Parameterized Extensible Effects and Session Types

Extended Abstract

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Abstract

Parameterized monad goes beyond monads in letting us represent type-state. An effect executed by a computation may change the set of effects it may be allowed to do afterwards. We describe how to easily 'add' and 'subtract' such type-state effects. Parameterized monad is often used to implement session types. We point out that extensible type-state effects are themselves a form of session types.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives

1. Introduction

The type of a value determines the set of operations on it. The set may be further restricted by the computation state: e.g., read should not be applied to a file handle if it has been closed. *Type-state* is the refinement of the notion of type to characterize such restriction [6]. Session types, in the news lately, are one manifestation of the type-state. Whereas monads represent stateful computations, type-state calls for a *parameterized monad* [1, 3], as a poor programmer's substructural type system.

Monad transformers – and recently, algebraic, extensible and other 'effects' – let us combine in the same program independently developed monadic computations, each with their own side-effects. The effects may need to interact in a controlled fashion. The similar combining of parameterized monads has not been addressed so far. The problem is not straightforward, as we shall see from several failed attempts. We eventually present the solution, along the lines of extensible effects [4]. The inferred types curiously look like session types.

For clarity, this abstract uses a rather specific example: two mutable cells. The generalizations, to other effects and to truly open unions, and optimizations are all doable and have been already described [2, 4]. We elide them to demonstrate the problem and the solution in the purest, barest form.

The complete source code is available at `http://okmij.org/ftp/Haskell/extensible/ParamEff1.hs`. Therefore, the code in the abstract is heavily abbreviated.

The running example We borrow the motivating example of a state-varying computation from Mezzo. Its pseudo-code is shown below. In the end, we write such code in Haskell, type-safely.

```
let x = ref ()        -- allocate with a placeholder
-- let y = !x + 1     -- type error: not initialized
x := 1                -- updated with a real value, and type
let y = !x + 1
x : = "str"           -- new value and type
let z = !x ++ "ing"
```

Contrived the code is, it illustrates the real-life "initialization problem" (for which type-state was initially developed, [5]).

2. Ordinary State Monad and Extensible Effects

As a warm-up, the section describes the simple generalization of the familiar State Monad to support several independent pieces of state. We use the simplicity of the example to demonstrate the extensible effects, setting the stage for later.

Computations that access and modify a named piece of mutable state of the type s are described by the following interface, which is the trivial extension of MonadState, well-known from the monad transformer library.

```
class Monad m ⇒ MonadMState var s m | var m → s where
  modify :: var → (s→ s) → m s
```

We add the variable name var, that together with the monad m uniquely determines the type of the variable state s. The interface supports several variables, with distinct types, whose names must be statically known:

```
data Var1 = Var1; data Var2 = Var2
```

For the sake of explanation, we chose modify as primitive: it takes as the argument the state transformer function and applies it to the current state, returning the original. The familiar get and put operations are easily expressible in terms of modify.

Below are two examples of state computations. The shown signatures have been inferred.

```
ts1 :: MonadMState Var1 Char m ⇒ m Char
ts1 = do {x ← liftM succ $ get Var1; put Var1 'c'; return x}

ts2 :: MonadMState Var2 Int m ⇒ m Int
ts2 = do
  put Var2 (10:: Int ); x ← get Var2
  put Var2 20;          y ← get Var2
  return (x+y)
```

Moreover, we easily combine ts1 and ts2 in the same program:

```
ts3 :: (MonadMState Var1 Char m, MonadMState Var2 Int m) ⇒
       m (Char, Int , Char)
ts3 = do {x ←ts1; y ← ts2; z ← ts1; return (x, y, z)}
```

The inferred signature tells that ts3 is the computation over two pieces of mutable state with their own types.

Extensible Effects To actually implement the MonadMState interface, we use the bare-bones Freer monad [2], eliding for clarity any optimizations. Eff f is certainly a monad, for every f :: $*\to*$, which does not have to be a functor (and it is not, in our examples).

```
data Eff f a where
  Pure   :: a → Eff f a
  Impure :: f x → (x → Eff f a) → Eff f a
```

The parameter f is the signature of the effect represented by the Eff f monad; for state, it is State var s:

```
data State var s x where
  Modify :: (s→ s) → State var s s
```

Modify tr is hence the request to modify the current state according to tr and return the result of type s. The request is handled by an interpreter, such as

```
runState :: var → s → Eff (State var s) a → (a, s)
```

which runs the state effectful computation and returns its final result and the final state.

Eff State indeed implements the MonadMState interface:

```
instance MonadMState var s (Eff (State var s)) where
  modify _ tr   = Impure (Modify tr) Pure
```

letting us run the examples ts1, and ts2, as in runState Var2 0 ts2 – but not ts3 that has two different state effects. We have to define how to combine effects.

The computation Eff f1 propagates requests of the signature f1 and delivers them to their interpreter; the same for Eff f2. The combined computation hence should be able to propagate both sorts of requests, either for an f1 action or for an f2 action. We thus need a union of the two signatures. There are many sophisticated and efficient ways of building such unions, described in [2, 4]. Here we use the dumbest but the clearest one, the ordinary sum:

```
data Sum f1 f2 x = L (f1 x) | R (f2 x)
```

The following interpreter (code elided) deals with state requests in the first component of the union, leaving the other requests for their own interpreter.

```
runState1 :: var → s → Eff (Sum (State var s) f2) a →
             Eff f2 (a, s)
```

The unioned signature that includes State var s clearly still implements MonadMState:

```
instance MonadMState var s (Eff (Sum (State var s) f2)) where
  modify _ tr   = Impure (L (Modify tr)) Pure
```

(and likewise for the R invariant). We now can run the combined computation (runState Var2 0 ∘ runState1 Var1 'a') ts3, whose effects are handled by the composition of two interpreters.

Variable-state Effect We have learned how to handle modifications for multiple mutable cells each with its own state type. The modification updates the value of the state but preserves its type. Attempting to update the type:

```
tsx = do {put Var1 'c'; put Var1 True}
```

raises a type error. Each put operation gives rise to a constraint, MonadMState Var1 Char m and MonadMState Var1 Bool m, implicitly conjoined in the type of tsx. The conjunction is clearly unsatisfiable, which is the source of the type error. Without doubt, the current framework does not support variable-state computations: in order for tsx for type-check, the latter constraint should hold only *after* the former is asserted. We need some notion of 'sequencing', to reflect the evolution of the computational state in types – which is what session types (and type-state, introduced much earlier) are set to accomplish.

3. Extensible Parameterized Monad

A parameterized monad m s t is the well-known way to reflect the evolution of the computational state in types:

```
class Monadish m where
  (⋙=) :: m s t a → (a → m t u b) → m s u b
```

It looks like a State monad with two parameters: the type state before s and after t its action. If the two parameters are the same, m s s is the ordinary monad. An example is the Freer parameterized monad, the straightforward generalization of Eff:

```
data EffP f s1 s2 a where
  PureP   :: a → EffP f s s a
  ImpureP :: f s1 s x → (x → EffP f s s2 b) → EffP f s1 s2 b
```

The parameter f is the action signature, e.g., for the case of state:

```
data StateP var s1 s2 x where
  ModifyP :: (s1→ s2) → StateP var s1 s2 s1
```

What remains to support multiple state-varying effects is to find a way to union action signatures and to generalize MonadMState to parameterized monads. This is not easy. For example, the straightforward generalization

```
class Monadish m ⇒ MonadMPState var s1 s2 m where
  modifyP :: var → (s1→ s2) → m s1 s2 s1
```

does not work: it assumes that the state of one var is the overall type-state. In reality, the var's state is only a part of the type-state that tracks other vars and other effects. To find var's state, we have to project it out from the type-state:

```
class Monadish m ⇒
  MonadMPState var s1 s2 t1 t2 m
     | var m t1 → s1, var m t1 → s2, var m t1 s2 → t2 where
  modifyP :: var → (s1→ s2) → m t1 t2 s1
```

One may read the 6-place relation MonadMPState as: the security policy m in the type-state t1 permits to access the var's state as s1 and change it to s2, advancing the type-state to t2. As before, putP is implemented in terms of modifyP. Normally, reading the state does not change its value, and hence its type. Therefore, we may continue to use get: a type-sate–preserving parameterized monad is the ordinary monad, and the simple MonadMState still applies. Likewise we can use the old put for a type-preserving update.

MonadMPState suggests the way to union parameterized action signatures, which now requires GADTs. For the lack of space, we refer to the accompanying code.

The sample two-variable type-varying code (in the do-notation)

```
tsP2 = do
 x ← get Var1   −− of the non−parameterized MonadMState
 y ← get Var2   −− of the non−parameterized MonadMState
 putP Var2 'c'
 putP Var1 $ show (x+y+1:: Int)
```

gets the inferred type

```
tsP2 :: (Monad (m t1 t1), Monad (m u u),
   MonadMState Var1 Int (m t t), MonadMState Var2 Int (m t t),
   MonadMPState Var2 Int Char t t1 m,
   MonadMPState Var1 a String t1 u m) ⇒ m t u ()
```

The type looks like a session type: In the initial type-state t, Var1 and Var2 store Int values. Var2 is then updated, including its type, from Int to Char; the type state correspondingly changes from t to t1. In the new type-state, Var1 is updated to a string, moving the final type-state u. If we switch the third and fourth lines of the code, placing putP ahead of get, the code no longer compiles, since in the type-state t1 the variable Var2 no longer has the type Int needed for y. The motivating example is implemented similarly (see the accompanying code).

The shown example is deliberately simple. One may contemplate various extensions: for example, ensuring that updating and reading of the state strictly alternate. (In this case, reading the variable is no longer type-state preserving.) Parameterized monad were first mentioned in the context of the general (answer-type modifying) delimited continuations. It is interesting to investigate what is the continuation dual of the extensible parameterized monad.

References

[1] R. Atkey. Parameterised notions of computation. In *MSFP 2006*

[2] O. Kiselyov and H. Ishii. Freer monads, more extensible effects. In *Haskell*, pages 94–105. ACM, 2015.

[3] O. Kiselyov and C.-c. Shan. Lightweight static resources: Sexy types for embedded and systems programming. In *Draft Proc. TFP 2007*.

[4] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Haskell*, pages 59–70. ACM, 2013.

[5] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. SEng.*, 19(5):478–485, 1993.

[6] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.