Template Haskell or the $GeneralizedNewtypeDeriving$ language extension, or generic programming. Recently, Safe Haskell [24] has been proposed as an extension to Haskell, implemented in GHC (as of version 7.2). Safe Haskell protects referencial transparency and module boundaries by disabling the use of these unsafe features. Because the privacy of capabilities relies on effective module boundaries, we plan to integrate the effect capabilities library as an extension of Safe Haskell.

A second line of work aims to lower the amount of boilerplate code that is required, like the instances of $Capability$ and $Channel$ classes. This situation can be improved using *generic programming* (*e.g.*using the $GHC.Generics$ library), to provide default implementations for the $receive$ and $attenuate$ functions. This is already done in the downloadable implementation. A complementary approach is using TemplateHaskell [22], a template meta-programming facility for Haskell.

Another line of work concerns the integration of capabilities with tagged monads. In the model, each protected layer of the monad stack must be labeled with the capability namespace to which it is bound. This is similar to how a layer in a tagged monad setting must possess a tag in order to enable tag-directed type class resolution. The idea is to use the capability type constructor for both purposes at the same time; thus benefitting from the robustness with respect to the layout of the monad stack, provided by tagged monads, in addition to controlling access to each layer.

Finally, we are interested in studying effect capabilities for other effects, like nondeterminism, concurrency, continuations, and particularly I/O. Currently, access to I/O operations through the IO monad has no granularity. Using effect capabilities we can split access into several categories (*e.g.* file access, network access, etc.).

Acknowledgments This work was supported by the Inria Associated Team REAL.

References

1. E. Brady. Programming and reasoning with algebraic effects and dependent types. *ICFP'13*.

2. B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural exception analysis for Java. In *SAC 2001*.
3. R. Coelho, A. Rashid, A. Garcia, N. Cacho, U. Kulesza, A. Staa, and C. Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In *ECOOP 2008*.
4. M. Fluet and G. Morrisett. Monadic regions. *Journal of Functional Programming*, 2006.
5. D. K. Gifford and J. M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the ACM Conference on LISP and Functional Programming (LFP '86)*.
6. R. Harper. Exceptions are shared secrets. http://existentialtype.wordpress.com/, Dec. 2012.
7. R. Harper. *Practical Foundations for Programming Languages*. Cambridge U. Press, 2012.
8. J. Hughes. Global variables in Haskell. *Journal of Functional Programming*, 2004.
9. O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Haskell'13*.
10. J. Launchbury and S. Peyton Jones. Lazy functional state threads. *SIGPLAN Notices*, 1994.
11. H. M. Levy. *Capability-based computer systems*, volume 12. Digital Press Bedford, Massachusetts, 1984.
12. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters.In *POPL'95*.
13. M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, 2006.
14. R. Milner, R. Harper, D. MacQueen, and M. Tofte. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
15. E. Moggi. Notions of computation and monads. *Information and Computation*, 1991.
16. D. Piponi. Tagging monad transformer layers, 2010. http://blog.sigfpe.com/2010/02/tagging-monad-transformer-layers.html.
17. M. Robillard and G. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *TOSEM* 2003.
18. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems, 1975.
19. D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge U. Press, 2003.
20. C. F. Schaefer and G. N. Bundy. Static analysis of exception handling in Ada. *Software—Practice and Experience*, 23(10):1157–1174, Oct. 1993.
21. T. Schrijvers and B. C. Oliveira. Monads, zippers and views: virtualizing the monad stack. In *ICFP 2011*.
22. T. Sheard and S. P. Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, Dec. 2002.
23. M. Snyder and P. Alexander. Monad factory: type-indexed monads. In *Proceedings of the 11th International Conference on Trends in Functional Programming*, pages 198–213, 2010.
24. D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières. Safe Haskell. In *Haskell'12*.
25. P. Wadler. The essence of functional programming. In *POPL 92*.