Generalised Selection Monad

Johannes Hartmann[1], Tom Schrijvers[2], and Jeremy Gibbons[1]

[1] University of Oxford, Department of Computer Science, UK
firstname.lastname@cs.ox.ac.uk
[2] KULeuven, DepartmentofComputerScience, Belgium,
tom.schrijvers@kuleuven.be

Abstract. General setup and introduction words here

Keywords: Selection monad · Functional programming · Algorithm design · Performance Optimisation · Monads.

1 Introduction to the Selection Monad

This section introduces the selection monad, focusing on the `type J r a = (a -> r) -> a` for selection functions. The `pair` function is explored, showcasing its capability to compute new selection functions based on criteria from two existing functions. Illustrated with a practical example, the decision-making scenarios involving individuals navigating paths underscore the functionality of selection functions. An analysis of the inefficiency in the original `pair` function identifies redundant computational work. The primary contribution of the paper is then outlined: an illustration and proposal for an efficient solution to enhance `pair` function performance. This introductory overview sets the stage for a detailed exploration of the selection monad and subsequent discussions on optimizations.

1.1 Selection functions

Consider the tollowing already know type for selection functions:

```
type J r a = (a -> r) -> a
```

When given two selection functions, a `pair` function can be defined to compute a new selection function. This resultant function selects a pair based on the criteria established by the two given selection functions:

```
pair :: J r a -> J r b -> J r (a,b)
pair f g p = (a,b)
  where
      a = f (\x -> p (x, g (\y -> p (x,y))))
      b = g (\y -> p (a,y))
```

1.2 Example to illustrate the pair function

To gain a deeper understanding of the provided `pair` function, consider the following example. Picture two individuals walking on a path, one heading north and the other south. As they proceed, a collision is imminent. At this juncture, each individual must make a decision regarding their next move. This decision-making process can be modeled using selection functions. The decision they need to make is modeled as either going right or left:

```
data Decision = Left | Right
```

The respective selection functions decide given a predicate that tells them what decision is the correct one, select the correct one, and if there is no correct one, they default to walking right.

```
p1, p2 :: J Bool Decision
p1 p = if p Left then Left else Right
p2 p = if p Left then Left else Right
```

To apply the `pair` function, a predicate `pred` is needed that will judge two decisions and return `True` if a crash would be avoided and `False` otherwise.

```
pred :: (Decision, Decision) -> Bool
pred (Left,Right) = True
pred (Right,Left) = True
pred _            = False
```

With the `pair` function, the merging of the two selection functions into a new one that identifies an optimal decision can now be calculated.

```
pair p1 p2 pred
--> (Left,Right)
```

Examining how the `pair` function is defined reveals that the first element `a` of the pair is determined by applying the initial selection function `f` to a newly constructed property function. Intuitively, selection functions can be conceptualized as entities containing a collection of objects, waiting for a property function to assess their underlying elements. Once equipped with a property function, they can apply it to their elements and select an optimal one. Considering the types assigned to selection functions, it is evident that an initial selection function `f` remains in anticipation of a property function of type `(a -> r)` to determine an optimal `a`. The `pair` function is endowed with a property function `p :: ((a,b) -> r)`. Through the utilization of this property function, a property function for `f` can be derived by using the second selection function `g` to select a corresponding `b` and subsequently applying `p` to assess `(a,b)` pairs as follows: `(\x -> p (x, g (\y -> p (x,y))))`. Upon the determination of an optimal `a`, a corresponding `b` can then be computed as `g (\y -> p (a,y))`. In this case, the `pair` function can be conceptualized as a function that constructs all possible combinations of the elements within the provided selection function and subsequently identifies the overall optimal one. It might feel intuitive to consider the following modified `pair` function that seems to be more symmetric.

```
pair' :: J r a -> J r b -> J r (a,b)
pair' f g p = (a,b)
  where
      a = f (\x -> p (x, g (\y -> p (x,y))))
      b = g (\y -> p (f (\x -> p (x,y)), y))
```

However, applying this modified `pair'` to our previous example this results in a overall non optimal solution.

```
pair' p1 p2 pred
--> (Left,Left)
```

This illustrates how the original `pair` function keeps track of its first decision when determining its second element. It is noteworthy that, in the example example, achieving a satisfying outcome for both pedestrians is only possible when they consider the direction the other one is heading. The specific destination does not matter, as long as they are moving in different directions. Consequently, the original `pair` function can be conceived as a function that selects the optimal solution while retaining awareness of previous solutions, whereas our modified `pair'` does not. An issue with the original `pair` function might have been identified by the attentive reader. There is redundant computational work involved. Initially, all possible pairs are constructed to determine an optimal first element `a`, but the corresponding `b` that renders it an overall optimal solution is overlooked, resulting in only `a` being returned. Subsequently, the optimal `b` is recalculated based on the already determined optimal `a` when selecting the second element of the pair. The primary contribution of this paper will be to illustrate and propose a solution to this inefficiency.

1.3 Sequence

The generalization of the pair function to accommodate a sequence of selection functions is the initial focus of exploration. In the context of selection functions, a `sequence` operation is introduced, capable of combining a list of selection functions into a singular selection function that, in turn, selects a list of objects:

```
sequence :: [J r a] -> J r [a]
sequence [] p     = []
sequence (e:es) p = a : as
  where
      a  = e (\x -> p (x : sequence es (p . (x:))))
      as = sequence es (p . (a:))
```

Here, similar to the pair function, the sequence function extracts elements from the resulting list through the corresponding selection functions. This extraction is achieved by applying each function to a newly constructed property function that possesses the capability to foresee the future, thereby constructing an optimal future based on the currently examined element. However, a notable inefficiency persists, exacerbating the issue observed in the pair function. During the determination of the first element, the `sequence` function calculates an optimal remainder of the list, only to overlook it and redundantly perform the same calculation for subsequent elements. This inefficiency in `sequence` warrants further investigation for potential optimization in subsequent sections of this research paper.

1.4 Selection monad

The formation of a monad within the selection functions unfolds as follows:

```
(>>=) :: J r a -> (a -> J r b) -> J r b
(>>=) f g p = g (f (p . flip g p)) p

return :: a -> J r a
return x p = x
```

These definitions illustrate the monadic structure inherent in selection functions. The Haskell standard library already incorporates a built-in function for monads, referred to as `sequence'`, defined as:

```
sequence' :: [J r a] -> J r [a]
sequence' (ma:mas) = ma >>=
                     \x -> sequence' mas >>=
                     \xs -> return (x:xs)
```

Notably, in the case of the selection monad, this built-in `sequence'` function aligns with the earlier provided `sequence` implementation. This inherent consistency further solidifies the monadic nature of selection functions, underscoring their alignment with established Haskell conventions.

1.5 Illustration of Sequence in the Context of Selection Functions

To ilustrate the application of the sequence function within the domain of selection functions, consider a practical scenario: the task of cracking a secret password. In this hypothetical situation, a black box predicate `p` is provided that returns `True` if the correct password is entered and `False` otherwise. Additionally, knowledge is assumed that the password is six characters long:

```
p :: String -> Bool
p "secret" = True
p _        = False
```

Suppose access is available to a `maxWith` function, defined as:

```
maxWith :: Ord r => [a] -> (a -> r) -> a
maxWith [x] f                      = x
maxWith (x:y:xs) f | (f x) > (f y) = maxWith (x:xs) f
                   | otherwise     = maxWith (y:xs) f
```

With these resources, a selection function denoted as `selectChar` can be constructed, which, given a predicate that evaluates each character, selects a single character satisfying the specified predicate:

```
selectChar :: J Bool Char
selectChar = maxWith ['a'..'z']
```

It's worth noting that the use of maxWith is facilitated by the ordered nature of booleans in Haskell, where `True` is considered greater than `False`. Leveraging this selection function, the sequence function can be employed on a list comprising six identical copies of `selectChar` to successfully crack the secret password. Each instance of the selection function focuses on a specific character of the secret password:

```
sequence (replicate 6 selectChar) p
-> "secret"
```

This illustrative example not only showcases the practical application of the `sequence` function within the domain of selection functions but also emphasizes its utility in addressing real-world problems, such as scenarios involving password cracking. Notably, there is no need to explicitly specify a predicate for judging individual character; rather, this predicate is constructed within the monads bind definition, and its utilization is facilitated through the application of the `sequence` function. Additionally, attention should be drawn to the fact that this example involves redundant calculations. After determining the first character of the secret password, the system overlooks the prior computation of the entire password and initiates the calculation anew for subsequent characters. To address this specific inefficiency within the selection monad, concerning the pair and sequence functions, two new variations of the selection monad will be introduced. Initially, an examination of a new type, denoted as `K`, will reveal its isomorphism to the selection monad `J`. Subsequently, an exploration of the generalization of this `K` type will enhance its intuitive usability. Remarkably, it will be demonstrated that the `J` monad can be embedded into this generalized `K` type.

2 Special K

The following type `K` is to be considered:

```
type K r a = forall b. (a -> (r,b)) -> b
```

While selection functions of type `J` are still in anticipation of a predicate capable of judging their underlying elements, a similar operation is performed by the new `K` type. The predicate of the `K` type also assesses its elements by transforming them into `r` values. Additionally, it converts the `x` into any `y` and returns that `y` along with its judgment `r`.

```
pairK :: K r a -> K r b -> K r (a,b)
pairK f g p = f (\x ->
                g (\y -> let (r, z) = p (x,y)
                         in (r, (r,z))))
```

The previously mentioned inefficiency is now addressed by the definition of `pairK`. This is achieved by examining every element `x` in the selection function `f`. For each element, a corresponding result is extracted from the second selection function `g`. Utilizing the additional flexibility provided by the new `K` type, the property function for `g` is now constructed differently. Instead of merely returning the result `z` along with the corresponding `r` value, a duplicate of the entire result pair calculated by `p` is generated and returned. As this duplicate already represents the complete solution, the entire result for an optimal `x` can now be straightforwardly yielded by `f`, eliminating the need for additional computations.

The sequenceK for this novel K type can be defined as follows:

```
sequenceK :: [K r a] -> K r [a]
sequenceK [e] p    = e (\x -> p [x])
sequenceK (e:es) p = e (\x -> sequenceK es
                        (\xs -> let (r,y) = p (x:xs)
                                in (r,(r,y))))
```

This `sequenceK` implementation employs the same strategy as the earlier `pairK` function. It essentially generates duplicates of the entire solution pair, returning these in place of the result value. The selection function one layer above then unpacks the result pair, allowing the entire solution to be propagated. The efficiency issues previously outlined are addressed by these novel `pairK` and `sequenceK` functions. It will be further demonstrated that this fresh `K` type is isomorphic to the preceding `J` type. This essentially empowers the transformation of every problem previously solved with the `J` type into the world of the `K` type. Subsequently, the solutions can be computed more efficiently before being transformed back to express them in terms of `J`.

2.1 Special K is isomorphic to J

To demonstrate the isomorphism between the new Special `K` type and the `J` type, two operators are introduced for transforming from one type to the other:

```
j2k :: J r a -> K r a
j2k f p = snd (p (f (fst . p)))
```

When provided with a selection function `f` of type `J r a`, the `j2k` operator constructs an entity of type `K r a`. For a given `f :: (a -> r) -> a` and `p :: forall b. (a -> (r,b))`, the objective is to return an entity of type `b`. This is achieved by initially extracting an a from `f` using the constructed property function `(fst . p)`. Subsequently, this a is employed to apply `p`, yielding an `(r,b)` pair, from which the `b` is obtained by applying `snd` to the pair. The transformation of a selection function of type `K` into a selection function of type `J` is accomplished as follows:

```
k2j :: K r a -> J r a
k2j f p = f (\x -> (p x, x))
```

Given a selection function `f :: forall b. (a -> (r,b)) -> b` and a `p :: (a -> r) -> a`, an `a` can be directly extracted from `f` by constructing a property function that utilizes `p` to obtain an `r` value while leaving the corresponding `x` of type a untouched. To validate that these two operators indeed establish an isomorphism between `J` and `K`, the following equations must be proven: `(k2j . j2k) f = f` and `(j2k . k2j) g = g`.

Proof. The equality (k2j . j2k) f = f can be straightforwardly demonstrated by applying all the lambdas and the definitions of fst and snd:

```
(k2j . j2k) f
-- {{ Apply definitions}}
= (\f p -> f (\x -> (p x, x))) (\p -> snd (p (f (fst . p))))
-- {{ Simplyfy }}
= f
```

This proof involves a direct application of lambda expressions and the definitions of `fst` and `snd` for simplification. To facilitate the proof of the second isomorphism, we initially introduce the free theorem for the special K type:

Theorem 1 (Free Theorem for 'K'). *Given the following functions with thier corrisponding types:*

```
g :: forall y. (x -> (r, y)) -> y
h :: Y1 -> Y2
p :: x -> (r, Y1)
```

We have:

```
h (g p) = g (\x -> (id *** g) (p x))
```

With the free theorem for `K`, the other half of the isomorphism can now be proven as follows:

Proof. The equality (j2k . k2j) g = g is established through the following steps:

```
(j2k . k2j) g
-- {{ Apply definitions and simplify}}
= \p -> snd (p (g (\x -> ((fst . p) x, x))))
-- {{ Free Theorem for K }}
= \p -> g (\x -> ((fst . p) x, (snd . p) x))
-- {{ Simplify }}
= g
```

The monad definitions and `sequence` definition for the new `K` type can be derived from the isomorphism. While the desired performance improvements are achieved by the definition of `K`, significant data structure copying is required, only to be deconstructed and discarded at a higher layer. This process significantly complicates the associated definitions for `sequence` and `pair`, rendering them challenging to handle and lacking in intuitiveness. Introducing another type, GK, that returns the entire tuple rather than just the result value seems more intuitive. This exploration is detailed in the following chapter, where similar performance improvements are observed with GK while the definitions become more straightforward. This approach also eliminates the need for unnecessary copying of data. However, it is revealed that GK is not isomorphic to J and K; instead, they can be embedded into GK. Conversely, we will explore a specific precondition under which GK can be embedded into J or K.

3 Generalised K

Consider the more general type GK, derived from the previous special K type:

```
type GK r a = forall b. (a -> (r,b)) -> (r,b)
```

Unlike its predecessor, GK returns the entire pair produced by the predicate, rather than just the result value. The implementation of pairGK for the new GK type no longer necessitates the creation of a copy of the data structure. It suffices to return the result of the predicate's application to the complete pair:

```
pairGK :: GK r a -> GK r b -> GK r (a,b)
pairGK f g p = f (\x -> g (\y -> p (x,y)))
```

In terms of readability, this definition of pairGK is significantly more concise, conveying the essence of the pair function without unnecessary boilerplate code. For every element x :: a within f, all y :: b within g are inspected and judged by the given predicate p. The resulting pair selection function returns the optimal pair of (a,b) values according to the provided predicate. Furthermore, we define sequenceGK as follows:

```
sequenceGK :: [GK r a] -> GK r [a]
sequenceGK [e] p    = e (\x -> p [x])
sequenceGK (e:es) p = e (\x -> sequenceGK es (\xs -> p (x:xs)))
```

Following a similar pattern, this sequenceGK function builds all possible futures for each element within e. Once an optimal list of elements is found, this list is simply returned along with the corresponding r value.

```
bindGK :: GK r a -> (a -> GK r b) -> GK r b
bindGK e f p = e (\x -> f x p)


returnGK :: a -> GK r a
returnGK x p = p x
```

- give pair and sequence
- ilustrate how nice it is to deal with

3.1 Relationship to J and Special K

– Show that generalised K is an embedding

```
-- k2gk :: K r a -> GK r a
-- k2gk f = snd . f

-- gk2k :: GK r a -> K r a
-- gk2k f p =  f (\x -> let (r,y) = p x in (r, (r,y)))
```

– intoduce free theorem and precondition
– counterexamples to ilustrate what precondition means and why we want it
– introduce new theorem baced on free theorem and precondition
– calculate monad definition from k2j and j2k

4 Performance analisys

– give some perfomance analysis examples that ilustrate improvement

5 Related work

J was researched in the context of Sequential games, but slowly found its way to other applications

6 Outlook and future work

– Need to investigate further whats possible with the more general type
– Alpha beta pruning as next step of my work

7 Conclusion

– We should use generalised K istead of J because more useful and more intuitive once understood
– performance improvements are useful
– monad pair and sequence implementation much more intuitive and useful

8 Appendix

Proofs![1]

References

1. Escardó, M., Oliva, P.: What sequential games, the tychonoff theorem and the double-negation shift have in common. In: Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming. pp. 21–32 (2010)

Appendix