The Key Monad: Type-Safe Unconstrained Dynamic Typing

Atze van der Ploeg Koen Claessen
Chalmers University of Technology, Sweden
{atze, koen}@chalmers.se

Pablo Buiras
Harvard University, United States
pbuiras@seas.harvard.edu

Abstract

We present a small extension to Haskell called the Key monad. With the Key monad, unique keys of different types can be created and can be tested for equality. When two keys are equal, we also obtain a concrete proof that their types are equal. This gives us a form of dynamic typing, without the need for *Typeable* constraints. We show that our extension allows us to safely do things we could not otherwise do: it allows us to implement the ST monad (inefficiently), to implement an embedded form of arrow notation, and to translate parametric HOAS to typed de Bruijn indices, among others. Although strongly related to the ST monad, the Key monad is simpler and might be easier to prove safe. We do not provide such a proof of the safety of the Key monad, but we note that, surprisingly, a full proof of the safety of the ST monad also remains elusive to this day. Hence, another reason for studying the Key monad is that a safety proof for it might be a stepping stone towards a safety proof of the ST monad.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Functional programming, Haskell, Higher-order state, ST monad, Arrow notation, Parametric HOAS

1. Introduction

The ST monad (Launchbury and Peyton Jones 1994) is an impressive feat of language design, but also a complicated beast. It provides and combines three separate features: (1) an abstraction for *global memory references* that can be efficiently written to and read from, (2) a mechanism for embedding computations involving these memory references in *pure computations*, and (3) a design that allows references in the same computation to be of *arbitrary, different types*, in a type-safe manner.

In this paper, we provide a new abstraction in Haskell (+ GADTs and rank-2 types) that embodies only feature (3) above: the combination of references (which we call *keys*) of different, unconstrained types in the same computation. In the ST monad, the essential invariant that must hold for feature (3), is that when two references are the same, then their types *must also be the same*. Our new abstraction splits reasoning based on this invariant into a separate interface, and makes it available to the user. The result is a small library called *the Key monad*, of which the API is given in Figure 1.

Haskell'16, September 22-23, 2016, Nara, Japan
ACM. 978-1-4503-4434-0/16/09...$15.00
http://dx.doi.org/10.1145/2976002.2976008

```
type KeyM s a
type Key s a
instance Monad (KeyM s)
newKey      :: KeyM s (Key s a)
testEquality :: Key s a → Key s b → Maybe (a :~: b)
runKeyM     :: (∀ s. KeyM s a) → a
data a :~: b where Refl :: a :~: a
```

Figure 1. The Key monad interface

The Key monad $KeyM$ is basically a crippled version of the ST monad: we can monadically create keys of type $Key\ s\ a$ using the function $newKey$, but we cannot read or write values to these keys; in fact, keys do not carry any values at all. We can convert a computation in $KeyM$ into a pure value by means of $runKeyM$, which requires the argument computation to be polymorphic in s, just like $runST$ would.

The only new feature is the function $testEquality$, which compares two keys for equality. But the keys do not have to be of the same type! They just have to come from the same $KeyM$ computation, indicated by the s argument. If two keys are not equal, the answer is $Nothing$. However, if two keys are found to be equal, *then their types must also be the same*, and the answer is $Just\ Refl$, where $Refl$ is a constructor from the GADT $a :\sim: b$ that functions as the "proof" that a and b are in fact the same type[1]. This gives us a form of dynamic typing, *without* the need for $Typeable$ constraints.

Why is the Key monad interesting? There are three separate reasons.

First, the Key monad embodies the insight that when two Keys are the same then their types must be the same, and makes reasoning based on this available to the user via $testEquality$. This makes the Key monad applicable in situations where the ST monad would not have been suitable. In fact, the bulk of this paper presents examples of uses of the Key monad that would have been impossible without $testEquality$.

Second, the Key monad is simpler than the ST monad, because it does not involve global references, or any updatable state at all. We would like to argue that therefore, the Key monad is easier to understand than the ST monad. Moreover, given the Key monad, the ST monad is actually implementable in plain Haskell, albeit in a less

[1] It is actually possible to add $testEquality$ to the standard interface of $STRef$s, which would provide much the same features in the ST monad as the Key monad would, apart from some laziness issues. However, because of its simplicity, we think the Key monad is interesting in its own right.

time- and memory-efficient way than the original ST monad (that is, missing feature (1) above, but still providing features (2) and (3)).

This second reason comes with a possibly unexpected twist.

After its introduction in 1994, several papers have claimed to establish the safety, fully or partially, of the ST monad in Haskell (Launchbury and Peyton Jones 1994; Launchbury and Sabry 1997; Ariola and Sabry 1998; Moggi and Sabry 2001). By safety we mean three things: (a) type safety (programs using the ST monad are still type safe), (b) referential transparency (programs using the ST monad are still referentially transparent), and (c) abstraction safety (programs using the ST monad still obey the parametricity theorem). It came as a complete surprise to the authors that *none of the papers we came across in our literature study actually establishes the safety of the* ST *monad in Haskell!*

So, there is a third reason for studying the Key monad: A safety proof for the Key monad could be simpler than a safety proof for the ST monad. The existence of such a proof would conceivably lead to a safety proof of the ST monad as well; in fact this is the route that we would currently recommend for anyone trying to prove the ST monad safe.

This paper does not provide a formal safety proof of the Key monad. Instead, we will argue that the safety of the Key monad is just as plausible as the safety of the ST monad. We hope that the reader will not hold it against us that we do not provide a safety proof. Instead, we would like this paper to double as a call to arms, to find (ideally, mechanized) proofs of safety of both the Key monad and the ST monad!

Our contributions are as follows:

- We present the Key monad (Section 2), its implementation (Section 6), and informally argue for its safety (Section 5).
- We show that the added power of the Key monad allows us to do things we cannot do without it, namely it allows us
  - to implement the ST monad (Section 2);
  - to implement an *embedded* form of arrow notation (Section 3);
  - to represent typed variables in typed representations of syntax (Section 4);
  - to translate parametric HOAS to nested de Bruijn indices, which allows interpretations of parametric HOAS terms, such as translation to Cartesian closed categories, which are not possible otherwise (Section 4).
- We identify some key missing pieces in the correctness proof of the ST monad (Section 7). The Key monad may be a stepping stone to finding the complete proof.

The Haskell code discussed in this paper can be found online at: `https://github.com/koengit/KeyMonad`

2. The Key Monad

In this section, we describe the Key monad, what it gives us, and its relation to the ST monad.

The interface of the Key monad (Figure 1) features two abstract types (i.e., types with no user-accessible constructors): Key and $KeyM$. The Key monad gives the user the power to create a new, unique value of type $Key\ s\ a$ via $newKey$. The only operation that is supported on the type Key is $testEquality$, which checks if two given keys are the same, and if they are returns a "proof" that the types associated with the names are the same types.

2.1 Unconstrained Dynamic Typing

The power to prove that two types are the same allows us to do similar things as with $Data.Typeable$, but *without* the need for $Typeable$ constraints. For instance, we can create a variant of $Dynamic$ using Keys instead of type representations. When given a key and value, we can "lock up" the value in a box, which, like $Dynamic$, hides the type of its contents.

```
data Box s where
  Lock :: Key s a → a → Box s
```

If we have a Key and a Box, we can try to unlock the box to recover the value it contains.

```
unlock :: Key s a → Box s → Maybe a
unlock k (Lock k' x) =
  case testEquality k k' of
    Just Refl → Just x
    Nothing   → Nothing
```

If we used the right key, we get $Just$ the value in the box, and we get $Nothing$ otherwise.

Note that the only way to unlock a Box successfully involves using the Key that was used to create it, not merely a Key with the right type. The function $testEquality$ only returns $Just\ Refl$ when the keys are identical, not when the types are the same. Thus, the Key monad does not provide a means of checking type equality at run-time.

2.2 Heterogeneous Maps

We can use Boxes to create a kind of *heterogeneous maps*: a data structure that that maps keys to values of the type corresponding to the key. The interesting feature here is that the type of these heterogenous maps does not depend on the types of the values that are stored in it, nor do the functions have $Typeable$ constraints. We can implement such maps straightforwardly as follows[2]:

```
newtype KeyMap s = KM [Box s]

empty :: KeyMap s
empty = KM []

insert :: Key s a → a → KeyMap s → KeyMap s
insert k v (KM l) = KM (Lock k v : l)

lookup :: Key s a → KeyMap s → Maybe a
lookup k (KM [])      = Nothing
lookup k (KM (h : t)) =
  unlock k h `mplus` lookup k (KM t)

(!) :: KeyMap s → Key s a → a
m ! k = fromJust (lookup k m)
```

2.3 Implementing the ST Monad

Armed with our newly obtained $KeyMap$s, we can implement an (inefficient) version of the ST monad as follows. The implementation of $STRef$s is simply as an alias for Keys:

```
type STRef s a = Key s a
```

We can now use the Key monad to create new keys, and use a $KeyMap$ to represent the current state of all created $STRef$s.

```
newtype ST s a =
  ST (StateT (KeyMap s) (KeyM s) a)
  deriving (Functor, Applicative, Monad)
```

It is now straightforward to implement the operations for $STRef$s:

```
newSTRef :: a → ST s (STRef s a)
newSTRef v = ST $
```

[2] For simplicity, this is a rather inefficient implementation, but a more efficient implementation (using $IntMap$s) can be given if we add a function $hashKey :: Key\ s\ a \to Int$ to the Key monad.

```
  do k ← lift newKey
     modify (insert k v)
     return k
readSTRef :: STRef s a → ST s a
readSTRef r = ST $ (! r) <$> get
writeSTRef :: STRef s a → a → ST s ()
writeSTRef k v = ST $ modify (insert k v)
```

Finally, the implementation of $runST$ simply runs the monadic computation contained in the ST type:

```
runST :: (∀ s. ST s a) → a
runST m = runKeyM $ case m of
  ST n → evalStateT n empty
```

2.4 Relation with the ST Monad

While the Key monad can be used to implement the ST monad, the converse is not true. The problem is that there is no function:

```
testEquality :: STRef s a → STRef s b →
                Maybe (a :∼: b)
```

It is straightforward to implement this function using $unsafeCoerce$:

```
testEquality x y
  | x ≡ unsafeCoerce y = Just (unsafeCoerce Refl)
  | otherwise          = Nothing
```

The $unsafeCoerce$ in $x \equiv unsafeCoerce\ y$ is needed because the types of the references might not be the same. Hence, another way to think of this paper is that we claim that the above function is *safe*, that this allows us to do things which we could not do before, and that we propose this as an extension of the ST monad library.

With the above $testEquality$ function for $STRef$s it is possible to implement something similar to the Key monad, but the Key monad is more lazy. In particular, for the Key monad, the following holds:

$$\bot \gg m \ \equiv\ m$$

This does not hold even for the lazy ST monad, which will give $\bot$ on for example $\bot \gg newSTRef\ 0$.

The extra laziness of the Key monad allows us to create an infinite list of Keys for example:

```
newKeys :: KeyM s [Key s a]
newKeys = liftM2 (:) newKey newKeys
```

This is something you can only do using $unsafeInterleaveST$ in the ST monad.

Why is the $testEquality$ function for $STRef$s safe? The reason is that if two references are the same, then their types must also be the same. This invariant must already be true for ST references, because otherwise we could have two references pointing to the same location with different types. Writing to one reference and then reading from the other would coerce the value from one type to another! Hence, the Key monad splits reasoning based on this invariant into a separate interface and makes it available to the user via $testEquality$.

Following the same line of reasoning, it is already possible to implement a similar, but weaker, version of $testEquality$ using only the standard ST monad functions. If we represent keys of type $Key\ s\ a$ as a pair of an identifier and an $STRef$ containing values of type a, then we can create a function that casts a value of type a to b, albeit monadically, i.e. we get a monadic cast function $a \to ST\ s\ b$ instead of a proof $a :\sim: b$:

```
data Key s a = Key { ident :: STRef s (),
                     ref :: STRef s a }
newKey :: ST s (Key s a)
newKey = Key <$> newSTRef () <*> newSTRef ⊥
testEqualityM :: Key s a → Key s b →
                 Maybe (a → ST s b)
testEqualityM ka kb
  | ident ka ≢ ident kb  = Nothing
  | otherwise            = Just $ λx →
      do writeSTRef (ref ka) x
         readSTRef (ref kb)
```

This implementation, although a bit brittle because it relies on strong invariants, makes use of the insight that if the two references are actually the same reference, then writing to one reference must trigger a result in the other.

2.5 Relation to $Typeable$

The `base` library $Data.Typeable$ provides similar functionality to the Key monad. Typeable is a type class that provides a value-level representation of the types that implement it. The $Typeable$ library provides a function

$$eqT :: \forall\ a\ b.\ (Typeable\ a, Typeable\ b) \Rightarrow Maybe\ (a :\sim: b)$$

where $(:\sim:)$ is the GADT from Figure 1. This function gives $Just\ Refl$ if both *types* are the same, whereas $testEquality$ from the Key monad only gives $Just\ Refl$ if the *keys* are the same. If we have two keys with the same type, but which originate from different $newKey$ invocations, the result of $testEquality$ will be $Nothing$.

Another difference is that with the Key monad, to obtain a key for a type a, we do not need a constraint on the type a, which we do need to get a value-level type representation using $Typeable$. These constraints can leak to the user-level interface. For example, we can also implement a variant of the ST monad using $Typeable$, by storing in each $STRef$ a unique number and a representation of its type. We will then need to change the interface such that we have access to the value-level type representations, by adding $Typeable$ constraints. For example, the type of $newSTRef$ then becomes

$$newSTRef :: Typeable\ a \Rightarrow a \to ST\ s\ (STRef\ s\ a)$$

In fact, all example usages of the Key monad in this paper can also be implemented by using $Typeable$ and unique numbers and adding constraints to the user interface. We could even implement the Key monad itself by adding a $Typeable$ constraint to $newKey$. However, using the Key monad has the benefit that it is *unconstrained*: we can use it even when $Typeable$ dictionaries are unavailable.

2.6 Key Monad Laws

Informally, the Key monad allows us to create new keys and compare them, maybe obtaining a proof of the equality of their associated types. To give a more precise specification and to allow equational reasoning, we also present the Key monad laws shown in Figure 2, which we will now briefly discuss.

The $sameKey$ and $distinctKey$ laws describe the behavior of $testEquality$. The $commutative$ law states that the Key monad is a commutative monad: the order of actions does not matter. The $purity$ law might be a bit surprising: it states that doing some Key computation and then throwing away the result is the same as not doing anything at all! The reason for this is that the only property of each key is that it is distinct from all other keys: making keys and then throwing them away has no (observable) effect on the rest of the computation.

The last two laws, $runReturn$ and $runF$, state how we can get the values out of a $KeyM$ computation with $runKey$. The $runF$ law states that we can lazily get the results of a (potentially)

$$
\begin{array}{lcll}
\textbf{do}\ k \leftarrow newKey & = & \textbf{do}\ k \leftarrow newKey & \\
\quad f\ (testEquality\ k\ k) & & \quad f\ (Just\ Refl) & (sameKey) \\
\textbf{do}\ k \leftarrow newKey & & \textbf{do}\ k \leftarrow newKey & \\
\quad l \leftarrow newKey & = & \quad l \leftarrow newKey & (distinctKey) \\
\quad f\ (testEquality\ k\ l) & & \quad f\ Nothing & \\
\textbf{do}\ x \leftarrow f & & \textbf{do}\ y \leftarrow g & \\
\quad y \leftarrow g & = & \quad x \leftarrow f & (commutative) \\
\quad h\ x\ y & & \quad h\ x\ y & \\
m \gg n & = & n & (purity) \\
runKey\ (return\ x) & = & x & (runReturn) \\
runKey\ (f <\$> m) & = & f\ (runKey\ m) & (runF) \\
\multicolumn{4}{l}{\quad (\text{if } m :: \forall\ s.\ KeyM\ s\ a)}
\end{array}
$$

Figure 2. Key monad laws

```
class Arrow a where
  arr   :: (x → y) → a x y
  (>>>) :: a x y → a y z → a x z
  first  :: a x y → a (x, z) (y, z)
  second :: a x y → a (z, x) (z, y)
  second x = flip >>> first x >>> flip
    where flip = arr (λ(x, y) → (y, x))
```

Figure 3. The arrow type class.

infinite $KeyM$ computation. The side condition that m has type $\forall\ s.\ KeyM\ s\ a$ (for some type a) rules out wrong specialization of the law, such as:

$$runKey\ (f <\$> newKey) = f\ (runKey\ newKey)$$

This specialization does *not* hold because, the left hand side typechecks, but the right hand side does not: the "s" would escape.

3. Embedding Arrow Notation

In this section, we show that the Key monad gives us the power to implement an *embedded* form of *arrow syntax*. Without the Key monad, such syntax is, as far as we know, only possible by using specialized compiler support.

3.1 Arrows vs Monads

The $Arrow$ type class, shown in Figure 3, was introduced by Hughes (Hughes 2000) as an interface that is like monads, but which allows for more static information about the constructed computations to be extracted. However, in contrast to monads, arrows do not directly allow intermediate values to be *named*; instead, expressions must be written in *point-free style*.

As an example, an arrow computation which feeds the same input to two arrows, and adds their outputs, can be expressed in point-free style as follows:

```
addA :: Arrow a ⇒ a x Int → a x Int → a x Int
addA f g = arr (λx → (x, x)) >>> first f >>>
           second g          >>> arr (λ(x, y) → x + y)
```

With monads, a similar computation can be written more clearly by naming intermediate values:

```
addM :: Monad m ⇒ (x → m Int) → (x → m Int) →
                  (x → m Int)
addM f g = λz →
  do x ← f z
     y ← g z
     return (x + y)
```

To overcome this downside of arrows, Paterson introduced arrow notation (Paterson 2001). In this notation, the above arrow computation can be written as follows:

```
addA :: Arrow a ⇒ a b Int → a b Int → a b Int
addA f g = proc z → do
  x ← f ≺ z
  y ← g ≺ z
  returnA ≺ x + y
```

Specialized compiler support is offered by GHC, which desugars this notation into point free expressions.

With the Key monad, we can name intermediate values in arrow computations using *regular* monadic do notation, without relying on specialized compiler support. The $addA$ computation above can be expressed using our *embedded* arrow notation as follows:

```
addA :: Arrow a ⇒ a b Int → a b Int → a b Int
addA f g = proc $ λz → do
  x ← f ≺ z
  y ← g ≺ z
  return $ (+) <$> x <*> y
```

We use a function conveniently called $proc$ and use an infix function conveniently called $(\prec)$. Slightly less nice is that we now have to use the $Applicative$ interface to combine values resulting from arrow computations: we have to write $(+) <\$> x <*> y$ instead of $x + y$. Note that $proc$ is a *function*, which does all the plumbing to rewrite the syntax to a point-free expression, which is normally done in a compiler pass.

The difference between **do** notation and arrow notation is that in arrow notation, one cannot observe intermediate values to decide what to do next. For example, we *cannot* do the following:

```
ifArrow :: a Int x → a Int x → a Int x
ifArrow t f = proc z → do
  case z of
    0 → t ≺ z
    _ → f ≺ z
```

Allowing this kind of behavior would make it impossible to translate arrow notation to arrow expressions, because this is exactly the power that monads have but that arrows lack (Lindley et al. 2011). To mimic this restriction in our embedded arrow notation, our function $(\prec)$ has the following type:

```
(≺) :: Arrow a ⇒ a x y → Cage s x →
       ArrowSyntax a s (Cage s y)
```

The type $ArrowSyntax$ is the monad which we use to define our embedded arrow notation. The input and output of the arrow computations are enclosed in $Cage$s, a type which disallows observation of the value of type x it "contains".

3.2 Implementing Embedded Arrow Syntax

The implementation of a $Cage$ is as follows:

```
newtype Cage s x = Cage { open :: KeyMap s → x }
  deriving (Functor, Applicative)
```

Informally, a $Cage$ "contains" a value of type x, but in reality it does not contain a value of type x at all: it is a function from a $KeyMap$ to a value of type x. Hence we can be sure that arrow computations returning a $Cage$ do not allow pattern-matching on the result, because the result is simply not available. The expression

$(+) \mathbin{<\$>} x \mathbin{<*>} y$ we saw earlier in the function $addA$, uses the $Applicative$ instance of $Cage$s.

In our construction, we use Keys as names, and $KeyMap$s as *environments*, i.e. mappings from names to values. Each result of an arrow via $(\prec)$ has its own name. A $Cage$ stands for an expression, i.e. a function from environment to value, which may lookup names in the environment. As seen before, the Key monad in conjunction with $KeyMap$s allows us to model *heterogeneous* environments which can be extended *without changing* the *type* of the environment, which is exactly the extra power we need to define this translation.

By using $(\prec)$ and the monad interface, we can construct the syntax for the arrow computation that we are expressing. Afterwards, we use the following function to convert the syntax to an arrow:

```
proc :: Arrow a ⇒
  (∀ s. Cage s x → ArrowSyntax a s (Cage s y))
   → a x y
```

Internally, the $ArrowSyntax$ monad builds an *environment arrow*: an arrow from environment to environment, i.e. an arrow of type $a\ (KeyMap\ s)\ (KeyMap\ s)$. The $ArrowSyntax$ monad creates names for values in these environments using $KeyM$.

```
newtype ArrowSyntax a s x =
  AS (WriterT (EnvArrow a s) (KeyM s) x)
    deriving (Functor, Applicative, Monad)
newtype EnvArrow a s =
  EnvArrow (a (KeyMap s) (KeyMap s))
```

The type declaration for $EnvArrow$ suggests that this approach would not have worked if we had used the ST monad instead of the Key monad; $KeyMap$s are used as inputs as well as outputs for general arrows, which needs $KeyMap$s to be tangible objects rather than hidden inside a monadic computation, as they are in the ST monad.

Like any arrow from a type x to the same type, $EnvArrow$s form a monoid as follows:

```
instance Arrow a ⇒ Monoid (EnvArrow a x) where
  mempty = EnvArrow (arr id)
  mappend (EnvArrow l) (EnvArrow r) =
    EnvArrow (l >>> r)
```

The definition of $ArrowSyntax$ uses the standard writer monad transformer, $WriterT$, which produces $mempty$ for $return$, and composes the built values from the left and right hand side of $>>=$ using $mappend$, giving us precisely what we need for building arrows.

To define the operations $proc$ and $(\prec)$, we first define some auxiliary functions for manipulating environments. We can easily convert a name (Key) to the expression ($Cage$) which consists of looking up that name in the environment:

```
toCage :: Key s a → Cage s a
toCage k = Cage (λenv → env ! k)
```

We can introduce an environment from a single value, when given a name (Key) for that value:

```
introEnv :: Arrow a ⇒ Key s x → a x (KeyMap s)
introEnv k = arr (λv → insert k v empty)
```

We also define an arrow to eliminate an environment, by interpreting an expression ($Cage$) using that environment:

```
elimEnv :: Arrow a ⇒ Cage s x → a (KeyMap s) x
elimEnv c = arr (open c)
```

Apart from functions to introduce and eliminate environments, we also need functions to extend an environment and to evaluate an expression while keeping the environment:

```
extendEnv :: Arrow a ⇒ Key s x →
                       a (x, KeyMap s) (KeyMap s)
extendEnv k = arr (uncurry (insert k))
withEnv :: Arrow a ⇒ Cage s x →
                     a (KeyMap s) (x, KeyMap s)
withEnv c = dup >>> first (elimEnv c)
  where dup = arr (λx → (x, x))
```

With these auxiliary arrows, we can define functions that convert back and forth between a regular arrow and an environment arrow. To implement $(\prec)$, we need to convert a regular arrow to an environment arrow, for which we need an expression for the input to the arrow, and a name for the output of the arrow:

```
toEnvArrow :: Arrow a ⇒
  Cage s x → Key s y →
  a x y → EnvArrow a s
toEnvArrow inC outK a = EnvArrow $
  withEnv inC >>> first a >>> extendEnv outK
```

We first produce the input to the argument arrow, by interpreting the input expression using the input environment. We then execute the argument arrow, and bind its output to the given name to obtain the output environment.

The $(\prec)$ operation gets the arrow and the input expression as an argument, creates a name for the output, and then passes these three to $toEnvArrow$:

```
(≺) :: Arrow a ⇒
  a x y →
  (Cage s x → ArrowSyntax a s (Cage s y))
a ≺ inC = AS $
  do outK ← lift newKey
     tell (toEnvArrow inC outK a)
     return (toCage outK)
```

In the other direction, to implement $proc$ we need to convert an environment arrow to a regular arrow, for which we instead need the name of the input and an expression for the output:

```
fromEnvArrow :: Arrow a ⇒
  Key s x → Cage s y →
  EnvArrow a s → a x y
fromEnvArrow inK outC (EnvArrow a) =
  introEnv inK >>> a >>> elimEnv outC
```

We first bind the input to the given name to obtain the input environment. We then transform this environment to the output environment by running the arrow from environment to environment. Finally, we interpret the output expression in the output environment to obtain the output.

The $proc$ operation creates a name for the input and passes it to the function as an expression to obtain the output expression and the environment arrow. We then convert the obtained arrow from environment to environment using $fromEnvArrow$:

```
proc :: Arrow a ⇒
  (∀ s. Cage s x → ArrowSyntax a s (Cage s y)) →
  a x y
proc f = runKeyM $
  do inK ← newKey
     let AS m = f (toCage inK)
     (outC, a) ← runWriterT m
     return (fromEnvArrow inK outC a)
```

3.3 Discussion

Altenkirch, Chapman and Uustalu (Altenkirch et al. 2010) show a related construction: in category theory arrows are a special case of *relative monads*, which are themselves a generalization of monads. In Haskell, a relative monad is an instance of the following type class:

```
class RelativeMonad m v where
  rreturn :: v x → m x
  (>>=)   :: m x → (v x → m y) → m y
```

The only difference with regular monads is that the values resulting from computations must be wrapped in a type constructor v, instead of being "bare". The relative monad laws are also the same as the regular (Haskell) monad laws. The construction of Altenkirch et al. which shows that arrows are an instance of relative monads is not a relative monad in Haskell, only in category theory. In particular their construction uses the Yoneda embedding, which does not allow us freely use bound values, instead it requires us to manually lift values into scope, in the same fashion as directly using de Bruijn indices.

Because all the operations in *ArrowSyntax* (namely $(\prec)$) return a *Cage*, it might be more informative to see it as a relative monad, i.e.:

```
data ArrowRM a s x = ArrowRM
  (ArrowSyntax a s (Cage s x))
instance RelativeMonad (ArrowRM a s) (Cage s)

(≺) :: a x y → Cage s x → ArrowRM a s y
proc :: (∀ a. Cage s x → ArrowRM a s y) → a x y
```

In this formulation, it is clear that the user cannot decide what to do next based on the outcome of a computation: all we can get from a computation is *Cage*s. The monadic interface does not add extra power: while we cannot decide what to do next based on the output of a computation of type *ArrowSyntax s (Cage s x)*, we can, for example, decide what to next based on the outcome of a computation of type *ArrowSyntax s Int*. This does not give our embedded arrow notation more power than regular arrow notation or the relative monad interface: the value of the integer cannot depend on the result of an arrow computation and hence must be the result of a pure computation. This is essentially the same trick as described in Svenningsson and Svensson(Svenningsson and Svensson 2013).

As an aside, more generally, this trick can be used to give a monadic interface for *any* relative monad:

```
data RelativeMSyntax rm v a where
  Pure :: a → RelativeMSyntax rm v a
  Unpure :: rm a → (v a → RelativeMSyntax rm v b)
            → RelativeMSyntax rm v b

instance Monad (RelativeMSyntax rm v) where
  return = Pure
  (Pure x) >>= f = f x
  (Unpure m f) >>= g = Unpure m (λx → f x >>= g)

fromRelativeM :: RelativeMonad rm v ⇒
  rm a → RelativeMSyntax rm v (v a)
fromRelativeM m = Unpure m return

toRelativeM :: RelativeMonad rm v ⇒
  RelativeMSyntax rm v (v a) → rm a
toRelativeM (Pure x)      = rreturn x
toRelativeM (Unpure m f) = m >>=(toRelativeM ∘ f)
```

The insight is that because a computation must eventually return a value of $v\ a$ to convert a relative monad computation via *toRelativeM*, any pure value that is used, can eventually be removed via the monad law $return\ x \gg\!= f \equiv f\ x$. Our embedded arrow construction can be seen as a relative monad, where we apply this trick to obtain a monadic interface.

Our construction hence suggests that arrows are also a special case of relative monad in Haskell with the key monad, but a formal proof (using the Key monad laws from Figure 2) is outside the scope of this paper. In the code online, we also show that this construction can be extended to use *relative monadfix* (with function $rmfix :: (v\ a \to m\ a) \to m\ a$) to construct arrows using *ArrowLoop*, but we cannot use recursive monad notation in this case, because the above trick does not extend to Monadfix.

The *Arrow Calculus*(Lindley et al. 2010) describes a translation of a form of arrow syntax (not embedded in Haskell) to arrows which is very similar to the construction presented here. Their calculus has five laws, three of which can be considered to be relative monad laws, which they use to prove the equational correspondence between their calculus and regular arrows. Due to the similarity, their paper should provide a good starting point for anyone trying to prove the same for this construction.

4. Representations of Variables in Syntax

What else can we do with the Key monad? The Key monad allows us to associate types with "names" (*Key*s), and to see that if two names are the same, then their associated types are also the same. Use cases for this especially pop up when dealing with representations of syntax with binders, as we will show next.

4.1 Typed Names

A straightforward way to represent the syntax of a programming language is to simply use strings or integers as names. For example, the untyped lambda calculus can be represented as follows:

```
type Name = Int
data Exp  = Var Name
          | Lam Name Exp
          | App Exp Exp
```

If we want to represent a *typed* representation of the lambda calculus, then this approach does not work anymore. Consider the following GADT:

```
data TExp a where
  Var :: Name → TExp a
  Lam :: Name → TExp b → TExp (a → b)
  App :: TExp (a → b) → TExp a → TExp b
```

We cannot do much with this datatype. If we, for example, want to write an interpreter, then there is no way to represent the environment: we need to map names to values of different types, but there is no type-safe way to do so.

We could add an extra argument to *Var* and *Lam* containing the type-representation of the type of the variable, obtained using *Typeable*. With the Key monad, we extend this simple naming approach to typed representations without adding *Typeable* constraints. Consider the following data type:

```
data KExp s a where
  KVar :: Key s a → KExp s a
  KLam :: Key s a → KExp s b → KExp s (a → b)
  KApp :: KExp s (a → b) → KExp s a → KExp s b
```

Because the names are now represented as keys, we can represent an environment as a *KeyMap*. We can even offer a Higher Order Abstract Syntax (HOAS) (Pfenning and Elliott 1988) interface for constructing such terms by threading the key monad computation, which guarantees that all terms constructed with this interface are well-scoped:

```
class Hoas f where
  lam :: (f a → f b) → f (a → b)
  app :: f (a → b)   → (f a → f b)
newtype HoasKey s a =
  HK {getExp :: KeyM s (KExp s a)}
instance Hoas (HoasKey s) where
  lam f = HK $
    do k ← newKey
       b ← getExp $ f $ HK $ pure $ KVar k
       return (KLam k b)
  app f x = HK $ KApp <$> getExp f <*> getExp x
```

For instance, the lambda term $(\lambda x\ y \to x)$ can now be constructed with: $lam\ (\lambda x \to lam\ (\lambda y \to x))$.

Note that we only need the Key monad to *create* keys. Once we have created the necessary keys, we can stay fully within normal, non-monadic Haskell. The above example can also be done with the ST monad (using $STRef$s as names), but we would have to perform every computation that would do something with these names (for example an interpreter) inside the ST monad.

4.2 Translating Well-scoped Representations

The datatype $KExp$ does not ensure that any value of type $KExp$ is well-scoped. There are two well-known approaches to constructing data types for syntax which ensure that every value is well-scoped. The first is parametric Higher Order Abstract Syntax (HOAS) (Chlipala 2008; Oliveira and Löh 2013; Oliveira and Cook 2012), and the second is using typed de Bruijn indices (Bird and Paterson 1999).

Previous work (Atkey 2009; Atkey et al. 2009) has shown how to translate parametric HOAS terms to terms with typed de Bruijn indices by relying on parametricty. Interestingly, it seems there is no type-safe way to do this in Haskell without adding $Typeable$ constraints to the $Phoas$ datatype or using $unsafeCoerce$. The Key monad does allow us to cross this chasm.

In parametric HOAS, typed lambda terms are represented by the following data type:

```
data Phoas v a where
  PVar :: v a → Phoas v a
  PLam :: (v a → Phoas v b) → Phoas v (a → b)
  PApp :: Phoas v (a → b) → Phoas v a → Phoas v b
```

The reading of the type parameter v is the type of *variables*. For example, the lambda term $(\lambda x\ y \to x)$ can be constructed as follows:

```
phoasExample :: Phoas v (x → y → x)
phoasExample = PLam (λx → PLam (λy → x))
```

An attractive property of parametric HOAS is that we use Haskell binding to construct syntax, and that terms of type $(\forall\ v.\ Phoas\ v\ a)$ are always well-scoped (Chlipala 2008).

The second way to ensure well-scopedness is to use typed de Bruijn indices. Here, we present our own modern variant of this technique using Data Kinds and GADTs, but the idea is essentially the same as presented by Bird and Paterson (Bird and Paterson 1999). Our representation of typed de Bruijn indices is an index in a heterogeneous list (Figure 4). A typed de Bruijn index of type $Index\ l\ a$ is an index for a variable of type a in an environment where the types of the variables are represented by the type level list l. We can use these indices to represent lambda terms as follows:

```
data Bruijn l a where
  BVar :: Index l a → Bruijn l a
  BLam :: Bruijn (a : l) b → Bruijn l (a → b)
  BApp :: Bruijn l (a → b) → Bruijn l a → Bruijn l b
```

```
data Index l a where
  Head :: Index (h : t) h
  Tail :: Index t x → Index (h : t) x
data TList l f where
  TNil :: TList [] f
  (:::) :: f h → TList t f → TList (h : t) f
index :: TList l f → Index l a → f a
index (h ::: _) Head = h
index (_ ::: t) (Tail i) = index t i
instance FFunctor (TList l) where
  ffmap f TNil = TNil
  ffmap f (h ::: t) = f h ::: ffmap f t
```

Figure 4. Heterogeneous list and indexes in them.

A closed term of type a has type $Bruijn\ []\ a$.

The types $(\forall\ v.\ Phoas\ v\ a)$ and $(Bruijn\ []\ a)$ both represent well-scoped typed lambda terms (and $\bot$), and translating from the latter to the former is straightforward. However, there currently seems to be no way to translate the former to the latter (without using the Key monad). In other words there seems to be no function of type:

```
phoasToBruijn :: (∀ v. Phoas v a) → Bruijn [] a
```

This seems to be impossible not only in Haskell without extensions, but in dependently typed languages without extensions as well. For example, when using $Phoas$ in *Coq* to prove properties about programming languages, a small extension to the logic in the form of a special well-scopedness axiom for the $Phoas$ data type is needed to translate Phoas to de Bruijn indices(Chlipala 2008).

The well-scopedness of a $Bruijn$ value follows from the fact that the value is well-typed. With $Phoas$, the well-scopedness relies on the meta-level (i.e. not formalized through types) argument that no ill-scoped values can be created using the $Phoas$ interface. The internal (i.e. formalized through types) well-scopedness of $Bruijn$, allows interpretations of syntax which seem to not be possible if we are using terms constructed with $Phoas$. As an example of this, consider translating lambda terms to *Cartesian closed category* combinators (the categorical version of the lambda calculus), which is useful for translating lambda terms to hardware (Elliott 2013). This can be done if the lambda terms are given as $Bruijn$ values; we elide the translation for space reasons but it can be found in the online repository for this paper. Without the Key monad, there seem to be no way to do the same for terms constructed with the $Phoas$ terms, but with the Key monad we can for example first translate to de Bruijn indices and then to Cartesian closed categories.

Our implementation of $phoasToBruijn$ works by first translating $Phoas$ to the $KExp$ from the previous subsection, and then translating that to typed de Bruijn indices. The first step in this translation is straightforwardly defined using the $Hoas$ interface from the previous subsection:

```
phoasToKey :: (∀ v. Phoas v a) → KeyM s (KExp s a)
phoasToKey v = getExp (go v) where
  go :: Phoas (HoasKey s) a → HoasKey s a
  go (PVar v)   = v
  go (PLam f)   = lam (go ∘ f)
  go (PApp f x) = app (go f) (go x)
```

We will now show how we can create a function of type:

```
keyToBruijn :: KExp s a → Bruijn [] a
```

Using this function, we can then implement $phoasToBruijn$ as follows:

```
phoasToBruijn :: (∀ v. Phoas v x) → Bruijn [] x
phoasToBruijn p =
  runKeyM (keyToBruijn <$> phoasToKey p)
```

To implement the $keyToBruijn$ function, we need a variant of the Box we saw in Section 2.1:

```
data FBox s f where
  FLock :: Key s a → f a → FBox s f

funlock :: Key s a → FBox s f → Maybe (f a)
funlock k (FLock k' x) =
  case testEquality k k' of
    Just Refl → Just x
    Nothing   → Nothing
```

The difference with Box is that we now store values of type $f\ a$ instead of values of type a in the box. We provide a variant of *fmap* for this container:

```
class FFunctor p where
  ffmap :: (∀ x. f x → g x) → p f → p g

instance FFunctor (FBox s) where
  ffmap f (FLock k x) = FLock k (f x)
```

We also need a variant of the $KeyMap$, where we store $FBox$es instead of regular boxes:

```
newtype FKeyMap s f = FKM [FBox s f]
empty :: FKeyMap s f
insert :: Key s a → f a → FKeyMap s f → FKeyMap s f
lookup :: Key s a → FKeyMap s f → Maybe (f a)
instance FFunctor (FKeyMap s)
```

To translate to de Bruijn indices, we store the current "environment" as an $FKeyMap$ mapping each Key to an $Index$ in the current environment. When we enter a lambda-body, we need to extend the environment: we add a mapping of the new variable to the de Bruijn index $Head$, and add one lookup step to each other de Bruijn index currently in the $FKeyMap$. This is be done as follows:

```
extend :: Key s h → FKeyMap s (Index t) →
          FKeyMap s (Index (h : t))
extend k m = insert k Head (ffmap Tail m)
```

The type of $extend$ suggests that we could not have used the ST monad instead of the Key monad, because the internal state of the computation (represented by the $FKeyMap$) changes *type* after $extend$. To support this in the ST monad, we would need the equivalent of *ffmap*, which would map a function over all existing $STRef$s.

With this machinery in place, we can translate $KExp$ to $Bruijn$ as follows:

```
keyToBruijn :: KExp s a → Bruijn [] a
keyToBruijn = go empty where
  go :: FKeyMap s (Index l) → KExp s x → Bruijn l x
  go e (KVar v)   = BVar (e ! v)
  go e (KLam k b) = BLam (go (extend k e) b)
  go e (KApp f x) = BApp (go e f) (go e x)
```

Note that $keyToBruijn$ fails if the input $KExp$ is ill-scoped. This cannot happen when $keyToBruijn$ is called from $phoasToBruijn$ because $phoasToKey$ will always give well-scoped values of type $KExp$. This relies on the meta-level argument that values of type $PHoas$ are always well-scoped. We stress that hence the key monad extension *cannot* serve as a replacement of well-scopedness axiom used in a dependently typed setting.

5. Safety of the Key Monad

In this section, we state more precisely what we mean by safety, and informally argue for the safety of the Key monad.

5.1 Type Safety

The first safety property that we conjecture the Key monad has is *type safety*: $testEquality$ will never allow us to prove that $a :\sim: b$ if a and b are *distinct* types. Informally, the justification for this is that a key value k of type $Key\ s\ a$ together with the type s, which we call the *scope type variable*, *uniquely determines* the associated type a of the key. Hence, when two key values and scope type variables are the same[3], their associated types *must be the same* as well.

The argument why the scope type variable s and the key value k together uniquely determine type a goes as follows:

1. Each execution of a Key monad computation has a scope type variable s that is distinct from the scope type variables of all other Key monad computations. This is ensured by the type of $runKeyM$, namely $(\forall\ s.\ KeyM\ s\ a) \to a$, which states that the type s cannot be unified with any other type.
2. Each $newKey$ operation in such a Key monad computation gives a value that is unique within the scope determined by s, i.e. distinct from other keys created in the same computation.
3. Each key only has *a single type* associated with it. This is ensured by the type of $newKey$, which only allows us to construct a key with a single type, i.e. not a key of type $\forall\ a.\ Key\ s\ a$. The type of a hypothetical function $newPolymorphicKey :: KeyM\ s\ (\forall\ a.\ Key\ s\ a)$ does not unify with the type of $newKey$.

5.2 Referential Transparency

The second safety property that we are concerned with is *referential transparency*. More precisely, in an otherwise pure language with the Key monad extension, does the following still hold?

$$(\mathbf{let}\ x = e\ \mathbf{in}\ f\ x\ x) \equiv f\ e\ e$$

In other words, referential transparency means that an expression always evaluates to the same result in any context. Our implementation of the key monad only relies on *unsafeCoerce*; it does not use *unsafePerformIO*, nor does it use $unsafeCoerce$ to convert an $IO\ a$ action to a pure value (if we assume type safety) and hence referential transparency cannot be broken by this implementation. Since the ST monad can be implemented using the Key monad, the same can be said for the ST monad. However, more efficient implementations of the ST monad use *global* pointers respectively, which do rely on features that might potentially break referential transparency.

5.3 Abstraction Safety

Abstraction safety is the property that we cannot break the abstraction barriers which are introduced through existential types. For example, consider the following existential type:

```
data AbsBool where
  AbsBool :: Eq a ⇒ a → a → (a → b → b → b)
             → AbsBool
```

Let us consider two different uses of this type:

```
boolBool =
  AbsBool True False (λc t f → if c then t else f)
boolInt =
  AbsBool 0 1 (λc t f → if c ≡ 0 then t else f)
```

[3] Even though users cannot compare keys explicitly, implementations of the Key monad internally represent keys by some underlying value that can be compared for equality.

If a language is abstraction safe, then it is impossible to observe any difference between $boolBool$ and $boolInt$. This property is formalized by *parametricity* (which also gives "free" theorems (Wadler 1989)). A typical example of a primitive which is not abstraction safe (but is type-safe) is a primitive that allows us to check the equality of any two types:

```
badTest :: a → b → Maybe (a :~: b)
```

The primitive $testEquality$ is similar to the $badTest$ primitive above, and indeed our operations on Box do allow us to "break the abstraction barrier": if $unlock$ succeeds, we have learned which type is hidden in the Box. However, finding out which type is hidden by an existential type can not only be done with the Key monad, but also by the established Generalized Algebraic Data types extension of Haskell. For example, suppose we have the following type:

```
data IsType a where
  IsBool :: IsType Bool
  IsInt  :: IsType Int
  IsChar :: IsType Char
```

We can then straightforwardly implement a variant of $testEquality$:

```
testEquality :: IsType a → IsType b → Maybe (a :~: b)
testEquality IsBool  IsBool  = Just Refl
testEquality IsInt   IsInt   = Just Refl
testEquality IsChar  IsChar  = Just Refl
testEquality _       _       = Nothing
```

There are, however, formulations of parametricity which state more precisely exactly which abstraction barrier cannot be crossed (Vytiniotis and Weirich 2007; Bernardy et al. 2012). In these formulations, $boolBool$ and $boolInt$ are indistinguishable. We can think of $runKeyM$ as an operation which dreams up a specific Key GADT for the given computation, for example:

```
data Key A a where
  Key0 :: Key A Int
  Key1 :: Key A Bool
  ...
```

Here A is a globally unique type associated with the computation. This interpretation is a little tricky: since a Key computation might create an infinite number of keys, this hypothetical datatype might have an infinite number of constructors. Alternatively, we can interpret keys as GADTs that index into a type-level list or type-level tree, as we do in Section 6. We conjecture that there is a variant of parametricity for Haskell extended with the Key monad in which, in analogy with parametricity for GADTs, $boolBool$ and $boolInt$ above are considered to be indistinguishable.

5.4 Normalization

A fourth desirable property of a type system extension is preservation of normalization, i.e., the property that ensures well-typed terms always have a normal form. Although standard typed λ-calculi (such as System F) are normalizing, Haskell is not, as we can write non-terminating programs. Even without term-level recursion, we can create programs that do not terminate by using type-level recursion. However, if we disallow contravariant recursion at the type level (i.e. type-level recursive occurrences that occur to the left of a function arrow), then all Haskell programs without term-level recursion do terminate.

It turns out that extending a normalizing language with the Key monad breaks normalization. We show this by implementing a general fixpoint combinator *fix* which uses neither contravariant recursion at the type level nor term-level recursion.

Figure 5 presents the implementation of *fix*. First, we introduce a datatype $D\ s\ a$ for domains representing models of the untyped lambda calculus. (We are going to encode the standard fixpoint combinator $\lambda f \to (\lambda x \to f\ (x\ x))\ (\lambda x \to f\ (x\ x))$ in this domain.) An element of $D\ s\ a$ is either a function over $D\ s\ a$ or a value of type a. Normally, we would use contravariant recursion for the argument of Fun, but we are not allowed to, so we mask it by using a $Box\ s$ instead. As a result, $D\ s\ a$ is not contravariantly recursive, and neither are any of its instances.

```
data D s a
  = Fun (Box s → D s a)
  | Val a

lam :: Key s (D s a) → (D s a → D s a) → D s a
lam k f = Fun (f . fromJust . unlock k)

app :: Key s (D s a) → D s a → D s a → D s a
app k (Fun f) x = f (Lock k x)

fix :: (a → a) → a
fix f = runKeyM $
  do k ← newKey
     let f'   = lam k (Val . f . unVal)
         xfxx = lam k (λx → app k f' (app k x x))
         fixf = app k xfxx xfxx
     return (unVal fixf)
where unVal (Val x) = x
```

Figure 5. Implementing a general fixpoint combinator without term-level recursion nor type-level contravariant recursion

Second, we introduce two helper functions: lam, which takes a function over the domain, and injects it as an element into the domain, and app, which takes two elements of the domain and applies the first argument to the second argument. Both need an extra argument of type $Key\ s\ (D\ s\ a)$ to lock/unlock the forbidden recursive argument.

Third, the fixpoint combinator takes a Haskell function f, wraps it onto the domain $D\ s\ a$ resulting in a function f', and then uses lam and app to construct a fixpoint combinator from the untyped lambda calculus. Lastly, we need to convert the result from the domain $D\ s\ a$ back into Haskell-land using $unVal$.

What this shows is that (1) adding the Key monad to a normalizing language may make it non-normalizing, (2) the Key monad is a genuine extension of Haskell without term-level recursion and type-level contravariant recursion. Incidentally, this is also the case for the ST monad. In a stratified type system with universe levels, such as Agda or Coq, it should be possible to omit this problem by making keys of a higher level than their associated types. In Haskell, this would defeat the "unconstrained" part of the title of the paper; then we could just as well have used $Typeable$.

6. Implementing the Key Monad

Is the Key monad expressible in Haskell directly, without using $unsafeCoerce$? Can we employ more recent advancements such as GADTs to "prove" to the type system that the Key monad is safe? In this section, we explore how far we can get (and fail).

The question of whether or not the Key monad is implementable in Haskell (with extensions) is related to the question of whether or not the ST monad is implementable in Haskell (with extensions): a negative answer to the latter implies a negative answer to the former. The latter question, about ST, was (as far as we know) first publicly asked by the second author on the Haskell mailing list in 2001 (Claessen 2001), accompanied by a proposal of an early version of the Key monad, then called the "Object monad". Since then, the question has regularly popped up on online discussion forums (e.g.

(alexeyr 2015)). The question has never been answered positively, which we take as a strong indication that the Key monad is also not implementable in Haskell (with extensions).

6.1 Implementation Using $unsafeCoerce$

To get a feel for possible implementations of the Key monad, let us first consider a straightforward implementation, using *unsafeCoerce*, in which we give each key a unique name. One could implement generating unique names using a state monad, but the $(purity)$ key monad law $(m \gg n \equiv n)$ would then not hold. Instead, we implement the Key monad using a splittable name supply (McBride and McKinna 2004), with the following interface:

```
newNameSupply :: NameSupply
split         :: NameSupply →
                 (NameSupply, NameSupply)
supplyName    :: NameSupply → Name
```

One implementation of the $NameSupply$ uses paths in a binary tree:

```
data TreePath = Start | Left TreePath | Right TreePath
```

When reading left-to-right, these paths are given in reverse order from the root: the path $Left\ (Right\ Start)$ is a path to the left child of the right child of the root. A name is then a path to leaf in a tree, and a name supply is a path to a subtree. To split a $NameSupply$, we convert a path to a node into a path to the two children of that node:

```
type NameSupply = TreePath
type Name       = TreePath
newNameSupply = Start
split s       = (Left s, Right s)
supplyName    = id
```

Using such name supplies, the implementation of the Key monad is as follows:

```
data KeyM s a =
  KeyM {getKeyM :: NameSupply → a}
data Key s a = Key Name

newKey :: KeyM s (Key s a)
newKey = KeyM $ λs → Key (supplyName s)

instance Monad (KeyM s) where
  return x = KeyM $ \_ → x
  m >>= f  = KeyM $ λs →
    let (sl, sr) = split s
    in getKeyM (f (getKeyM m sl)) sr

runKeyM :: (∀ s. KeyM s a) → a
runKeyM (KeyM f) = f newNameSupply

testEquality (Key l) (Key r)
  | l ≡ r     = Just (unsafeCoerce Refl)
  | otherwise = Nothing
```

A $KeyM$ computation consisting of $\gg\!=$,$return$ and $newKey$ can also be seen as a binary tree where binds are nodes, $newKey$s are leaves and $return$s are empty subtrees. The $Name$ associated with each key is the path to the $newKey$ that created it, in the tree that corresponds to the $KeyM$ computation. For example, the Key resulting from the $newKey$ in the expression:

```
runKeyM $ (m >> newKey) >>= f
```

will get the name $Right\ (Left\ Start)$.

Note that the Key monad laws from Figure 2 only hold for this implementation *up to observation*. If we have access to the definition of Keys, we can discriminate between, for example, $m \gg n$ and n. However, in the interface Key is an abstract type, and thus users can be blissfully ignorant of this.

A downside of this implementation is that $testEquality$ is linear in the length of the tree paths. A more efficient implementation of the Key monad uses $Integer$s to represent keys and deals out unique names by unsafely reading and updating a mutable variable which is unsafely created in $runKey$. A full implementation of this version of the Key monad can be found in the code online.

6.2 The Key Indexed Monad

Can we formalize through types the invariant that when two keys are the same their types must also be the same? It turns out we can, but this adds more types to the interface, leading to a loss of power of the construction.

The crucial insight is that is needed for this implementation, is that it *is* possible to implement to compare two indices in a heterogeneous list (Fig 4), and if they are equal, then produce a proof that the types are equal, as follows:

```
testEquality :: Index l a → Index l b → Maybe (a :~: b)
testEquality Head     Head     = Just Refl
testEquality (Tail l) (Tail r) = testEquality l r
testEquality _        _        = Nothing
```

We can employ the same insight to construct $testEquality$ function for other data types. Instead of indexes in a heterogeneous list, we add types to the paths in a tree to obtain *paths in a heterogenous tree*. For this datatype we need to be able to construct type-level trees, for which we use the following data type as a data-kind:

```
data Tree a = Empty | Single a | Tree a :++: Tree a
```

With this datatype, we can construct types of kind $Tree * $ such as:

```
Single Int :++: (Single Bool :++: Single String)
```

We can now adapt the datatype $TreePath$ to provide paths in type-level trees instead of value-level trees, in a similar fashion to how $Index$ is an index in a type-level list instead of a value-level list:

```
data TTreePath p w where
  Start :: TTreePath w w
  Left  :: TTreePath (l :++: r) w → TTreePath l w
  Right :: TTreePath (l :++: r) w → TTreePath r w
```

We can now construct a $testEquality$-like function of the following type:

```
samePath :: TTreePath p w → TTreePath p' w
         → Maybe (p :~: p')
```

The implementation of this function is a bit more involved than for $Index$, but is unsurprising:

```
samePath Start      Start      = Just Refl
samePath (Left l)   (Left r)   = weakL <$> samePath l r
samePath (Right l)  (Right r)  = weakR <$> samePath l r
samePath _          _          = Nothing where
  weakL :: ((l :++: r) :~: (l' :++: r')) → l :~: l'
  weakL x = case x of Refl → Refl

  weakR :: ((l :++: r) :~: (l' :++: r')) → r :~: r'
  weakR x = case x of Refl → Refl
```

We can use this function to implement a function that produces a proof that if two paths to a leaf are the same, then their associated types are the same:

```
sameLeaf :: TTreePath (Single p) w →
            TTreePath (Single p') w →
            Maybe (p :~: p')
```

$$sameLeaf\ l\ r = weakenLeaf <\$> samePath\ l\ r\ \textbf{where}$$
$$weakenLeaf :: (Single\ p :\sim: Single\ p') \to p :\sim: p'$$
$$weakenLeaf\ x = \textbf{case}\ x\ \textbf{of}\ Refl \to Refl$$

Now that we have encoded the invariant that when two key values are the same then their associated types must also the same, we can use this to implement a *typed* name supply with the following interface:

$$\textbf{type}\ TNameSupply\ l\ s = TTreePath\ l\ s$$
$$\textbf{type}\ TName\ s\ a = TTreePath\ (Single\ a)\ s$$
$$newTNameSupply :: TNameSupply\ s\ s$$
$$tsplit :: TNameSupply\ (l :+: r)\ s \to (TNameSupply\ l\ s, TNameSupply\ r\ s)$$
$$supplyTName :: TNameSupply\ (Single\ a)\ s \to TName\ s\ a$$
$$sameName :: TName\ s\ a \to TName\ s\ b \to Maybe\ (a :\sim: b)$$

A typed name supply of type $TNameSupply\ \ l\ \ s$ gives unique names for the types in the subtree l which can be tested for equality, using $sameName$, with all names which are created in the context s. The implementations of the name supply functions are completely analogous to their untyped counterparts.

By using the typed name supply instead of the regular name supply and altering the types in the interface to reflect this change, we obtain an implementation of what we call the *indexed* Key monad, with the following interface:

$$newKeyIM :: KeyIM\ s\ (Single\ a)\ (Key\ s\ a)$$
$$rreturn :: a \to KeyIM\ Empty\ s\ a$$
$$(>\!\!>\!\!=) :: KeyIM\ s\ l\ a \to (a \to KeyIM\ s\ r\ b) \to KeyIm\ s\ (l :+: r)\ b$$
$$runKeyIM :: (\forall\ s.\ KeyIM\ s\ l\ a) \to a$$
$$testEquality :: Key\ s\ a \to Key\ s\ b \to Maybe\ (a :\sim: b)$$

The implementation of this interface is completely analogous to the implementation of the Key monad in the previous subsection. The only difference is that $testEquality$ now uses $sameName$, omitting the need for $unsafeCoerce$. This interface is an instance of the *parametric effect monad* type class(Orchard and Petricek 2014).

Note that in the implementation of $runKeyIm$ the universally quantified type variable s gets unified with l in order to use $newTNameSupply$. This "closes the context", stating that the context is precisely the types which are created in the computation. In contrast, in $runKeyM$ the type variable was not given an interpretation.

While we have succeeded in avoiding $unsafeCoerce$, this construction is *less powerful* than the regular Key monad because the types of the keys which are going to be created must now be *statically known*. All example use cases of the Key monad in this paper rely on the fact that the type of the keys which are going to be created do not have to be statically known. For example, it is not possible to implement a translation from parametric HOAS to de Bruijn indices with $KeyIM$, because the type of the keys which would have to be created is precisely the information that a parametric HOAS representation lacks.

6.3 Attempting to Recover the Key Monad

Can we formalize the invariant through types and provide the regular Key monad interface? We believe not.

An obvious attempt at this is hiding the extra type of $KeyIM$:

$$\textbf{data}\ KeyM\ s\ a\ \textbf{where}$$
$$KeyM :: KeyIM\ s\ p\ a \to KeyM\ s\ a$$

We denote this type by $\exists\ p.\ KeyIM\ s\ p\ a$ for presentational purposes, although it is not valid Haskell. While this allows us to provide type-safe implementations of $testEquality$, $fmap$, $newKey$ and $return$, things go awry for $join$ (or $>\!\!>\!\!=$) and $runKeyM$.

Here is the problem that arises for $runKeyM$. We get the type:

$$runKeyM :: (\forall\ s.\ \exists\ p.\ KeyIM\ s\ p\ a) \to a$$

But to use $runKeyIM$ the type should be:

$$runKeyM :: (\exists\ p.\ \forall\ s.\ KeyIM\ s\ p\ a) \to a$$

These types are *not* equivalent: the latter implies the former, but not the other way around. In the former, the type which is bound to p may depend on s, which cannot happen in the latter.

Let us take a look at what happens if we allow this coercion and p does depend on s, for example when we create a key of type $Key\ s\ (Key\ s\ Int)$. When the type s is now unified with the tree of types of the keys, it leads to a *cyclic* type:

$$s\ \sim\ (Key\ s\ Int) :+: t$$

In the previous section, we demonstrated that allowing such keys, where the type of the key mentions s, allows us to write *fix* without recursion. In the worst case, allowing such cyclic types may lead to type unsoundness.

For $join$ other problems arise. We need an implementation of type:

$$join :: (\exists\ p.\ KeyIM\ s\ p\ (\exists\ q.\ KeyIM\ s\ q\ a)) \to \exists\ r.\ KeyIM\ s\ r\ a$$

Expanding the definition of $KeyIM$, the type of the *argument* we have is:

$$\exists\ p.\ TNameSupply\ p\ s \to \exists\ q.\ TNameSupply\ q\ s \to a$$

However, to use the implementation of $join$ of $KeyIM$, we need the argument to be of type:

$$\exists\ p\ q.\ TNameSupply\ p\ s \to TNameSupply\ q\ s \to a$$

Again, these two types are *not* equivalent: the latter implies the former, but not the other way around. In general, q may depend on the value of $TNameSupply\ p\ s$. However, this is not the case in this implementation of $KeyM$ because the name supply and Key types are abstract and hence cannot influences the choice of type q. Unfortunately, invariants like these are very hard to express in the type system. Also, when a computation creates a potentially infinite number of keys, will also lead to an *infinite* type, which may again lead to type unsoundness.

7. Discussion on the ST Monad Proof

The ST monad was introduced in (Launchbury and Peyton Jones 1994) and contained some safety statements and also a high-level description of a proof. The proof sketch mentions the use of parametricity, which is a doubtful proof technique to use because it is not established that parametricity still holds for a language with the ST monad. A follow-up paper (Launchbury and Sabry 1997) mentions another problem with the first paper, in particular that implementations of the lazy ST monad may actually generate the wrong result in a setting that is more eager. The follow-up paper claims to fix those issues with a new semantics and a proof sketch. However, a bug in this safety proof was discovered, which led to a series of papers (Launchbury and Sabry 1997; Ariola and Sabry 1998) formalizing the treatment of different versions of encapsulating strict and lazy state threads in a functional language, culminating in (Moggi and Sabry 2001). This final paper gives different formulations of strict and lazy state threads, one of them corresponding more or less to lazy state threads in Haskell (although not using global pointers). The aim of this final paper is to establish *type safety* of state threads. However, the paper only provides a proof sketch of type safety for one of the formulations, and only claims

type safety (without a proof) for the other ones. With the exception of the original paper (Launchbury and Peyton Jones 1994), all these papers consider only *local state*, that is, each state thread has its own memory, in contrast to the actual implementation of the ST monad.

Even if type safety may now be considered to have been established by these papers, we are still left with referential transparency and abstraction safety. We are unaware of any work that establishes parametricity or referential transparency in the presence of the ST monad. Referential transparency is quite tricky for actual implementations of the ST monad since efficient implementations use global pointers. Abstraction safety is also very important because most people assume that parametricity in Haskell actually holds, without giving it a second thought that the ST monad may destroy it.

Now, we actually believe that the ST monad (and also the Key monad) is safe in all of these senses. But we have also realized that *there exist no actual proofs of these statements in the published literature*. We think that the Key monad, which is arguably simpler than the ST monad, could be a first step on the way to proving the ST monad safe.

8. Conclusions

In the ST monad, one of the invariants that must hold is that when two references are the same, then their types must also be the same. We presented the Key monad, which splits reasoning based on this invariant into a separate interface, and makes it available to the user. We showed that this new interface gives a form of dynamic typing without the need for *Typeable* constraints, which allows us to do things we could not do before: it allows us to implement heterogeneous maps, to implement the ST monad, to implement an embedded form of arrow syntax and to translate parametric HOAS to typed de Bruijn indices. The Key monad is simpler than the ST monad, since the former embodies just one aspect of the latter. A full proof of the safety of the ST monad remains elusive to this day. We feel that the Key monad might be the key to the proof of the ST monad.

Acknowledgements We thank Gershom Bazerman, Jonas Duregård and John Hughes for helpful comments and insightful discussions. This work was supported in part by The Sloan Foundation.

References

alexeyr (2015). SO question: Can a ST-like monad be executed purely? thread on reddit, https://www.reddit.com/r/haskell/.

Altenkirch, T., Chapman, J., and Uustalu, T. (2010). Monads need not be endofunctors. *Foundations of Software Science and Computational Structures*, pages 297–311.

Ariola, Z. M. and Sabry, A. (1998). Correctness of monadic state: An imperative call-by-need calculus. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 62–74, New York, NY, USA. ACM.

Atkey, R. (2009). *Syntax for Free: Representing Syntax with Binding Using Parametricity*, pages 35–49. Springer Berlin Heidelberg, Berlin, Heidelberg.

Atkey, R., Lindley, S., and Yallop, J. (2009). Unembedding domain-specific languages. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 37–48, New York, NY, USA. ACM.

Bernardy, J.-P., Jansson, P., and Paterson, R. (2012). Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152.

Bird, R. S. and Paterson, R. (1999). De bruijn notation as a nested datatype. *J. Funct. Program.*, 9(1):77–91.

Chlipala, A. (2008). Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA. ACM.

Claessen, K. (2001). A monad for type casting. Message to the Haskell mailing list, https://mail.haskell.org/pipermail/haskell/.

Elliott, C. (2013). From haskell to hardware via cartesian closed categories. http://conal.net/blog/posts/circuits-as-a-bicartesian-closed-category.

Hughes, J. (2000). Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111.

Launchbury, J. and Peyton Jones, S. L. (1994). Lazy functional state threads. In *PLDI*, pages 24–35.

Launchbury, J. and Sabry, A. (1997). Monadic state: Axiomatization and type safety. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 227–238, New York, NY, USA. ACM.

Lindley, S., Wadler, P., and Yallop, J. (2010). The arrow calculus. *J. Funct. Program.*, 20(1):51–69.

Lindley, S., Wadler, P., and Yallop, J. (2011). Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electron. Notes Theor. Comput. Sci.*, 229(5):97–117.

McBride, C. and McKinna, J. (2004). Functional Pearl: I Am Not a Number–i Am a Free Variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 1–9, New York, NY, USA. ACM.

Moggi, E. and Sabry, A. (2001). Monadic encapsulation of effects: A revised approach (extended version). *Journal of Functional Programming*, 11:2001.

Oliveira, B. C. and Cook, W. R. (2012). Functional programming with structured graphs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 77–88, New York, NY, USA. ACM.

Oliveira, B. C. d. S. and Löh, A. (2013). Abstract syntax graphs for domain specific languages. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, pages 87–96, New York, NY, USA. ACM.

Orchard, D. and Petricek, T. (2014). Embedding effect systems in haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 13–24, New York, NY, USA. ACM.

Paterson, R. (2001). A new notation for arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 229–240, New York, NY, USA. ACM.

Pfenning, F. and Elliott, C. (1988). Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 199–208, New York, NY, USA. ACM.

Svenningsson, J. and Svensson, B. J. (2013). Simple and compositional reification of monadic embedded languages. In *ICFP*, pages 299–304.

Vytiniotis, D. and Weirich, S. (2007). Type-safe cast does not harm.

Wadler, P. (1989). Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA. ACM.