Science of Computer Programming 119 (2016) 3–30

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Effect capabilities for Haskell: Taming effect interference in monadic programming

Ismael Figueroa[a,*], Nicolas Tabareau[b], Éric Tanter[c]

[a] *Escuela de Ingeniería Informática, Pontificia Universidad Católica de Valparaíso, Valparaíso, Chile*
[b] *Ascola group, INRIA, Nantes, France*
[c] *PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile*

A R T I C L E I N F O

Article history:
Received 13 March 2015
Received in revised form 19 November 2015
Accepted 22 November 2015
Available online 28 November 2015

Keywords:
Monads
Capabilities
Computational effects
Tagged monads

A B S T R A C T

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 first exemplify how to tame effect interferences using effect capabilities by treating state and exceptions. Then we focus on taming I/O by proposing a fine-grained lattice of I/O permissions based on the current classification of its operations. Finally, we show that integrating effect capabilities with modern tag-based monadic mechanisms provides a practical, modular and safe mechanism for monadic programming in Haskell.

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

* Corresponding author.
E-mail addresses: ismael.figueroa@pucv.cl (I. Figueroa), nicolas.tabareau@inria.fr (N. Tabareau), etanter@dcc.uchile.cl (É. Tanter).

http://dx.doi.org/10.1016/j.scico.2015.11.010
0167-6423/© 2015 Elsevier B.V. All rights reserved.

declarations in C++. Exception handling in Standard ML — with the use of dynamic classification [8] 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 granted selectively by communicating unforgeable tokens named capabilities [13,16]. 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 [7]. 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, for flexibly and securely handling computational effects. The prime focus of effect capabilities is to statically guarantee, through the type system, that there is no unauthorized access to a given effectful operation. The approach combines capabilities at the type level as static restrictions on operation, and capabilities at runtime as first-class unforgeable *tokens* that must be presented to run a protected operation and that can be passed around in order to establish secure effect-related interaction channels. 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, as this can only be done by modules that were already trusted at compile time. The implementation of effect capabilities in the GHC compiler is available online at http://pleiad.cl/effectcaps.

1.1. Contributions

This work features two main contributions: the first is a generic framework for capabilities and permissions, which can be statically shared between modules. The second, is the application of this framework in the context of monadic programming in order to provide protected effectful computations. More specifically, we present several technical contributions:

- The definition of capabilities and *protected computations* as a computational effect based on the capability transformer *CapT*. A computation of type *CapT c m a* can only be evaluated by presenting a runtime value of capability type *c*. Hence, the type system, without any modification, ensures that there is no unauthorized access to protected computations.
- A user-definable and user-extensible encoding of permission lattices to be used alongside capabilities. This encoding uses type classes and *closed type families*, a recent feature of the GHC Haskell compiler.
- A static secret sharing mechanism implemented using type classes and mutually recursive modules. Any module can declare a *channel* that can be used by other modules to send values specifically to that channel. A value is sent by declaring an instance of the *Send* type class on the proper channel, while receiving a value requires calling the *receive* function, the only operation of the *Send* class. All calls to *receive* must be backed by a *Send* instance declaration, otherwise type checking fails.
- A set of capability- and permission-observing monad transformers and related type classes, based on the design of the standard monad transformers, which can be flexibly composed together without effect interference. In particular we implement state, exceptions and I/O effects.
- Regarding I/O, we propose a fine-grained lattice of permissions based on the current classification given in the Haskell documentation. This is a first step towards a practical lattice for permission-based I/O in Haskell.
- The integration of effect capabilities with the recent approach of *tagged monads*. Specifically, we combine the benefits of *monad views*, a mechanism that is robust to layout changes in the composition of monad transformers, with the security guarantees of effect capabilities. We show how this mechanism allows developers to implement generic imperative abstract data types.

1.2. Effect capabilities by example

Before delving into all the technical developments of this work we present a simplified example of what is like to program using effect capabilities. The example consists of the implementation of a queue data structure using an internal state which is made private using an effect capability. Then, we implement a priority queue to which is granted read-only access to the internal state of the queue, using the static sharing mechanism. At this point the goal is to provide an overall intuition of how capabilities are declared, used, and shared rather than a fully detailed example. The full example with all the technical details is shown in Section 4.3.

A higher-level overview of the situation is depicted in Fig. 1. There are two modules: *Queue* and *PriorityQueue*. Module *Queue* sends a capability value *QState ReadPerm* to the *PriorityQueue* module. The capability is sent on a channel *PQChan*. The diagram also shows that both modules are mutually dependent. The *Queue* needs to import the *PriorityQueue* to use the *PQChan* channel defined on that module, and conversely, the *PriorityQueue* needs to import *Queue* to receive the message.

Fig. 2 shows snippets of the implementation of both modules. In module *Queue* we see that *QState* is a regular datatype with a single constructor which takes a permission *p* as argument. *QState* is registered as a capability by declaring an instance of the *Capability* type class. Crucially, notice that the *QState* data constructor *is not exported by the module*, otherwise every other module would have access to the state of the queue. The module imports *PriorityQueue*, thus having access

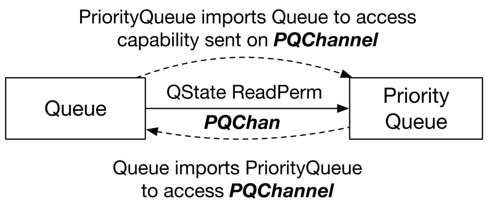
PriorityQueue imports Queue to access capability sent on PQChannel
Queue
QState ReadPerm
PQChan
Priority Queue
Queue imports PriorityQueue to access PQChannel

Fig. 1. Communication and mutual dependency of *Queue* and *PriorityQueue* modules.

```
module Queue (QState (), ...) where   -- QState constructor is private
import PriorityQueue
  -- capability for queue state
data QState p = QState p
instance Capability QState ⊐RW
  -- send read-only permission to priority queue module
instance Send PQChan QState ReadPerm
enqueue s = do queue ← getp `withCapability` (QState ReadPerm)
               putp (queue ++ [s]) `withCapability` (QState WritePerm)
...
```

```
module PriorityQueue (...) where
import Queue
data PQChan = PQChan
  -- receive read-only permission from queue module
queueCap :: QState ReadPerm
queueCap = fromChannel PQChan $ receive ReadPerm
peekBy comp = do queue ← getp `withCapability` queueCap
                 if null queue then return Nothing
                               else return (Just $ maximumBy comp queue)
...
```

Fig. 2. Code snippets for modules *Queue* and *PriorityQueue*.

to the channel *PQChan*. The instance declaration **instance** *Send PQChan QState ReadPerm* denotes that capability *QState* with permission *ReadPerm* is send over channel *PQChan*. The figure also shows the implementation of the *enqueue* operation, which uses the protected operations *getp* and *putp* to read and write to the protected state, respectively. Both operations need to provide the proper *QState* capability, and this is done using the *withCapability* function.

On the other hand, the *PriorityQueue* module declares the *PQChan* type, which serves as the name of the communication channel with *Queue*. By importing *Queue* the declared instance of *Send* is put into the scope of this module. Using *fromChannel PQChan* $ *receive ReadPerm* retrieves the capability sent by the *Queue* module. The capability is assigned to the *queueCap* identifier, and is then used in the implementation of the *peekBy* operation. This function also uses *getp* and presents the corresponding capability using *withCapability*.

This example illustrates the fundamental concepts, design and usage of effect capabilities. Capabilities are defined as simple datatypes with a single constructor — that must be kept private — and the capability type must be declared as instance of the *Capability* type class. Capabilities are shared by declaring instances of the *Send* type class. A module desiring to receive a capability must export a channel, which will be used in the *Send* instances, thus introducing a mutual dependency between the involved modules. Finally, protected operations, like *getp* and *putp*, are similar to standard monadic operations such as *get* or *put*, but must present a capability using *withCapability*.

1.3. Outline

The rest of this article is structured as follows: first, we briefly review the essential concepts of monadic programming in Haskell, including plain monads and monad transformers (Section 2). Then, we illustrate the main problem addressed by effect capabilities in Haskell: the issue of *effect interference* in the monad stack (Section 3). After that we present the main technical development: a generic framework for capabilities and permissions and their static sharing mechanism (Section 4). Then, effect capabilities are implemented using this framework in the particular case of monadic operations (Section 5). We first illustrate the use of effect capabilities to implement private and shared state (Section 5.1 and Section 5.2), as well as protected exceptions (Section 5.3). In addition, we illustrate how to tame I/O access using a fine-grained lattice of permissions (Section 5.4). Finally, we show that integrating effect capabilities with modern tag-based monadic libraries produces a practical, modular, and secure mechanism for monadic programming in Haskell (Section 6).

2. Monadic programming in a nutshell

Monads [17,31] are a denotational approach to embed and reason about computational effects such as state, I/O or exception handling, in purely functional languages like Haskell. The purpose of this section is to serve as a brief background for readers that are familiar with Haskell but may not be experts in monadic programming. For a general introduction to Haskell we recommend the *Gentle Introduction to Haskell*,[1] the *Try Haskell*[2] website, and the excellent *Learn You a Haskell* book and website.[3] Readers already proficient in monadic programming in Haskell can safely skip this section.

2.1. A primer on monads

A monad is defined by a type constructor *m* and functions $\gg=$ (called *bind*) and *return*. At the type level a monad is a regular type constructor, although conceptually we distinguish a value of type *a* from a *computation in monad m* of type *m a*. Monads provide a uniform interface for computational effects, as specified in the *Monad* type class:

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

Here *return* promotes a value of type *a* into a computation of type *m a*, and $\gg=$ is a pipeline operator that takes a computation, extracts its value, and applies an action to produce a new computation. The precise meanings for *return* and $\gg=$ are specific to each monad. The computational effect of a monad is "hidden" in the definition of $\gg=$, which imposes a sequential evaluation where the effect is performed at each step. To avoid cluttering caused by using $\gg=$ Haskell provides the **do**-notation, which directly translates to chained applications of $\gg=$. The $x \leftarrow k$ expression binds identifier *x* with the value extracted from performing computation *k* for the rest of a **do** block. The simplest monad is the *identity monad*, which has no computational effect:

```
newtype Identity a = Identity a
instance Monad Identity where
  return a             = Identity a
  (Identity a) >>= f = f a
```

Perhaps the second simplest monad is the *Maybe monad*, which represents computations that may not yield a value:

```
newtype Maybe a = Just a | Nothing
instance Monad Maybe where
  return a = Just a
  m >>= f  = case m of
    Just x   → f x
    Nothing → Nothing
```

A computation of type *Maybe a* is either *Just* a value, or *Nothing*. Although this monad is very similar to *Identity*, the main difference lies in the $\gg=$ operation: the initial computation *m* is examined to see whether it contains a value or not. If it does, the operation *f* is then applied, otherwise the result keeps being *Nothing*. In other words, once a non-value is produced it is transparently kept until the end of the computation. This allows developers to write programs focusing in the *successful* control flow, leaving the propagation of non-values to the internal mechanisms of this monad.

Another prevalent monad is the *state monad*, whose computational effect is to thread a value with read-write access.

```
newtype State s a = State (s → State (a, s))
instance Monad (State s) where
  return a          = State $ λs → (a, s)
  (State h) >>= f = State $ λs → let (a, s')      = h s
                                      (State g) = f a
                                  in g s'
```

[1] http://www.haskell.org/tutorial.
[2] http://tryhaskell.org.
[3] http://learnyouahaskell.com.

A *State* computation is a function that requires an initial state *s* and yields a pair with a resulting value *a* and a potentially modified state, also of type *s*. To promote a value to a computation *return* creates a function that keeps whatever state *s* preceded the operation. Similar to the *Maybe* monad, the state threading happens in $\gg=$ where the initial function *h* is evaluated in the current state *s*. This yields a value *a* and a new state s'. By evaluating *f a*, we get a new state computation whose function is *g*. The final value/state pair results from applying *g* to the new state s'. Based on this definition, more abstract operations like *get* and *put* can be defined:

```
  -- gets current state as a value
get :: State s s
get = State $ λs → (s, s)
```

```
  -- puts new state s'
put :: s → State s a → State s ()
put s' = State $ λ_ → ((), s')
```

To illustrate the usage of the state monad and the **do** notation, let us represent a simple queue of integers with operations to *enqueue* and *dequeue* its elements. For this we define **type** *Queue s* = *State* [*s*] a state monad holding a list of values. The queue operations are defined as follows:

```
enqueue :: Int → Queue Int ()
enqueue n = do queue ← get
               put (queue ++ [ n ])
               return ()
```

```
dequeue :: Queue Int Int
dequeue = do queue ← get
             put (tail queue)
             return (head queue)
```

Other common monads include the *Error* monad for handling exceptions and the *Reader* and *Writer* monads, which can be seen as specific read-only or write-only versions of a state monad. The main issue regarding programming with plain monads is the complexity of combining several computational effects. For instance, having a monad with both state and error handling effects requires tangling both concerns into the definitions of *return* and $\gg=$. The traditional solution for modular monadic programming is to use monad transformers.

2.2. Programming with monad transformers

Using monad transformers [14] 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. More specifically, a monad transformer is defined by a type constructor *t* and the *lift* operation, as specified in the *MonadTrans* type class:

```
class MonadTrans t where
  lift :: m a → t m a
```

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 [22].

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 [19,24,22].

To illustrate the standard usage of monad transformers with implicit lifting let us consider again the *dequeue* operation. One issue of this function is that it cannot return a value when the internal list is empty; an exception is raised at runtime when this occurs. However, we can handle the situation by combining the *State* and the *Maybe* effects. Just as before, the *State* effect is required to keep the internal state of the queue, and additionally the *Maybe* effect is required to signal that some operations may not yield values. Using monad transformers the code is as follows:

```
type MaybeQueue s = MaybeT (StateT [s] Identity)
  -- A value of type MaybeQueue s a is either:
  -- a computation (StateT [s] Identity) (Just a), or
  -- a computation (StateT [s] Identity) Nothing.

dequeue :: MaybeQueue Int Int
dequeue = do queue ← get             -- with explicit lifting: lift get
             if (null queue)
               then fail Nothing     -- returns a Nothing
               else  do put (tail queue)
                        return (head queue)
```

First we define a type synonym *MaybeQueue* to denote the monad stack that combines both effects. The stack has the *Identity* monad at the bottom, upon which the *StateT* transformer is applied. The application *StateT* [*s*] *Identity* denotes a new monad that is equivalent to the *Queue* type defined in Section 2.1. Under this monad, all computations have type (*StateT* [*s*] *Identity*) *a*. To add the option for computations that do not yield values we apply the *MaybeT* monad transformer,[4] which completes the definition of *MaybeQueue*. All computations in *MaybeQueue s a* have type (*StateT* [*s*] *Identity*) (*Maybe a*). Using *MaybeQueue* it is straightforward to implement the new version of *dequeue*: we just need to check if the inner state is an empty list, in which case we *fail* the computation — thus returning *Nothing* — or we return the value just as before.

Because the implementation of *MaybeT* specifies how to perform implicit lifting of state operations, such as *get*, to inner state monads, we do not need to use explicit *lift* operations in the code. However, this flexibility does not comes for free, as we discuss in Section 3.2.

From now on we focus our development of effect capabilities in the setting of explicit and implicit lifting. Later in Section 6 we address the issue of integrating capabilities with tagged monads.

3. Effect interference in monadic programming

In this section we illustrate the problem of effect interference in monadic programming. We first illustrate the particular issue of state interference (Section 3.1), also showing that the currently accepted workaround is not scalable (Section 3.2). Then we illustrate the issue of exception interference (Section 3.3).

3.1. 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:

```
newtype StateT s m a = StateT (s → m (a, s))

class Monad m ⇒ MonadState s m | m → s where
  get :: m s
  put :: s → m ()
```

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

```
enqueue₁ :: MonadState [Int] m ⇒ Int → m ()
enqueue₁ n = do queue ← get
                put $ (queue ++ [n])

dequeue₁ :: MonadState [Int] m ⇒ m Int
dequeue₁   = do queue ← get
                put $ tail queue
                return $ head queue

push₁ :: MonadState [Int] m ⇒ Int → m ()
push₁ n = do stack ← get
             put (n : stack)

pop₁  :: MonadState [Int] m ⇒ m Int
```

[4] Note that *MaybeT* is not a standard `mtl` transformer, but is relatively well-known, *e.g.* in https://wiki.haskell.org/New_monads/MaybeT

```
pop₁     = do stack ← get
              put $ tail stack
              return $ head stack
```

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.

```
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:

```
client₁ :: M Int
client₁ = do push₁ 1
             enqueue₁ 2   -- value is put into the state layer used by the stack
             x ← pop₁     -- removes first element in stack
             y ← pop₁     -- should raise error because stack should be empty
             return (x + y)   -- yields 3 due to state interference
```

yields 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:

```
enqueue′₁ n = do queue ← lift get
                 (lift ∘ put) (queue ++ [ n ])
dequeue′₁ n = do queue ← lift get
                 (lift ∘ put) (tail queue)
                 return (head queue)
```

Importantly, the type of these new operations will reflect the expected structure of the monad stack. For example, the type of $enqueue'_1$ is:

```
enqueue′₁ :: (
  Monad m,            -- A 'bottom' monad m ...
  MonadTrans t,       -- A 'top' transformer t ...
  Monad (t m),        -- A decomposed monad (t m), with top t and bottom m
  MonadState [ Int ] m   -- Accessible m which can be constrained
) ⇒ Int → (t m) ()
```

which means that it requires a monad stack that can be decomposed into a top transformer *t* and a base state monad *m*.

However, as discussed by Schrijvers and Oliveira [22], 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.

3.2. 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 using module encapsulation. 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:

```
newtype QueueT s m a = QueueT (StateT [ s ] m a) deriving ...
```

In general we do not show most of the type class instances after a **deriving** clause; the implementation uses *GeneralizedNewtypeDeriving* extension of GHC to automatically derive the necessary instances of the *Monad*, *MonadTrans* and other type classes as required.

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

```
enqueue₂ :: s → QueueT s m ()
dequeue₂ :: QueueT s m s
```

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:

```
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
import Stack
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. Fig. 3 depicts this situation using a UML-like diagram. Class nodes represent type class definitions, stacked rectangles represent monad stacks and edges denote the "is-instance-of" relationship. In the left, the integration of the new transformers with existing monad classes; in the right, the integration of existing transformers with the new *MonadQueue* and *MonadStack* classes.

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. As an illustration, consider the implementation of the non-standard *MaybeT* monad transformer,[5] which defines 4 instances for the *MonadIO*, *MonadState*, *MonadReader* and *MonadWriter* type classes. Even worse, as some instance declarations may require access to private data constructors, the only way to implement them would be by altering the source code of the monad transformers involved. 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 [9].

[5] Found at https://wiki.haskell.org/New_monads/MaybeT.

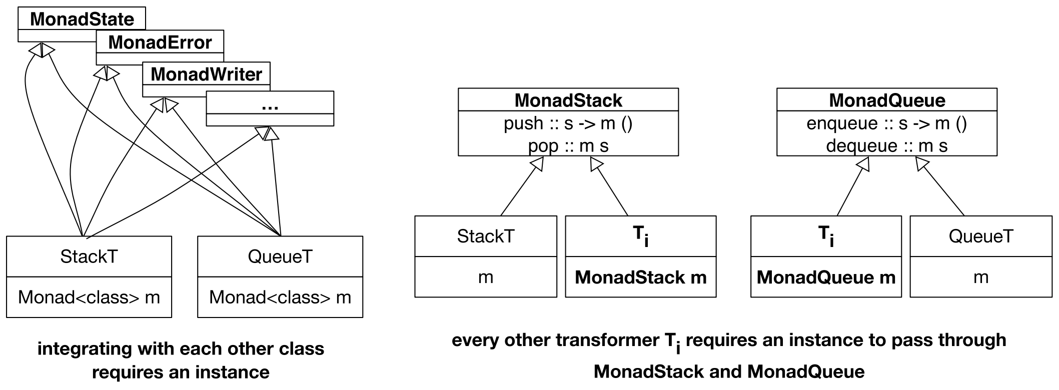

Fig. 3. Scalability issues with standard composition of monad transformers.

3.3. Exception interference

Another form of effect interference can occur 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.

```
consume :: (MonadQueue Int m, MonadError String m) ⇒ m Int
consume = do x ← dequeue
             if (x < 0) then throwError "Process error"
                        else  return x
```

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

```
process :: (MonadQueue Int m, MonadError String m) ⇒ Int → m Int
process val = consume ‘catchError‘ (λe → return val)
```

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*, $dequeue_{Ex}$, that raises an exception when trying to retrieve a value from an empty queue. Consider the type of $dequeue_{Ex}$ and let us update *consume* accordingly:

```
dequeue_Ex :: (MonadQueue s m, MonadError String m) ⇒ m s
consume = do x ← dequeue_Ex
             -- ... same as before ...
```

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

```
program_1 = do enqueue (−10)
               process 23   -- exception in consume, return default
```

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   -- empty queue exception in process, return default
```

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. In other words, exception interference arises whenever it is possible to have "catch-all" handlers, regardless of the representation of exception values as strings or other more complex types. Similar to state interference, current solutions rely on custom exception transformers and explicit lifting.

As argued by Harper [7], the standard semantics of exceptions hinders the modular composition of programs because of the potentially modified exception flows. Indeed, issues like this have been identified in the context of aspect-oriented programming [5]. In Section 5.3 we show how effect capabilities allows us to define exceptions that, like in Standard ML, can be protected from unwanted interception.

4. A generic static framework for capabilities and permissions

At the type level a capability declares a static restriction on the usage of a protected operation. Such operation can only be performed at runtime by presenting a *token* — an unforgeable value or witness of the required capability type. The fact that a token of the proper type is presented for each protected operation is verified statically by the type checker, using the standard mechanism to verify that functions are called with arguments of the correct types. Therefore the essential issue to address regarding capabilities is how are the tokens created and shared. Our approach is in part similar to abstract data types, where the abstract type is public but the implementation is kept hidden. This means the capability type is publicly exported, while the corresponding data constructor is kept private. Unlike abstract data types, data constructors can be statically shared using a mechanism based on type classes.

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 4.1). Then, we refine simple capabilities with type-based and user-definable permission lattices (Section 4.2). Finally we show how capabilities can be shared between modules (Section 4.3) while providing two key features of capabilities-based mechanisms: delegation and attenuation (Section 4.4).

4.1. Capabilities as a computational effect

A capability is a singleton type whose type is public but whose constructor is private. We refer to both the type and the constructor as *capability*, because the distinction is always clear from context: capability types appear only on type signatures and annotations, and capability tokens appear only in function implementations. For instance, consider the capability for read/write access to some state:

```
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*.

```
newtype CapT c m a = CapT (ReaderT c m a) deriving ...
fromCapT :: c → CapT c m a → m a
fromCapT !c (CapT ma) = runReaderT ma c
  -- syntactic sugar for infix operation
withCapability m c = fromCapT c m
```

As an example of a protected computation, consider a module *A* that uses *RWCap* to restrict access to a state monad holding a value of type *s*.

```
module A (getp, putp, RWCap ()) where
data RWCap = RWCap
getp :: CapT RWCap (State s) s
putp :: s → CapT RWCap (State s) ()
```

A module *B* that imports *A* will get access to both operations — and will be able to construct and pass around computations of type *CapT* — 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*.

Unforgeability The soundness of capability-based systems relies on the unforgeability of tokens. In our context this means that only the capability data constructor should be able to produce access tokens to perform operations protected by the corresponding capability type. However, Haskell has two features with which it is always possible to forge a capability. The first is *undefined*, represented as ⊥, which is an expression that pertains to all types, and directly fails with an error if evaluated. The second is lazy evaluation itself, meaning that an unused argument is never evaluated. Hence, a malicious module can use ⊥ to pass as any capability, fulfilling the expected type of *fromCapT*, while not triggering any errors if, due to lazy evaluation, the given capability value is never evaluated! Following the previous example, a module *B* could forge its access to protected operations like *getp*:

```
module B where
import A
malicious_get = getp (⊥ :: RWCap)
```

To avoid this situation, we use a strictness annotation ! in the implementation of *fromCapT*. In a setting with strict evaluation this runtime check would also be necessary if there exists a bottom value like *null* in typical imperative or object-oriented. Nevertheless, effect capabilities still are a mechanism for static verification of access. Effect capabilities guarantee that in a well-typed program protected operations will either: (i) be performed by authorized entities only, or (ii) they will crash the program due to the forced evaluation of $\perp$.

4.2. Private lattice of permissions

Capabilities are unforgeable authority tokens that unlock specific operations. Ideally, a system should follow the *principle of least privilege* [20], 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; permissions are also defined as singleton types.

Permission lattices Permissions can be organized in a lattice specified by a $\supset$ type class. $\supset$ is a simple reflexive and transitive relation on types defined as follows. 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.

```
class    a ⊃ b
instance a ⊃ a   -- generic instance for reflexivity
```

However, we cannot make $\supset$ be public because it would allow a malicious user to add a new undesirable relation in the lattice to effectively bypass permission checking altogether. Still, we want to be able to impose constraints based on the private lattice in other modules. A first solution to this issue is to define a private lattice $\sqsupset$ such that being an instance of $\supset$ requires also an instance of $\sqsupset$. If the private lattice is not exported then $\supset$ will not be updatable from outside of the module, because extending it always requires to define an instance on the private lattice. The code for this is simply:

```
class a ⊐ b   -- private lattice
instance a ⊐ a   -- reflexivity
class    a ⊐ b ⇒ a ⊃ b   -- public lattice
instance a ⊐ b ⇒ a ⊃ b
```

We use a solution that is more sophisticated but that allows us to implement general abstract data types (Section 6.3). In short, the problem is that using regular type classes it is not possible to be parametric on both the capabilities and the permissions bound to a specific lattice; the typechecker will demand an instance of $\sqsupset$, which cannot possibly be provided unless $\sqsupset$ is publicly exported.

The solution is to implement $\sqsupset$ as a *closed type family* [4,3] that can also be used as a constraint in the declaration of $\supset$. In essence, type families can be used to define "overloaded" type constructors whose concrete type will depend on the type of its arguments (which are also types); *e.g.* type families are to types as type classes are to regular functions. In a *closed* type family, all the overloading equations are defined only once and cannot be modified externally.

The code is similar to the previous one:

```
type family (a ⊐ b) :: Constraint where
      -- particular instances
   a ⊐ a = ()   -- reflexivity
class a ⊐ b ⇒ a ⊃ b
```

To implement this we require the *TypeFamilies* and *ConstraintKinds* extensions of GHC. The former allows us to define type families while the latter allows us to use instances of the type family as a constraint in the definition of $\supset$.

The difference between implementing the private lattice as a type class or as a closed type family is that in 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 and no instances can be added from outside, but such classes cannot be used in any external type signature.

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
type family (a ⊐RW b) :: Constraint where
  RWPerm ⊐RW ReadPerm = ()    -- we do not use the actual type,
  RWPerm ⊐RW WritePer  = ()   -- just the fact that the instance exists
  a ⊐RW a = ()                -- which then satisfies a given constraint.
  -- public lattice
class      a ⊐RW b ⇒ a ⊃RW b
instance a ⊐RW b ⇒ a ⊃RW b
```

Using the public permission lattice allows developers to safely 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 4.1 that type class resolution statically checks for proper permissions when a computation is evaluated using *fromCapT*.

Capabilities as permission namespaces Capability constructors, like *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 for permissions. This allows a module to have restricted read-only access to some state layer, while still having full read-write access to another state layer.

4.3. 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 5.2, 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* value passing, using type classes, loosely inspired by the π-calculus notion of messages and channels [21].

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

```
newtype Channel ch a = Channel (Reader ch a) deriving ...
fromChannel :: ch → Channel ch a → a
fromChannel ! ch (Channel ma) = runReader ma ch
```

We define a type class *Send* for sending capabilities over channels:

```
class Send ch c p where
  receive :: p → 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 value of type *c p* on channel *ch* amounts to declaring an instance *Send ch c p*. Conversely, receiving a value of type *c p* on *ch* amounts to applying the function *receive* to *p* and getting the value back using *fromChannel*. The expected result type *c p* has to be provided explicitly, because messages of different types can be sent on the same channel. Furthermore, only one value of type *c p* can be sent on a specific channel — but this is sufficient for our purposes since capabilities are singleton types.

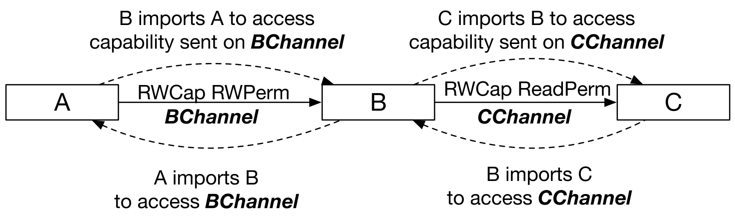
B imports A to access capability sent on BChannel
C imports B to access capability sent on CChannel
A
RWCap RWPerm
BChannel
B
RWCap ReadPerm
CChannel
C
A imports B to access BChannel
B imports C to access CChannel

Fig. 4. Static delegation and attenuation of capabilities. Module *A* sends a capability with read-write access to *B*. *B* sends a read-only capability to *C*.

Observe that the messaging protocol is rather asymmetric, because capabilities are sent statically by declaring type class instances, whereas they 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 a proper instance of *Send*, or else type checking will fail. Therefore, the protocol ensures that one module can only receive a message that has been sent to it.

Inter-module communication requires that both sender and receiver modules have knowledge of each other, thus they are both mutually dependent. We rely on GHC's support for mutually recursive modules for inter-module communication.[6] Although technically sound this requirement reduces the independent evolution and composition of modules, and is a challenge for further work.

For instance, following the motivating 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 5.2):

```
instance
  Send      -- send...
  PQChan    -- ... on channel PQChan
  RWCap     -- ... capability RWCap
  ReadPerm  -- ... with ReadPerm permission
  where
      -- receiving ReadPerm from channel PQChan yields a RWCap
      -- capability with ReadPerm permission
    receive ReadPerm = return $ RWCap ReadPerm
```

and dually, the *PriorityQueue* module requests the capability *RWCap ReadPerm* from the channel *PQChan*:

```
cap :: RWCap ReadPerm
cap = fromChannel PQChan $ receive ReadPerm
```

4.4. Delegation and attenuation

Capabilities-based mechanisms feature two characteristics called *delegation* and *attenuation* [13]. 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 and show a simple scenario as example, depicted conceptually in Fig. 4 and in actual code in Fig. 5. The scenario consists of a module *A* that sends a capability with read/write permission to module *B*, which in turn sends a restricted read-only capability to module *C*. Note the mutual dependency between modules is also reflected in Fig. 4 (dotted arrows denote module imports).

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*.

Attenuation 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 attenuation of capabilities, 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

[6] http://www.haskell.org/ghc/docs/latest/html/users_guide/separate-compilation.html.

```
module A where
import B   -- to send capability on BChannel

instance Send BChannel RWCap RWPerm where
  receive RWPerm = return $ RWCap RWPerm
```

```
module B where
import A   -- to get the capability sent by A
import C   -- to send capability on CChannel

data BChannel = BChannel |

  -- send attenuated version of the capability to C, only with read permission
instance Send CChannel RWCap ReadPerm where
  receive ReadPerm = return (attenuate rwCap ReadPerm)
    where rwCap = fromChannel BChannel $ receive RWPerm
```

```
module C where
import B   -- to get the capability sent on B
data CChannel = CChannel
cap :: RWCap ReadPerm
cap = fromChannel CChannel $ receive ReadPerm
```

Fig. 5. Code for the inter-module sharing depicted in Fig. 4.

```
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. 6. Protected versions of the monad state type class and state monad transformer.

function *attenuate*. Note that $\supset$ is a parameter of the class because different capabilities may be defined on different lattices, but the functional dependency $c \to \supset$ 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).

5. Effect capabilities: upgrading monads with capabilities

We now develop the core 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 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 computations 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 start illustrating how to implement private and shared state (Section 5.1 and Section 5.2) and protected exceptions (Section 5.3). Finally, we show how to control access to the *IO* monad, which provides access to a variety of operations (file operations, network communication, etc.), by providing a fine-grained lattice of permissions (Section 5.4).

5.1. Private persistent state

Based on the state permission lattice (Section 4.2), we define the protected versions of the state monad transformer and corresponding type class (Fig. 6). To use the *getp* function, one needs to have a capability *c* that implies the *ReadPerm* permission; and dually to use the *putp* function, one needs the capability that implies the *WritePerm* permission. Abstracting state operations on capability *c* results in the use of capabilities as namespaces for permissions, as mentioned before in Section 4.2. Notice that the left-hand side in the declaration of *StateTP* uses the capability type *c* as a *phantom type variable* [12] used solely to index the layer of the monad stack with the respective capability. In Section 6 we exploit this type index to combine effect capabilities with the mechanism of tagged monads. To illustrate, consider the following polymorphic *Queue* using private state:

```
module Queue (enqueue, dequeue, QState ()) where
import EffectCapabilities
import Control.Monad.MonadStateP
data QState p = QState p
instance Capability QState ⊃_RW where
  attenuate (QState _) perm = QState perm
type QStateLayer m s =
  MonadStateP   -- given a monad m with a protected state layer
    QState      -- protected by capability QState
    [s]         -- holding a list of values of type s.
    m
enqueue :: QStateLayer m s ⇒ s → m ()
enqueue s = do queue ← getp `withCapability` (QState ReadPerm)
               putp (queue ++ [s]) `withCapability` (QState WritePerm)
dequeue :: QStateLayer m s ⇒ m s
dequeue   = do queue ← getp `withCapability` (QState ReadPerm)
               putp (tail queue) `withCapability` (QState WritePerm)
               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. For convenience we define the *QStateLayer* type synonym, which can be used as a class constraint thanks to the *ConstraintKinds* extension of GHC. 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:

```
type M = StateTP (QState ()) [Int] Identity
```

To construct a monad stack we are only interested in 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*.

5.2. Shared persistent state

We now illustrate capability sharing with shared persistent state. We define a *PriorityQueue* module 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   -- to get QState capability type
data PQChan = PQChan
queueState :: QState ReadPerm
queueState = fromChannel PQChan $ receive ReadPerm
peekBy :: (Ord s, QStateLayer m s)   -- QStateLayer for an ordered type s
```

```
data ThrowPerm = ThrowPerm
data CatchPerm  = CatchPerm
data TCPerm     = TCPerm

  -- private lattice
type family (a ⊐_Ex b) :: Constraint where
  TCPerm ⊐_Ex ThrowPerm
  TCPerm ⊐_Ex CatchPerm
  a ⊐_Ex a

  -- public lattice
class      a ⊐_Ex b ⇒ a ⊃_Ex b
instance a ⊐_Ex b ⇒ a ⊃_Ex b

class (Monad m, Error e) ⇒ MonadErrorP c e m | c m → e where
  throwErrorp :: p ⊃_Ex ThrowPerm ⇒ e → CapT (c p) m a
  catchErrorp ::  p ⊃_Ex CatchPerm   ⇒ m a → (e → m a) → CapT (c p) 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. 7. Implementing protected exceptions: permission lattice and protected versions of the monad error type class and error monad transformer.

```
            ⇒ (s → s → Ordering)    -- comparator to order s values
            → m (Maybe s)            -- computation that maybe has a value s
peekBy comp = do queue ← getp ‘withCapability‘ queueState
                 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 sent by *Queue*. Then *peekBy* can access the internal state of the queue as usual in the capabilities framework.

5.3. 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 also be implemented using exception capabilities, similarly to the implementation of protected state (Fig. 7). First, we define the private and public lattices, $\sqsupset_{Ex}$ and $\supset_{Ex}$, as well as permissions *ThrowPerm*, *CatchPerm* and *TCPerm* for throw-only, catch-only, and throw-catch permissions. Then we define the protected versions of the standard *ErrorT* monad transformer and *MonadError* type class.

Recall from Section 3.3 the example of exception interference, involving the *consume* and *process* functions:

```
program₂ = process 23   -- empty queue exception caught by process, return 23
```

We can address exception interference using a *QError* exception capability in the implementation of $dequeue_{Ex}$. This allows the *Queue* module to control which other modules are allowed to define their own handlers. To this end, let us consider a *QueueEx* module, similar to *Queue* but where the *dequeue* operation, now $dequeue_{Ex}$, raises an exception if the queue is empty.

```
module QueueEx (enqueue, dequeue_Ex, QState ()) where
  -- ... QState definition, type class instances and enqueue as before ...
```

Similarly to protected state, we define the *QErrorLayer* type synonym which represents an error monad protected by the *QError* capability:

```
data QError p = QError p
instance Capability QError ⊃_Ex  where
  attenuate (QError _) perm = QError perm
type QErrorLayer m =
  MonadErrorP   -- given a monad m with protected exception layer
  QError           -- protected by capability QError
  String            -- bound to String as the error representation
  m
```

We focus on the implementation of *dequeue*$_{Ex}$. This function requires access to both effects: the state of the queue, *QStateLayer*, and the exception mechanism, *QErrorLayer*.

```
dequeue_Ex :: (QStateLayer m s   -- Access to QState-protected state layer
              , QErrorLayer m    -- Access to QError-protected error layer
              ) ⇒ m s
dequeue_Ex =
  do queue ← getp `withCapability` (QState ReadPerm)
     if null queue
       then throwErrorp "Empty..." `withCapability` (QError ThrowPerm)
       else do putp (tail queue) `withCapability` (QState WritePerm)
               return (head queue)
```

By implementing *consume* in terms of *dequeue*$_{Ex}$ we can distinguish between errors due to an empty stack, or errors due to the invariant in *consume*:

```
consume :: ( QStateLayer m Int    -- Access to protected state layer
           , QErrorLayer m        -- Access to protected exception layer
           , MonadError String    -- Access to unprotected exception layer
           ) ⇒ m Int
consume = do x ← dequeue_Ex
             if (x < 0) then throwError"Process error"
                        else return x
```

This code assumes a monad stack *m* with two error layers: one layer uses the regular *MonadError* class, while the second one uses the protected *MonadErrorP* class. Then we adapt the type of *process* based on the new definition of *consume*:

```
process :: (QStateLayer m Int,
            QErrorLayer m,
            MonadError String m) ⇒ Int → m Int
process val = consume `catchError` (λe → return val)
```

Crucially, the new type signature of *process* makes explicit the fact that *process* cannot catch or throw exceptions bound to *QError*, unless it has the proper capability. Furthermore, since the implementation uses *catchError*, which refers to the regular *MonadError* layer, *process* will only handle errors produced by a violation of the invariant in *consume*. Indeed, the following programs yield the expected results:

```
program_1 = do enqueue (−10)
               produce 23   -- returns the default value
program_2 = produce 23   -- returns an exception due to empty queue
```

Now consider a *debug* function in a module that has access to the *QError* capability with permission implying *CatchPerm*. In practice this means that *debug* can define a custom exception handler:

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

Finally, for cases where the client is not trusted and cannot catch the exception, we can export a function *dequeue*$_{Err}$, that reraises the exception using Haskell's built-in *error* function:

```
dequeue_Err :: (QStateLayer m s, QErrorLayer m) ⇒ m s
dequeue_Err = dequeue_Ex `catchErrorp` error
                `withCapability` (QError CatchPerm)
```

Notice that the implementation of *consume* and *process* is asymmetric: we use a protected error layer and also a regular non-protected error layer. This design is rather arbitrary and considers a general "for-all-purposes" error layer and a specific layer for communicating queue exceptions. This is not a restriction nor an imposition of effect capabilities, in fact, the precise design and specification of how many exception layers depends on the particular requirements of a software component.

5.4. Taming the IO monad

Monadic I/O is a defining characteristic of Haskell as a purely functional language, in contrast to *e.g.* ML or Scheme where I/O is performed through direct side-effecting operations. The *IO* monad is special in that it interacts with the *real world*; it is also unique because there exists no *IO* monad transformer, and as such it can only be used at the bottom of a monad stack [32]. Because of its interaction with the real world, a misbehaving module that uses *IO* is particularly problematic and could have severe consequences to the user of a program (*e.g.* it could delete all files in the user directory). Components evaluated on a stack with *IO* at the bottom can access its effects even when using effect capabilities due to the implicit instances of the *MonadIO* class that provide access to the *liftIO* :: *IO a* → *m a* operation. To address this situation we define the protected *IOP* monad as a replacement for *IO*; and *MonadIOP* as a replacement for the unprotected *MonadIO* class. Performing protected I/O operations now requires the *IOPC* capability. The definitions for protected *IO* are as follows:

```
module IOP (IOP (), IOPC (), liftIOP, runIOP) where
data IOPC p = IOPC p
instance Capability IOPC ⊃IO where
  attenuate (IOPC _) = IOPC perm
newtype IOP a = IOP { unIOP :: IO a } deriving ...
  -- lattices ⊐IO and ⊃IO to be defined...
runIOP :: IOP a → IO a
runIOP pio = unIOP pio
class Monad m ⇒ MonadIOP m where
  liftIOP :: IOPC perm → IO a → m a
instance MonadIOP IOP where
  liftIOP ! c io = IOP io
```

To use protected I/O a client program still needs to use *IO* at the bottom of the monad stack, however, the fundamental idea is that *IO* operations should only be performed by trusted parts of the system, and only from *IOP* computations constructed elsewhere. The $\sqsupset_{IO}$ permission lattice is specified similarly to the previous lattices, and is described later in this section.

Existing *IO*-based libraries can be reused thanks to the function *liftIOP*, which is the only operation defined by *MonadIOP*. Using *liftIOP* one can construct protected I/O computations by wrapping existing *IO* computations, provided that the proper *IOPC* capability is presented. For example, the protected version of *readFile* is implemented as follows (note we keep the same name as its *IO* counterpart):

```
readFile :: (Capability IOPC ⊃IO          -- capability to use protected IO
            , perm ⊃IO FileInputPerm   -- actual permission for file input operations
            ) ⇒ IOPC perm → FilePath → IOP String
readFile c path = liftIOP c $ IO.readFile path
```

Finally, to actually perform *IOP* effects, *runIOP* unwraps *IOP* computations. Note that this function is not protected as it just runs computations that have already exhibited their capability to perform *IO* effects. As an illustration consider a trusted module *Config*, which manages configuration files. Trusted means that *IOP* passes it the *IOPC* permissions for reading and writing files on channel *ConfigChan* — but no other permissions. To avoid the direct dependency of *IOP* to each trusted module, it should be possible to use a compile-time flag that specifies which modules receive *IOPC* on which channels. This would be similar to how Safe Haskell [30] manages the trust relationship between modules. Then, *Config* can export functionality to access configuration files to external modules.

```
module Config (readConfig, writeConfig, ...) where
import IOP as IOP
data ConfigChan deriving Generic
instance Channel ConfigChan
fileInputPerm    :: IOPC FileInputPerm
fileInputPerm    = fromChannel ConfigChan $ receive ConfigChan
fileOutputPerm :: IOPC FileOutputPerm
fileOutputPerm = fromChannel ConfigChan $ receive ConfigChan
readConfig  f = IOP.readFile fileInputPerm f
writeConfig f = IOP.writeFile fileOutputPerm f
main = runIOP $ readConfig "config.txt"
```

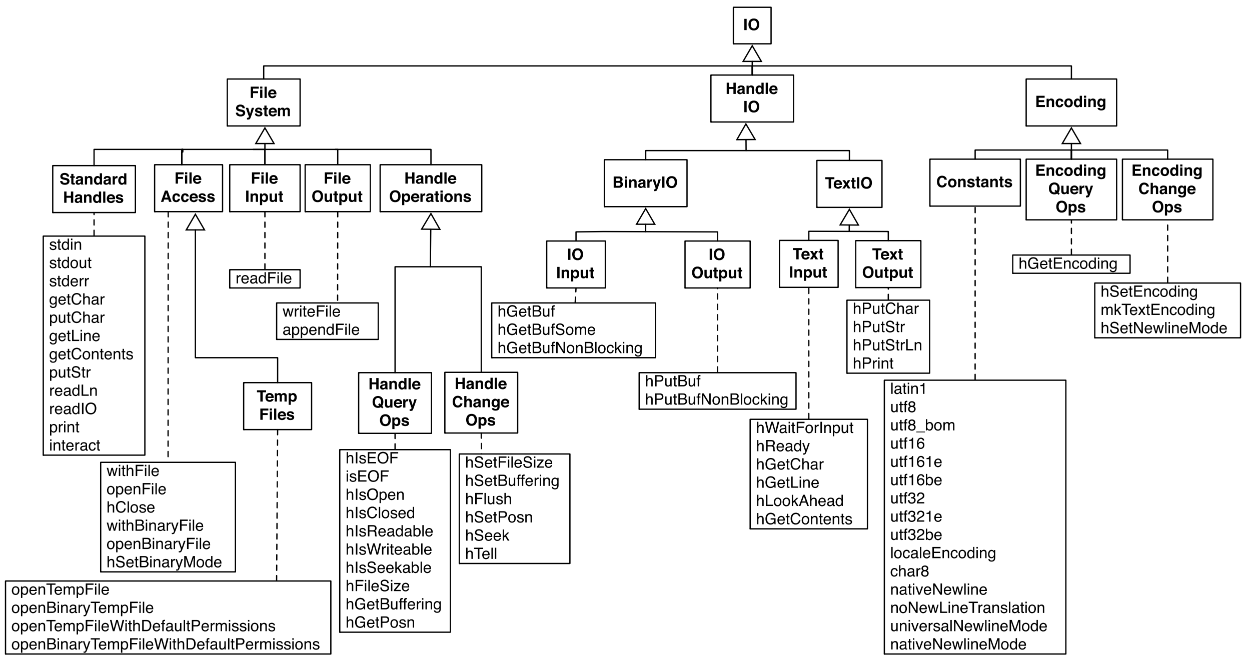

Fig. 8. Lattice of permissions for the *IO* monad, and association to monadic operations.

Note that external modules still have no access to I/O operations, except through the *Config* module. In Haskell the *main* function is the entry point to the real world where the I/O operations *declared* in the *IO* monad are actually *performed*. Therefore, in this program I/O effects are only evaluated from trusted and protected computations by using *IOP*.

The lattice of IO permissions A particularly interesting application of effect capabilities with permissions is the *IO* monad. The *IO* monad in Haskell provides 76 portable operations, suggesting that a single capability for the whole *IO* monad is clearly too coarse-grained. Providing a module with the ability to use *stdin* and *stdout* should not necessarily imply granting it the ability to use any of these 76 operations, such as that for altering the content of files. Similarly, one may want to grant a module with the ability to write *text* to a file, while ensuring that the module does not use this ability to write arbitrary *binary data*. Another example at the file level is to grant permission to open only *temporary* files — which comes with the guarantee that existing files cannot be overwritten.

We studied the 76 portable operations of the *IO* monad, and propose a lattice of permissions that allows fine-grained control over these operations (Fig. 8). Based on this lattice of permissions, we have redefined the *IOP* monad such that the type of each operation specifies the required permission. Note that the lattice we propose is directly based on the documentation of the *IO* monad in GHC,[7] with few refinements. It should be considered as a first step towards a practical lattice for permission-based I/O in Haskell.

6. The marriage of tags and capabilities

In this section we show how to leverage the benefits of tagged monads (Section 2), namely the robustness with respect to layout changes to the monad stack, with the security guarantees of effect capabilities. We use *monad views* [22] as a specific implementation of tagged monads, although integrating effect capabilities in other mechanisms should be very similar. Before developing the technical contribution of this section, we go back to the motivating example of effect interference (Section 3.1), where we wanted to implement a stack and a queue ADT using two state layers:

```
type M = StateT [Int] (StateT [Int] Identity)
client = do push 1
            enqueue 2       -- value is put into the state layer used by the stack
            x ← pop
            y ← pop         -- should raise error because stack should be empty
            return (x + y)  -- yields 3 due to state interference
```

[7] http://hackage.haskell.org/package/base/docs/System-IO.html.

In the combined approach with effect capabilities and monad views, the correct implementation — without effect interference — is as straightforward:

```
type M = TStateTP (QState ()) [Int] (TStateTP (SState ()) [Int] Identity)
client₁ = do enqueue vq 1
              push vs 2
              x ← pop vs
              y ← pop vs
              return (x + y)
          where vq = structure (tag :: (QState ()))
                vs = structure (tag :: (SState ()))
```

The main differences between these two versions are:

- the use of a tagged monad stack, with tags *QState* and *SState* for the state of the queue and the stack, respectively, using the *TStateTP* monad transformer,
- the new implementation of *enqueue*, *pop* and other operations, which take an additional argument *vs* or *vq*, and
- the definition of *vq* and *vs* that correspond to the *views* associated to the *QState* and the *SState* tag, respectively. These views are created using *structure* and *tag* operations and represent the specific parts of the monad stack on which the operations will work — effectively separating the states of the queue and the stack.

The same mechanism also solves the issue of exception interference (Section 3.3). We first define a tagged monad stack with two layers for exceptions, which are defined with the *TErrorTP* transformer:

```
type M = TErrorTP (QError ()) String
           (TErrorTP (EError ()) String (TStateTP (QState ()) [Int] Identity))
```

then, using the updated implementations of *consume* and *process*, we get the expected behavior — without exception interference — in the following two programs:

```
  -- uses default value 23, catches invariant-related exception in process
program₁ = do enqueue vq (−10)
               process 23
            where vq = structure (tag :: QState ())
  -- raises "Queue is empty" exception, queue exception not caught by process
program₂ = process 23
```

In the rest of this section we briefly summarize monad views (Section 6.1) and then describe how to implement *protected tagged monads* using effect capabilities (Section 6.2). Finally, we show how to use this mechanism to program with abstract data types in a quite imperative-like fashion (Section 6.3).

6.1. Monad views in a nutshell

Monad views were introduced by Schrijvers and Oliveira [22] as a mechanism to address the lack of robustness of monadic components with respect to specific monad stack layouts. Theoretically, monad views are considered as monad morphisms in a category with monads as objects and monad morphisms as arrows. Views are denoted using $\rightsquigarrow$, and are instance of the *View* type class. All views support the *from* operation:

$$from :: (Monad\ m, Monad\ n, View\ (\rightsquigarrow)) \Rightarrow n \rightsquigarrow m \to n\ a \to m\ a$$

Additionally, *bidirectional views*, denoted as $\bowtie$, also support the *to* operation:

$$to :: (Monad\ m, Monad\ n) \Rightarrow n \bowtie m \to m\ a \to n\ a$$

To summarize the intuition behind monad views, given monads *m* and *n* a view allows us to transform computations in *n* to computations in *m*. A bidirectional view additionally supports the inverse *to* transformation.

View-specific operations Monadic operations can be parameterized by views, which are first-class values. For instance consider *getv* and *putv* defined in [22], where *getv*:

```
getv :: (Monad m, MonadState s n, View (⤳)) ⇒ (n ⤳ m) → m s
getv v = from v $ get
```

returns a computation $m\,s$ from an arbitrary state layer n. Conversely, *putv* puts a new value into the state layer n:

```
putv :: (Monad m, MonadState s n, View (⤳)) ⇒ (n ⤳ m) → s → m ()
putv v = from v ∘ put
```

Views and tagged monads Under the monad views mechanism a tagged monad is created using *nominal masks* [22]. A nominal mask refers to layers of the stack using *names* instead of relative positions. This is done with the *tag monad transformer* *TagT* (*tag*), which labels a particular position of the monad stack with an arbitrary type *tag*. For inspecting tagged monad stacks the type class $n \sqsubseteq_{tag} m$ exposes a monad n representing the layer of the stack m tagged with type *tag*. That is, we have the property:

$$n \sqsubseteq_{tag} m \quad \Rightarrow \quad \exists t. \quad m = t\ (TagT\ (tag)\ n).$$

To obtain the view associated to *tag* that relates n and m we must use the *structure* operation:

```
class (Monad m, Monad n) ⇒ n ⊑_tag m where
  structure :: View (⤳) ⇒ tag → (n ⤳ m)
```

In practice programmers use the tagged variants of the regular monad transformers, which are implemented in terms of *TagT*. For instance, the tagged state and exception transformers are defined as follows:

```
type TStateT tag s m = TagT tag (StateT s m)
type TErrorT tag e m = TagT tag (ErrorT s m)
```

6.2. Protected tagged monads

In all tagged monad approaches [19,24,22], we are left with the issue that explicit lifting can still be used to access a layer in the monad stack without holding the tag. For instance, the state interference example using tagged monads, but without effect capabilities is defined as:

```
type M = TStateT (SState ()) [Int] (TStateT (QState ()) [Int] Identity)
client₁ :: M Int
client₁ = do enqueue vq 1
             push vs 2
             x ← pop vs
             y ← pop vs    -- error empty list, as expected
             return (x + y)
          where vq = structure (QState ())
                vs = structure (SState ())
```

However, as views do not preclude the use of explicit lifting a function like *evil_pop* could deduce the specific layout of the monad stack used to run a computation and then access the internal state of the queue:

```
evil_pop tag = do stack ← lift ∘ lift $ get
  lift ∘ lift ∘ put $ (tail stack)
  return $ head stack
```

Indeed, in the following example the state of the queue is altered by *evil_pop*, yielding 3 as a result:

```
client₂ :: M Int
client₂ = do enqueue vq 1
             push vs 2
             x ← evil_pop vs   -- gets the 1 from the queue
             y ← pop vs        -- gets the 2 from the stack
             return (x + y)    -- returns 3
          where vq = structure (QState ())
                vs = structure (SState ())
```

We therefore propose to use tags as capabilities to perform the operations of a stack layer, thus benefiting both from the robustness provided by tagged monads and the access discipline of effect capabilities. As a naming convention, we append the *PV* suffix to the name of the transformers and type classes, *e.g. MonadStatePV* is the tagged protected version of *MonadState*.

Previously we mentioned that to build a concrete monad stack we use the capability type constructor along the () type as the permission (Section 5.1). Here we follow the same design, because to use a capability as a tag, we need to get rid of the permission to get a tag that does only depend on the capability type. For that, we can assume that the lattice of permissions of the capability as the unit type () as bottom element. This is done for instance for the $\sqsupset_{RW}$ lattice by defining the parametric instance, and by defining the corresponding instance in the closed type family $\sqsupset_{RW}$:

```
type family ($\sqsupset_{RW}$ a b) :: Constraint where
    -- ... other instances
  $\sqsupset_{RW}$ a ()
```

Then, any capability with permissions can be turned into a tag by using *attenuate*. However, this mechanism requires access to a capability value, which may not be available in the case of private capabilities! To address this situation we add the *tag* method to the *Capability* type class, which simply returns a witness of the capability with an empty permission:

```
class Capability c ⊃ | c → ⊃ where
  attenuate :: $p_1$ ⊃ $p_2$ ⇒ c $p_1$ → $p_2$ → c $p_2$
  tag :: c ()
```

We start by describing the implementation of *MonadStatePV*, which leverages the protected type class *MonadStateP*. Essentially, we require a monad stack *m*, that has a layer *n* tagged with capability *c*, and where *n* is an instance of *MonadStateP*. The class defines two operations, *getpv* and *putpv*, that return protected computations that alter the specific state layer *n*. The implementation of *MonadStatePV* is:

```
class (n $\sqsubseteq_{(c\ ())}$ m              -- in m there is a monad view n tagged with c ()
      , Capability c $\supset_{RW}$    -- capability c and its R/W permission lattice
      , MonadStateP c s n   -- n must be a state layer protected by c for state s
      ) ⇒ MonadStatePV c s n m where
  getpv :: (p $\supset_{RW}$ ReadPerm, View (⇝)) ⇒ n ⇝ m → CapT (c p) m s
  getpv tag = mapCapT (from tag) getp
  putpv :: (p $\supset_{RW}$ WritePerm, View (⇝)) ⇒ n ⇝ m → s → CapT (c p) m ()
  putpv tag s = mapCapT (from tag) $ putp s
```

The marriage of tags and capabilities is reflected in the constraints of the class definition. The $n \sqsubseteq_{(c\ ())} m$ constraint imposes that *m* exposes a layer *n* labeled with capability *c*. At the same time, *MonadStateP c s n* requires that *n* is a state layer protected by the same capability *c*. Class operations are straightforwardly defined based on the regular protected operations, using the machinery of monad views, that is, *getpv* and *putpv* take views as arguments. Finally, because the protected operations *getp* and *putp* yield computations in monad *n*, we use *mapCapT* to apply the *from tag* transformation, that turns computations in *n* to computations in *m*, to obtain protected computations in *m*, as is expected for each operation. The definition of *mapCapT* is:

```
mapCapT :: (m a → n b) → CapT c m a → CapT c n b
mapCapT f (CapT c) = CapT (mapReaderT f c)
```

For the exception effect we define the *MonadErrorPV* class in a similar way:

```
class (n $\sqsubseteq_{(c\ ())}$ m              -- in m there is a monad view n tagged with c ()
      , Capability c $\supset_{Ex}$    -- capability c and its throw/catch permission lattice
      , MonadErrorP c e n   -- n must be an error layer protected by c for error e
      ) ⇒ MonadErrorPV c e n m where
  throwErrorpv :: (p $\supset_{Ex}$ ThrowPerm, View (⇝)) ⇒ n ⇝ m →
                  e → CapT (c p) m a
  throwErrorpv tag e = mapCapT (from tag) $ throwErrorp e
  catchErrorpv ::  (p $\supset_{Ex}$ CatchPerm) ⇒ n ⋈ m →
                  m a → (e → m a) → CapT (c p) m a
  catchErrorpv tag ma hnd = mapCapT (from tag) $
                               catchErrorp (to tag ma) (to tag ∘ hnd)
```

A notable asymmetry in this class is that *catchErrorpv* specifically *requires* a bidirectional view (denoted as $\bowtie$). This is because throwing an exception at layer n means only to put a computation into the corresponding part of m, which is exactly the functionality provided by *from*. However when using *catchErrorpv* it is necessary to inspect a computation in m to detect whether an exception is present in layer n, which can be done only in bidirectional views using *to*.

6.3. *Programming with abstract data types*

A critical limitation of the methodology to program using abstract data types (ADTs) using effect capabilities, as presented so far, is that each queue or stack module represents *a single and standalone instance* of either a queue or a stack. In contrast, imperative programming languages like C allow developers to define an abstract set of operations — the ADT interface — which are parametrized by a *handle*, that holds the specific information of each instance. Indeed, since the queue and stack operations are implemented specifically in terms of the *QState* or *SState* capability, having more than one ADT would require error-prone duplication of modules. In this section we show that by parametrizing over the capability type it is possible to implement a generic stack abstract data type, just like in other imperative languages.

To start, we define a *StackHandle* as a structure that holds both the required tags — for accessing the proper layers in a monad stack — and the required capabilities.

```
data StackHandle m n_s n_e c_s c_e p_s p_e = StackHandle {
  tag_s :: n_s ⇝ m,     -- view to access state layer
  tag_e :: n_e ⋈ m,     -- view to access exception layer
  cap_s :: c_s p_s,     -- capability to perform state computations
  cap_e :: c_e p_e      -- capability to perform exception computations
}
```

To enforce the proper type constraints in the types of a *StackHandle*, we define the *mkStackHandle* function:

```
mkStackHandle :: (Monad m, Monad n, ...)
              ⇒ n_s ⋈ m → n_e ⋈ m → c_s perm_s → c_e perm_e
              → StackHandle m n_s n_e c_s c_e perm_s perm_e
mkStackHandle = StackHandle
```

Having access to a value of type *StackHandle* means having access to a particular instance of the stack ADT. The number of instances is only constrained by the size and layout of the monad stack used in a given program. The *push* and *pop* operations are defined as before, but are parametric on capability c:

```
push :: (Capability c_s ⊐_RW
        , perm_s ⊐_RW ReadPerm
        , perm_s ⊐_RW WritePerm
        , ...) ⇒ StackHandle m n_s n_e c_s c_e perm_s perm_e → s → m ()
push (StackHandle tag_s tag_e cap_s cap_e) x = do
  stack ← getpv tag_s ‘withCapability‘ cap_s
  putpv tag_s (x : stack) ‘withCapability‘ cap_s
  return ()
pop :: (Capability c_e ⊐_Ex, perm_e ⊐_Ex ThrowPerm, ...)
     ⇒ StackHandle m n_s n_e c_s c_e p_s p_e → m s
pop (StackHandle tag_s tag_e cap_s cap_e) = do
  stack ← getpv tag_s ‘withCapability‘ cap_s
  if null stack
    then throwErrorpv tag_e "Stack is empty" ‘withCapability‘ cap_e
    else do putpv tag_s (tail stack) ‘withCapability‘ cap_s
            return $ head stack
```

One of the technical challenges to define such a generic abstract data type is that we need to constrain both operations with respect to the permission lattices $\sqsupset_{RW}$ and $\sqsupset_{Ex}$. Indeed, this is not possible if the private lattices $\sqsupset_{RW}$ and $\sqsupset_{Ex}$ were implemented as a “secret” (*i.e.* not exported) type class. Recall from Section 4.2 the definition of $\sqsupset_{RW}$:

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

This definition requires that for every instance $a \sqsupset_{RW} b$ *there must exists another instance* $a \sqsupset_{RW} b$. When we use a specific capability, like *QState*, the typechecker can inspect the existing type class instances and verify that this condition holds.

However, this is not true in general for an arbitrary capability c. As the type checker cannot verify it, it prompt us to include a constraint $a \sqsupset_{RW} b$ in the definition of *push* — but the type class $\sqsupset_{RW}$ is secret and we cannot make it public without completely breaking permissions! Fortunately, one way out of this problem is to use a closed type family to represent the private lattice of permissions. Other encodings that would allows us to have type classes with closed instances, but with public visibility should also work.

Finally, the following example shows how to use two different instances of a stack, based on the generic implementation:

```
type M = TStateTP (QState₁ ()) [Int] (TStateTP (QState₂ ()) [Bool]
           (TErrorTP (QError₁ ()) String (TErrorTP (QError₂ ()) String Identity)))
program :: M (Int, Bool)
program =
  do push s₁ 10
     x ← pop s₁
     push s₂ True
     y ← pop s₂
     return (x, y)
  where
    s₁ = mkStackHandle tagQS₁ tagQE₁ (QState₁ RWPerm) (QError₁ TCPerm)
    s₂ = mkStackHandle tagQS₂ tagQE₂ (QState₂ RWPerm) (QError₂ TCPerm)
    tagQS₁ = structure (tag :: QState₁ ())
    tagQS₂ = structure (tag :: QState₂ ())
    tagQE₁ = structure (tag :: QError₁ ())
    tagQE₂ = structure (tag :: QError₂ ())
```

In this program there are two state layers, $QState_1$ and $QState_2$, and two exception layers, $QError_1$ and $QError_2$. Assuming that these types are also declared elsewhere as capabilities, the *program* defines two stack handles, s_1 and s_2, and performs some operations on them. The final result of *program* is the pair $(10, true)$.

7. Related work

There exists an extensive literature regarding side-effects and interference control. Two main approaches based on types are monads [17,31], type-and-effect systems [26], and their connections, like [33]. We believe that an expanded discussion and comparison is beyond the scope of this work, hence we limit ourselves to selected and recent related works, particularly in the context of Haskell and functional languages. In addition we review a few other approaches to control I/O and exception effects.

7.1. Alternatives to classic monads and monad transformers

Extensible effects (EE) [11] proposes an alternative representation of effects, in Haskell, that is not based on monads or monad transformers, and which can subsume 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 effectful 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 describe 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 while avoiding interference, the user is required to define a wrapper using a **newtype** declaration. This means that each effect can be uniquely identified by its type.

The *Effects* [1] library is an effect system implemented in the dependently-typed language Idris, based on algebraic effect handlers, and also designed 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, the Effects library does not address the issue of controlling the access to effects. Any available effect can be used by any part of the system. References to copies of a same effect (*e.g.* two integer states) are disambiguated using labels in the effect-tracking list.

In a recent development, Orchard and Petricek describe how to embed user-defined effect systems in Haskell [18] using *parametric effect monads*. A parametric effect monad uses *effect indices* to give a more detailed description of the effects of a computation. The annotations have a monoid structure $(F, \bullet, I)$ with the pure effect I, the set of effect indices F and the effect composition operator $\bullet$. Adapting the code from [18], the usual monadic operations *return* and $\gg\!=$ are augmented with annotations as follows:

$$return :: a \to M_I\, a$$
$$(\gg\!=) :: M_G\, a \to (a \to M_H\, b) \to M_{G \bullet H}\, b$$

for effect sets G and H. This definitions state that lifting a value to a computation incurs no computational effect, hence its type is indexed by I, and that the actual effects performed by $\gg\!=$ are described by the composition of the effects G used by its first argument, and the effects H used by the function argument.

Effect capabilities are orthogonal to the mechanism used to implement effects, because they protect specific effectful operations. Recent mechanisms like EE and Effects focus on how to enable flexible composition of effects — which is a well-known drawback of monad transformers — rather than on controlling access to them. We have shown how to apply effect capabilities to control access to effects in the context of "classic" monad transformers, mainly as a solution to known interference issues. We believe that the same approach should be applicable to other effect mechanisms.

7.2. Controlling I/O operations

Perhaps the most well-known approach for fine-grained I/O access in Haskell is the work of Wouter Swierstra: *Data Types à la Carte* (DLC) [25]. This work is proposed as a solution to the *expression problem* posed by Philip Wadler, which is related to the modular extensibility of a language regarding type extensions and new functions over an existing data type. The solution in DLC is based on the use of *open* expressions — which are not fixed to any specific type constructor — and the composition of *coproducts* to expand the possible types of valid expressions. For instance, an expression of type $Expr\ (Val :+: Add)$ is either a value or the sum of two such expressions. Here, $:+:$ is the coproduct composition whereas *Val* and *Add* are independent definitions of different kinds of expressions. This technique is applied to split the *IO* monad, regarded as a "sin bin" [25] of effectful operations, into specific classes of operations. The article provides an example of the *cat* program, which is defined over expressions that only use the *putChar*, *getChar*, *readFile* and *writeFile* I/O operations.

In general, our work is not based on the ideas of DLC, although the results are similar. The main difference is that DLC can be used to effectively split I/O effects into several classes of operations, but once these classes are defined, a programmer can still use all available effects. On the other hand, the protected *IOP* monad makes all operations visible, but restricted by the permission-lattice specified in Fig. 8.

7.3. Controlling exception interference

As we mentioned in 3.3, the conflicts of exception interference have been identified before in the context of aspect-oriented programming [5]. Two solutions to this issue, in the field of aspect-oriented programming, rely on the same essential idea: the need to establish different communication channels for exceptions. This idea is implemented in [2] with a sophisticated pointcut and advice mechanism; in other approaches, exceptions raised at a certain *execution level* [29,28] — a dynamically-scoped runtime parameter — can only be caught by handlers at the same level.

Protected exceptions use the same essential idea: a monad stack can be defined with several independent exception layers, which can only be accessed with the proper capability.

8. Discussion

8.1. Technology summary

The implementation of effect capabilities in Haskell relies on several important language mechanisms that allow us to enforce static type-based properties without custom modifications to the underlying type system. The fundamental mechanism for embedding type-level computation is provided by Haskell type classes. Indeed, as reported by McBride [15], the introduction of multi-parameter type classes with functional dependencies opens the door to using Haskell to *fake* part of the expressive power of dependently-typed languages. Since then further improvements to the language bring the expressiveness of type-level guarantees in Haskell closer to that given by dependently-typed languages. In particular, we fundamentally rely on four main mechanisms: multi-parameter type classes, functional dependencies, type families and constraint kinds.

Multi-parameter type classes and functional dependencies A multi-parameter type class $R\ t_1 \ldots t_n$ can be seen as a *relation* R on types $t_1 \ldots t_n$, and **instance** declarations as ways to inductively define this relation, in a manner very similar to logic programming. This technology enables the encoding of arbitrary relations at the type-level, which are then checked statically by the type system. Furthermore, a functional dependency [10] (like in database theory), denoted as expression $m \to s$, expresses that the type of m uniquely determines the type of s. This avoids type ambiguity issues because it can be used as a constraint to provide more precise control of type inference.

We are forced to use this technology because the implementation of the standard monad transformers and monad classes already use it. In addition, the definition of type classes *Capability*, *Send* and the permission lattices rely on multi-parameter type classes. More specifically, the *Capability* class also uses functional dependencies. Recall the definition (Section 4.4):

$$\textbf{class}\ Capability\ c\ \supset \mid c \to \supset$$

meaning that each capability is uniquely bound to a permission lattice, and therefore the capability type directly determines that permission lattice.

Closed type families and constraint kinds Closed type families and constraint kinds are crucial to implement user-definable permission lattices that cannot be bypassed by external modules and that can be used as an abstract constraint, as explained in Section 6.3. Let us recall that a permission lattice is encoded as follows:

```
type family (a ⊐ b) :: Constraint where
    -- particular instances
  a ⊐ a = ()   -- reflexivity
class a ⊐ b ⇒ a ⊃ b
```

Here, the closed family ⊐ defines the specific relation describing the permission lattice. However, as a type family cannot normally be used as a class constraint, this is not enough for our purposes. Therefore we add the *Constraint* kind annotation to ⊐, which enables the definition of ⊃. This definition states that $a \supset b$ holds only when $a \sqsupset b$ holds. The advantage is that ⊃ can be exported without fear because no external module can add instances to the closed type family, meaning that no further instances of ⊃ can be externally defined.

If either of these extensions were not available, the implementation of permission lattices would require a private type class ⊐. However, this restricts the usage of ⊃ as a class constraint, as explained in Section 6.3, meaning that it would not be possible to implement generic abstract data types. Only "concrete" abstract data types, bound to a specific capability, can be defined in this setting.

8.2. Limitations

We identify several drawbacks or limitations of effect capabilities. For instance, we have not considered the unsafe mechanisms that break proper module boundaries in Haskell, *e.g.* the use of *unsafePerformIO* or generic programming to access private data constructors. Another drawback is that the current usage of capabilities is very verbose, as it requires several type and instance declarations to be coherently defined. Similarly, the sharing mechanism requires mutually recursive modules, thus hampering modularity and independent development of software components. Finally, using effect capabilities requires explicit type signatures in the protected operations and in calls to *receive*, slightly hampering the benefits of type inference.

As a more general reflection, we need to recall that type-level programming in Haskell is only an approximation to the expressive power of dependent types. In such a type system, regular values can be used to index types, similar to how type families are indexed by type constructors. However, Haskell is not a dependently-typed language, meaning that there is a strict distinction between the type-level and the value-level. In practice, this means that access policies to protected operations are limited to the capabilities of Haskell's type system. Nevertheless, the design principles of effect capabilities should apply also in the context of dependently-typed languages.

8.3. Effect capabilities beyond Haskell

Despite the Haskell-specific technologies used in the implementation of effect capabilities, we believe that the conceptual mechanism is more broadly applicable: access to a protected operation depends on presenting a proper access token. This mechanism can be applied either to dynamically- or statically-typed languages and is orthogonal to a specific programming paradigm. The essence of the problem is the creation and sharing of those tokens. Crucially, the fundamental requirement in any language with capability-based security is the unforgeability of capabilities. Given this requirement, sharing capabilities is only a matter of (inter-module) scoping, that is, of controlling the visibility of tokens or token constructors.

9. Future work

We identify several directions for future work, mainly with the focus on overcoming the aforementioned limitations. 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. Even worse, arbitrary I/O operations can be posed as "pure" by using the highly controversial *unsafePerformIO* operation. Recently, Safe Haskell [30] has been proposed as an extension to Haskell, implemented in GHC (as of version 7.2), which protects referential transparency and module boundaries by disabling the use of these unsafe features. Because the privacy and unforgeability 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. Indeed, an initial application of generic programming is already used in the downloadable implementation. A complementary approach is using Template Haskell [23], a template meta-programming facility for Haskell. Note this does not conflict with the usage of Safe Haskell, because specific modules can be explicitly marked as trustworthy.

Another important line of work is related to modularity. The main impact of effect capabilities on modularity is the use of mutually dependent modules due to the static secret sharing mechanism. All involved modules must have access to the communication channels and to instances of the *Send* type class. The issue is specially complex for the case of protected I/O, because many different modules may require access to I/O permissions. Definitely it is not practical to have a big monolithic module that grants those permissions. As a solution we envision the development of a compiler plugin or some other external tool that combines code generation with an external registry of shared capabilities and permissions. Perhaps this can be similar to how Safe Haskell manually manages the trustworthiness of modules using a command-line tool.

Regarding the expressive power of effect capabilities, it would be interesting to study how to leverage the features of dependently-typed languages in order to define more expressive access policies. This also paves the road for a rigorous formalization of effect capabilities, for example in the context of recent developments on monadic meta-theory [6]. In addition, we consider the application of effect capabilities for other kinds of computations, like non-determinism, concurrency or continuations. Finally, it remains to be studied how effect capabilities can be provided in programming languages with imperative features but without explicit effects. An important insight to accomplish this goal is that capability-based access is essentially a matter of scoping, which can be addressed with modern solutions such as *scoping strategies* [27].

Acknowledgements

This work was supported by the Inria Associated Team REAL. We thank the anonymous reviewers of SBLP and SCP for their insightful and constructive comments.

References

[1] E. Brady, Programming and reasoning with algebraic effects and dependent types, in: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13, ACM, New York, NY, USA, 2013, pp. 133–144.
[2] N. Cacho, F.C. Filho, A. Garcia, E. Figueiredo, EJFlow: taming exceptional control flows in aspect-oriented programming, in: Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development, AOSD 2008, ACM Press, Brussels, Belgium, Apr. 2008, pp. 72–83.
[3] M.M.T. Chakravarty, G. Keller, S.P. Jones, Associated type synonyms, in: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, ICFP '05, ACM, New York, NY, USA, 2005, pp. 241–253.
[4] M.M.T. Chakravarty, G. Keller, S.P. Jones, S. Marlow, Associated types with class, in: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '05, ACM, New York, NY, USA, 2005, pp. 1–13.
[5] R. Coelho, A. Rashid, A. Garcia, N. Cacho, U. Kulesza, A. Staa, C. Lucena, Assessing the impact of aspects on exception flows: an exploratory study, in: J. Vitek (Ed.), Proceedings of the 22nd European Conference on Object-oriented Programming, ECOOP 2008, Paphos, Cyprus, July 2008, in: Lecture Notes in Computer Science, vol. 5142, Springer-Verlag, 2008, pp. 207–234.
[6] B. Delaware, S. Keuchel, T. Schrijvers, B.C.d.S. Oliveira, Modular monadic meta-theory, in: Proceedings of the 18th ACM SIGPLAN Conference on Functional Programming (ICFP 2013), ACM Press, Boston, MA, USA, Sept. 2013.
[7] R. Harper, Exceptions are shared secrets, http://existentialtype.wordpress.com/, Dec. 2012.
[8] R. Harper, Practical Foundations for Programming Languages, Cambridge University Press, 2012.
[9] J. Hughes, Global variables in Haskell, J. Funct. Program. 14 (5) (Sept. 2004) 489–502.
[10] M.P. Jones, Type classes with functional dependencies, in: Proceedings of the 9th European Symposium on Programming Languages and Systems, in: Lecture Notes in Computer Science, vol. 1782, Springer-Verlag, 2000, pp. 230–244.
[11] O. Kiselyov, A. Sabry, C. Swords, Extensible effects: an alternative to monad transformers, in: Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell, Haskell '13, ACM, New York, NY, USA, 2013, pp. 59–70.
[12] D. Leijen, E. Meijer, Domain specific embedded compilers, in: T. Ball (Ed.), Proceedings of the 2nd USENIX Conference on Domain-Specific Languages, 1999, pp. 109–122.
[13] H.M. Levy, Capability-Based Computer Systems, vol. 12, Digital Press, Bedford, Massachusetts, 1984.
[14] S. Liang, P. Hudak, M. Jones, Monad transformers and modular interpreters, in: Proceedings of the 22nd ACM Symposium on Principles of Programming Languages, POPL 95, San Francisco, California, Jan. 1995, ACM Press, 1995, pp. 333–343.
[15] C. McBride, Faking it simulating dependent types in Haskell, J. Funct. Program. 12 (5) (July 2002) 375–392.
[16] M.S. Miller, Robust composition: towards a unified approach to access control and concurrency control, PhD thesis, John Hopkins University, Baltimore, Maryland, USA, May 2006.
[17] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1) (July 1991) 55–92.
[18] D. Orchard, T. Petricek, Embedding effect systems in Haskell, in: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell, Haskell '14, ACM, New York, NY, USA, 2014, pp. 13–24.
[19] D. Piponi, Tagging monad transformer layers, http://blog.sigfpe.com/2010/02/tagging-monad-transformer-layers.html, 2010.
[20] J.H. Saltzer, M.D. Schroeder, The protection of information in computer systems, 1975.
[21] D. Sangiorgi, D. Walker, The Pi-Calculus: A Theory of Mobile Processes, Cambridge University Press, 2003.
[22] T. Schrijvers, B.C. Oliveira, Monads, zippers and views: virtualizing the monad stack, in: Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming, ICFP 2011, Tokyo, Japan, Sept. 2011, ACM Press, 2011, pp. 32–44.
[23] T. Sheard, S.P. Jones, Template meta-programming for Haskell, SIGPLAN Not. 37 (12) (Dec. 2002) 60–75.
[24] M. Snyder, P. Alexander, Monad factory: type-indexed monads, in: Proceedings of the 11th International Conference on Trends in Functional Programming, 2010, pp. 198–213.
[25] W. Swierstra, Data types á la carte, J. Funct. Program. 18 (4) (July 2008) 423–436.
[26] J.-P. Talpin, P. Jouvelot, The type and effect discipline, Inf. Comput. 111 (2) (June 1994) 245–296.
[27] É. Tanter, Beyond static and dynamic scope, in: Proceedings of the 5th ACM Dynamic Languages Symposium, DLS 2009, Orlando, FL, USA, Oct. 2009, ACM Press, 2009, pp. 3–14.
[28] É. Tanter, Execution levels for aspect-oriented programming, in: Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development, AOSD 2010, Rennes and Saint Malo, France, Mar. 2010, ACM Press, France, 2010, pp. 37–48.
[29] É. Tanter, I. Figueroa, N. Tabareau, Execution levels for aspect-oriented programming: Design, semantics, implementations and applications, Sci. Comput. Program. 80 (1) (Feb. 2014) 311–342.

[30] D. Terei, S. Marlow, S. Peyton Jones, D. Mazières, Safe Haskell, in: Proceedings of the 5th ACM Symposium on Haskell, ACM, 2012, pp. 137–148.
[31] P. Wadler, The essence of functional programming, in: Proceedings of the 19th ACM Symposium on Principles of Programming Languages, POPL 92, Albuquerque, New Mexico, USA, Jan. 1992, ACM Press, 1992, pp. 1–14.
[32] P. Wadler, How to declare an imperative, ACM Comput. Surv. 29 (3) (Sept. 1997) 240–263.
[33] P. Wadler, The marriage of effects and monads, ACM SIGPLAN Not. 34 (1) (Sept. 1998) 63–74.