

Towards a more efficient Selection Monad

Johannes Hartmann¹, Tom Schrijvers², and Jeremy Gibbons¹

¹ University of Oxford, Department of Computer Science, UK
`firstname.lastname@cs.ox.ac.uk`

² KU Leuven, Department of Computer Science, Belgium,
`tom.schrijvers@kuleuven.be`

Abstract. This paper explores a novel approach to selection functions through the introduction of a generalized selection monad. The foundation is laid with the conventional selection monad J , defined as $(A \rightarrow R) \rightarrow A$, which employs a pair function to compute new selection functions. However, inefficiencies in the original pair function are identified. To address these issues, a specialized type K is introduced, and its isomorphism to J is demonstrated. The paper further generalizes the K type to GK , where performance improvements and enhanced intuitive usability are observed. The embedding between J to GK is established, offering a more efficient and expressive alternative to the well established J type for selection functions. The findings emphasize the advantages of the generalized selection monad and its applicability in diverse scenarios, paving the way for further exploration and optimization.

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

1 Introduction

The selection monad, initially introduced by Paulo Oliva and Martin Escardo [1], serves as a valuable tool for modeling selection-based algorithms in functional programming. Widely explored in the context of sequential games [2], it has been applied to compute solutions for games with perfect information and has found applications in logic and proof theory through the Double-Negation Theorem and the Tychonoff Theorem [2]. Additionally, it has been effectively employed in modeling greedy algorithms [3]. These diverse applications of the selection monad heavily rely on its monadic behavior, particularly emphasizing the use of the *sequence* function for monads.

However, within the context of the selection monad, it becomes apparent that the monadic behavior of the selection monad J is needlessly inefficient. This inefficiency is scrutinized in greater detail through the examination of the *sequence* function, which redundantly duplicates previously calculated work. To address this, the paper introduces two alternative types, namely K and GK , for the selection monad. It establishes that the new K type is isomorphic to the existing J type, conveniently resolving the inefficiency associated with the monadic sequence function. Subsequently, the K type undergoes further generalization into

the *GK* type. The proposition presented in this paper advocates for the adoption of the *GK* type over the traditional *J* type, citing its efficiency advantages. Additionally, the *GK* type is argued to be more intuitive for programming and, given its broader type, provides enhanced versatility for a wide array of applications involving the selection monad.

The upcoming section delves into the selection monad, with a particular focus on the type: $J_{R,A} : (A \rightarrow R) \rightarrow A$ representing selection functions [1]. The exploration of the *pair* function highlights its ability to compute a new selection function based on criteria from two existing functions. Supported by a practical example involving decision-making scenarios and individuals navigating paths, this section underscores the functionality of selection functions. An analysis of the inefficiencies in the original *pair* function identifies redundant computational work. The paper's primary contribution is outlined: an illustration and proposal for an efficient solution to enhance the performance of the *pair* function. This introductory overview sets the stage for a detailed exploration of the selection monad and subsequent discussions on optimizations.

2 Selection Functions

Consider the type for selection functions introduced by Paulo Oliva and Martin Escardo [1] :

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

Now have a look at the following example. Two individuals are walking towards each other on the pavement. A collision is imminent. At this juncture, each individual must decide their next move. This decision-making process can be modeled using selection functions. The decision they need to make is going towards the street the or wall:

```
data Decision = Street | Wall deriving (Eq, Show)
```

The respective selection functions given a property function that tells them what decision is a good one, select the best one. If there are multiple best solutions, they select an arbitrary one. And if there is no correct one, they default to walking towards the wall.

```
s :: J Bool Decision
s p = if p Street then Street else Wall
```

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))
```

To apply the *pair* function, a property function *pair* is needed that will judge two decisions and return *True* if a crash is avoided and *False* otherwise.

```
pred :: (Decision, Decision) -> Bool
pred (d1, d2) = d1 /= d2
```

The *pair* function, merges the two selection functions into a new one that calculates an overall optimal decision.

```
ghci> pair s s pred
(Street,Wall)
```

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 $(A \rightarrow R)$ to determine an optimal *A*. The *pair* function is endowed with a property function *p* of type $((A, B) \rightarrow R)$. 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: $(\lambda x \rightarrow p(x, g(\lambda y \rightarrow p(x, y))))$. Upon the determination of an optimal *A*, a corresponding *B* can then be computed as $g(\lambda y \rightarrow p(a, y))$. 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.

```
ghci> pair' p1 p2 pred
(Wall,Wall)
```

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, 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.

2.1 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 [2]:

```
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.

2.2 Selection monad J

The formation of a monad within the selection functions unfolds as follows [1]:

```
(>>=) :: 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' [] = return []
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.

2.3 Illustration of Sequence in the Context of Selection Functions

To illustrate the application of the sequence function within the domain of selection functions, consider a practical scenario [3]: the task of cracking a secret password. In this hypothetical situation, a black box property function *p* is provided that returns whether the correct password is entered. 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] -> J r a
maxWith xs f = snd (maximumBy (compare `on` fst)
    (map (\x -> (f x , x)) xs))
```

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

```
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:

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

This illustrative example not only showcases the 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 property function for judging individual character; rather, this property function 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.

2.4 Efficiency Issues

This inefficiency will be examined in more detail. When the *sequence* function is utilized for the selection monad, an exhaustive search of all possible combinations of the values underlying the selection functions is executed. It is assumed that the *minWith* function precisely applies the property function p once to each of its elements. The efficiency of the *sequence* function is scrutinized to determine how often the property function p is invoked during the calculation of a solution. Given that *sequence* operates as an exhaustive search resembling a tree search with a branching factor of K , the number of times the property function p is called for a tree of depth n can be expressed as $T(n) = F(n) + T(n - 1)$, where $F(n) = K * T(n - 1)$. Substituting $F(n)$ into $T(n)$ yields $T(n) = K * T(n - 1) + T(n - 1)$. This simplifies to $T(n) = (K + 1)^n$. While an exhaustive search on a tree can be performed with $T(n) = K^n$ calls of p , the *sequence* function duplicates some of the work by forgetting previously computed results.

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 special K type will reveal its isomorphism to the selection monad J . Subsequently, an exploration of the generalization of this K type to the GK type will enhance its intuitive usability. Remarkably, it will be demonstrated that the J monad can be embedded into this general GK type.

3 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 in anticipation of a property function capable of judging their underlying elements, a similar operation is performed by the new K type. The property function of the K type also assesses its elements by transforming them into R values. Additionally, it converts the A into any B and returns that B 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 [] p      = (snd . p) []
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 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 .

3.1 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 $f : J_{R,A}$, the *j2k* operator constructs an entity of type $K_{R,A}$. For a given f of type $(A \rightarrow R) \rightarrow A$ and p of type $\forall B.(A \rightarrow (R,B))$, 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 ($fst \circ p$). 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 of type $\forall B.(A \rightarrow (R,B)) \rightarrow B$ and a p of type $(A \rightarrow R) \rightarrow A$, 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 $J_{R,A}$ and $K_{R,A}$, the following equations must be proven: $(k2j \circ j2k)f = f$ and $(j2k \circ k2j)g = g$.

Proof (J to K Embedding).

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

$$\begin{aligned}
& (k2j \circ j2k)f \\
&= \{ \text{Apply definitions} \} \\
& \quad (\lambda g \ p_2 \rightarrow g(\lambda x \rightarrow (p_2 \ x, x)))(\lambda p_1 \rightarrow snd(p_1(f(fst \circ p_1)))) \\
&= \{ \text{Simplify} \} \\
& \quad f
\end{aligned}$$

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 [4]:

Theorem 1 (Free Theorem for K).

Given the following functions with their corresponding types:

$$\begin{aligned}
& g : K_{R,A} \\
& h : B_1 \rightarrow B_2 \\
& p : A \rightarrow (R, B_1) \\
& *** : (A \rightarrow A') \rightarrow (B \rightarrow B') \rightarrow (A, B) \rightarrow (A', B')
\end{aligned}$$

We have:

$$h(gp) = g((id *** h) \circ p)$$

The free theorem essentially asserts that a function h of type $B_1 \rightarrow B_2$, when applied to the result of a selection function, can also be incorporated into the property function and applied to each individual element. This follows from the generalized type of K , where the only means of generating B_1 values is through the application of p . Consequently, it becomes inconsequential whether h is applied to the final result or to each individual intermediate result. Note that $***$ is the operator that given two functions $f : A \rightarrow A'$ and $g : B \rightarrow B'$ it returns a function of type $(A, B) \rightarrow (A', B')$, where f is applied to the first element of the tuple and g is applied to the second element of the tuple. With the free theorem for K , the remaining portion of the isomorphism can now be demonstrated as follows:

Proof (K to J Embedding).

The equality $(j2k \circ k2j)g = g$ is established through the following steps:

$$\begin{aligned}
& (j2k \circ k2j)g \\
&= \{ \text{Apply definitions and simplify} \} \\
& \quad \lambda p \rightarrow snd(p(g(\lambda x \rightarrow ((fst \circ p)x, x)))) \\
&= \{ \text{Free Theorem for K} \}
\end{aligned}$$

$$\begin{aligned}
& \lambda p \rightarrow g(\lambda x \rightarrow ((fst \circ p)x, (snd \circ p)x)) \\
= & \{ \text{Simplify} \} \\
& g
\end{aligned}$$

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 section, 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 .

4 General 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 property function, 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 property function'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 of type A within f , all y of type B within g are inspected and judged by the given property function p . The resulting pair selection function returns the optimal pair of (A, B) values according to the provided property function. 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.

4.1 Relationship to J and Special K

With the following operators, selection functions of type K can be embedded into GK .

```

gk2k :: GK r a -> K r a
gk2k f = snd . f

k2gk :: K r a -> GK r a
k2gk f p = f (\x -> let (r,y) = p x in (r, (r,y)))

```

Similar to the free theorem for the K type, it is equally possible to derive the free theorem for the new GK type [4]:

Theorem 2 (Free Theorem for GK).

Given the following functions with thier corresponding types:

```

g : GKR,A
f : B1 → B2
p : A → (R, B1)
(***) : (A → A') → (B → B') → (A, B) → (A', B')

```

We have:

$$((id *** f) \circ g)p = g((id *** f) \circ p)$$

This theorem essentially conveys the same concept as the free theorem for K . It asserts that given a function f applied to the result of a selection function, the order of application, whether at the final stage to the ultimate result or inside the property function of the selection function, does not impact the outcome. However, this formulation now accommodates the fact that the GK type also returns the R value. With the free theorem for GK , we can establish that selection functions of type K can be seamlessly embedded into GK :

Theorem 3 (K to GK Embedding).

Given:

$f : K_{R,A}$

We can embedd f into GK as follows:

$$(k2gk \circ gk2k)f = f$$

The proof for this embedding is straight forward utilising the free theorem for GK :

Proof (K to GK Embedding).

Assuming that for:

$f : K_{R,A}$

We can reason:

$$\begin{aligned}
& (gk2k \circ k2gk)f \\
&= \{ \text{Definitions and rewrite} \} \\
& \lambda p \rightarrow (snd \circ f)(\lambda x \rightarrow \text{let } (r,y) = p x \text{ in } (r, (r,y)))
\end{aligned}$$

$$\begin{aligned}
&= \{ \text{Free theorem of } GK \} \\
&\quad \lambda p \rightarrow f(\lambda x \rightarrow \text{let } (r, y) = p \ x \text{ in } (r, \text{snd}(r, y))) \\
&= \{ \text{Simplify} \} \\
&\quad f
\end{aligned}$$

Further we want to establish that selection functions of type GK can be embedded into the K type. This is not possible in general, but we can establish a specific precondition under which this embedding is possible:

Theorem 4 (GK to K Embedding).

Assuming that for:

$$\begin{aligned}
&g : GK_{R,A} \\
&\forall p : (\forall B.(A \rightarrow (R, B))), \exists x : A \text{ such that: } g \ p = p \ x
\end{aligned}$$

We have:

$$(k2gk \circ gk2k)g = g$$

The key requirement is to ensure that the selection function g does not alter the R value subsequent to applying p to its elements. With that precondition we can then prove the embedding as follows:

Proof (GK to K Embedding).

Assuming that for:

$$\begin{aligned}
&g : GK_{R,A} \\
&\forall p : (\forall B.(A \rightarrow (R, B))), \exists x : A \text{ such that: } g \ p = p \ x
\end{aligned}$$

We can reason:

$$\begin{aligned}
&(k2gk \circ gk2k)g \\
&= \{ \text{Definitions and rewrite} \} \\
&\quad \lambda p \rightarrow \text{snd}(g(\lambda x \rightarrow \text{let } (r, y) = p \ x \text{ in } (r, (r, y)))) \\
&= \{ \text{Assumption} \} \\
&\quad \lambda p \rightarrow \text{snd}(\exists x. \text{let } (r, y) = p \ x \text{ in } (r, (r, y))) \\
&= \{ \text{Exists commutes} \} \\
&\quad \lambda p \rightarrow \exists x. \text{let } (r, y) = p \ x \text{ in } \text{snd}(r, (r, y)) \\
&= \{ \text{Assumption} \} \\
&\quad \lambda p \rightarrow g(\lambda x \rightarrow \text{let } (r, y) = p \ x \text{ in } \text{snd}(r, (r, y))) \\
&= \{ \text{Simplify} \} \\
&\quad g
\end{aligned}$$

TODO:

- elaborate on the precondition for the embedding
- counterexamples to illustrate what precondition means and why we want it

5 GK forms a monad

The formation of the monad for GK follows a straightforward definition:

```

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

```

In this context, given a selection function e of type GK_{RA} , a function f of type $A \rightarrow GK_{R,A}$, and a property function p of type $\forall C.(B \rightarrow (R, C))$, the outcome of type (R, C) is assembled through the utilization of e . Each element x of type A underlying e undergoes assessment by applying f . This process yields a pair consisting of the R value, which serves as the basis for judgment, and the result value of type C . As the pair is already of the correct type, a straightforward return suffices.

The return for the GK type is defined as follows:

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

The proofs substantiating the monad laws are annexed in the appendix.

Exploring the alignment of these monad definitions with those of J or K , respectively, is our next objective. The aim is to ensure that the behavior of the GK monad aligns with that of the J and K monads. Therefore, consider the following two operators that transform between GK selection functions and J selection functions:

```
j2gk :: J r x -> GK r x
j2gk f p = p (f (fst . p))

gk2j :: GK r x -> J r x
gk2j f p = snd (f (\x -> (p x, x)))
```

Utilising these operators, we want to show that the GK monad definition aligns with the J monad definition in the case that the GK selection functions fulfill the previously introduced precondition for the embeddings. This is achieved by proving the following theorem:

Theorem 5 (GK Monad Embedding).

Given:

$f : GK_{R,A}$

$g : A \rightarrow GK_{R,B}$

$\forall p : (\forall B.(A \rightarrow (R, B))), \exists x : A$ such that: $g p = p x$ We have:

$$j2gk(gk2j f >>= gk2j \circ g) = bindGkfg$$

To derive the monad definitions from the embedding operators, it is imperative to introduce the following two lemmas:

Lemma 1. *Given:*

$f : (R, B_1) \rightarrow (R, B_2)$

$g : GK_{R,A}$

$p : A \rightarrow (R, B_1)$

We have:

$$fst \circ f \circ p = fst \circ p \implies (f \circ g)p = g(f \circ p)$$

This lemma asserts that when we have a function f applied to the outcome of a selection function of type $GK_{R,A}$, we can similarly apply f to each underlying element of $GK_{R,A}$ within the property function. This is contingent upon the condition that f only modifies the B value and does not alter the R value.

Proof (Lemma 1).

Assuming that for:

- (1) $f : (R, B_1) \rightarrow (R, B_2), g : GK_{R,A}, p : A \rightarrow (R, B_1)$
- (2) $\forall p : (\forall B.(A \rightarrow (R, B))), \exists x : A$ such that $g p = p x$
- (3) $f st \circ f \circ p = f st \circ p$

We can reason:

$$\begin{aligned}
& f(g p) \\
&= \{ \text{Assumption (2)} \} \\
& \quad \exists x. f(p x) \\
&= \{ \text{Rewrite as tuple} \} \\
& \quad \exists x. ((fst \circ f \circ p)x, (snd \circ f \circ p)x) \\
&= \{ \text{Assumption (3)} \} \\
& \quad \exists x. ((fst \circ p)x, (snd \circ f \circ p)x) \\
&= \{ \text{Rewrite as lambda} \} \\
& \quad \exists x. (\lambda(r, y) \rightarrow (r, (snd \circ f)(r, y))) p x \\
&= \{ \text{Assumption (2)} \} \\
& \quad (\lambda(r, y) \rightarrow (r, (snd \circ f)(r, y))) g p \\
&= \{ \text{Free Theorem for } GK \} \\
& \quad g((\lambda(r, y) \rightarrow (r, (snd \circ f)(r, y))) \circ p) \\
&= \{ \text{Rewrite} \} \\
& \quad g(\lambda x \rightarrow ((fst \circ p)x, (snd \circ f \circ p)x)) \\
&= \{ \text{Assumption (3)} \} \\
& \quad g(\lambda x \rightarrow ((fst \circ f \circ p)x, (snd \circ f \circ p)x)) \\
&= \{ \text{Simplify} \} \\
& \quad g(f \circ p)
\end{aligned}$$

To further simplify the calculation we also introduce the following lemma:

Lemma 2. *If q does apply p to get the R value but keeps the original value, and we then use that original value to compute the (R, Z) values with p we can call g with p directly.*

Given:

$$\begin{aligned}
p &:: A \rightarrow (R, B) \\
g &:: K_{R,A}
\end{aligned}$$

We have:

$$(p \circ snd)(g q) = g p \text{ where } q = \lambda x \rightarrow ((fst \circ p)x, x)$$

And we can proof Lemma 2 by utilising Lemma 1.

Proof (Lemma 2).

$$\begin{aligned}
& (p \circ \text{snd})(g \, q) \\
= & \{ \text{Definition of } q \} \\
& (p \circ \text{snd})(g (\lambda x \rightarrow ((fst \circ p)x, x))) \\
= & \{ \text{Lemma 1} \} \\
& g(\lambda x \rightarrow (p \circ \text{snd})((fst \circ p)x, x)) \\
= & \{ \text{Simplify} \} \\
& g \, p \\
\iff & \\
& (fst \circ p \circ \text{snd})(\lambda x \rightarrow ((fst \circ p)x, x)) \\
= & \{ \text{Simplify} \} \\
& \lambda y \rightarrow (fst(p(\text{snd}((\lambda x \rightarrow ((fst \circ p)x, x))y)))) \\
= & \{ \text{Simplify} \} \\
& \lambda y \rightarrow (fst(p(\text{snd}((fst \circ p)y, y)))) \\
= & \{ \text{Simplify} \} \\
& \lambda x \rightarrow (fst \circ p)x \\
= & \{ \text{Simplify} \} \\
& fst \circ (\lambda x \rightarrow ((fst \circ p)x, x))
\end{aligned}$$

– TODO: Give an intuition what these theorems mean

We can calculate the bind implementation for GK with the $j2gk$ and $gk2j$ operators and the previously introduced theorems:

Proof (GK Monad behaves similar to J).

$$\begin{aligned}
& j2gk(gk2j \, f \gg= gk2j \circ g) \\
= & \{ \text{Definition of } J_{\gg=} \} \\
& j2gk((\lambda f \, g \, p \rightarrow g(f(p \circ \text{flip } g \, p))p)(gk2j \, f)(gk2j \circ g)) \\
= & \{ \text{simplify} \} \\
& j2gk(\lambda p \rightarrow gk2j(g(gk2j \, f(p \circ (\lambda x \rightarrow gk2j(g \, x)p))))p) \\
= & \{ \text{Definition of } j2k \text{ and rewrite} \} \\
& \lambda p \rightarrow p(gk2j(g(gk2j \, f(\lambda x \rightarrow fst((p \circ \text{snd})((gx)(\lambda x \rightarrow ((fst \circ p)x, x)))))))(fst \circ p)) \\
= & \{ \text{Lemma 1} \} \\
& \lambda p \rightarrow p(gk2j(g(gk2j \, f(\lambda x \rightarrow fst(((gx)(\lambda x \rightarrow (p \circ \text{snd})((fst \circ p)x, x)))))))(fst \circ p)) \\
= & \{ \text{Definition of } j2gk \text{ and rewrite} \} \\
& \lambda p \rightarrow p(\text{snd}(g(\text{snd}(f(\lambda x \rightarrow (fst(g \, x \, p), x))))(\lambda x \rightarrow ((fst \circ p)x, x)))) \\
= & \{ \text{Lemma 2} \} \\
& \lambda p \rightarrow g(\text{snd}(f(\lambda x \rightarrow (fst(g \, x \, p), x))))p \\
= & \{ \text{Rewrite} \} \\
& \lambda p \rightarrow (\lambda y \rightarrow g(\text{snd } y)p)(f(\lambda x \rightarrow (fst(g \, x \, p), x))) \\
= & \{ \text{Lemma 1} \} \\
& \lambda p \rightarrow f((\lambda y \rightarrow g(\text{snd } y)p) \circ (\lambda x \rightarrow (fst(g \, x \, p), x))) \\
= & \{ \text{Simplify} \} \\
& \lambda p \rightarrow f(\lambda x \rightarrow g \, x \, p)
\end{aligned}$$

This shows that all *GK* selection functions satisfying the precondition behave the same when transformed to *K* or *J* selection functions.

- TODO: illustrate how nice it is to deal with

6 Performance analysis

In this section we will compare the performance of the *J*, *K* and *GK* monads. As all 3 of them will perform a basic exhaustive search, we will compare the number of calls to the property function *p*, as well as the time it takes for each of the monads to search the same tree.

Given the following `maxWith` functions:

```
maxWithJ :: Ord r => [a] -> J r a
maxWithJ xs f = snd (maximumBy (compare `on` fst)
                          (map (\x -> (f x, x)) xs))

maxWithK :: Ord r => [a] -> K r a
maxWithK xs f = snd (maximumBy (compare `on` fst) (map f xs))

maxWithGK :: Ord r => [a] -> GK r a
maxWithGK xs f = maximumBy (compare `on` fst) (map f xs)
```

First we compare the runtime for each type. For this we will use the following property functions:

```
pJ :: [Int] -> Int
pJ = sum

pK :: [Int] -> (Int, [Int])
pK x = (sum x, x)
```

and the following lists of selection functions:

```
js :: [J Int Int]
js = replicate 6 (maxWithJ [1..10])

ks :: [K Int Int]
ks = replicate 6 (maxWithK [1..10])

gks :: [GK Int Int]
gks = replicate 6 (maxWithGK [1..10])
```

We get the following result for a search tree of the size 10^6 :

```
ghci> sequence js pJ
[10,10,10,10,10,10]
(3.69 secs, 1,612,913,328 bytes)

ghci> sequenceK ks pK
[10,10,10,10,10,10]
```

```
(2.85 secs, 2,431,196,064 bytes)
```

```
sequenceGK gks pK
(60, [10,10,10,10,10,10])
(1.56 secs, 869,778,256 bytes)
```

This already demonstrates a significant improvement in performance for the *GK* and *K* monad. It further highlights the space efficiency of the *GK* monad over the *J* and *K* monad.

TODO:

- Make a graph that shows the exponential growth for each type
- Use trace to count the number of calls to p for each type and show that the *GK* and *k* monad only calls p once for each element in the list
- Do some performance analysis using the isomorphism and the embedding to show that even transforming between the monads leads to a performance increase

7 Related work

- J was researched in the context of Sequential games, but slowly found its way to other applications
- It can also be used for greedy algorithms, however this performance optimisation does not apply in this case
- But greedy algorithms can also be represented with the new General selection monad

8 Outlook and future work

- Need to investigate further what's possible with the more general type
- Alpha beta pruning as next step of my work

9 Conclusion

- We should use General K instead of J because more useful and more intuitive once understood
- performance improvements are useful
- monad, pair, and sequence implementation much more intuitive and useful

References

1. Escardó, M., Oliva, P.: Selection functions, bar recursion and backward induction. *Math. Struct. Comput. Sci.* **20**(2), 127–168 (2010)

2. 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)
3. Hartmann, J., Gibbons, J.: Algorithm design with the selection monad. In: International Symposium on Trends in Functional Programming. pp. 126–143. Springer (2022)
4. Wadler, P.: Theorems for free! In: Proceedings of the fourth international conference on Functional programming languages and computer architecture. pp. 347–359 (1989)

Appendix

Proof Monad Laws for GK

Proof (Left identity).

```

return a >>= h
= (flip ($)) a >>= h
= (\p -> p a) >>= h
= \p' -> (\p -> p a) ((flip h) p')
= \p' -> ((flip h) p') a
= \p' -> h a p'
= h a

```

Proof (Right identity).

```

m >>= return
= \p -> m ((flip return) p)
= \p -> m ((flip (flip ($))) p)
= \p -> m (($) p)
= \p -> m p
= m

```

Proof (Associativity).

```

(m >>= g) >>= h
= \p -> (m >>= g) ((flip h) p)
= \p -> (\p' -> m ((flip g) p')) ((flip h) p)
= \p -> (m ((flip g) ((flip h) p)))
= \p -> m ((\y x -> g x y) ((flip h) p))
= \p -> m ((\x -> g x ((flip h) p)))
= \p -> m ((\p' x -> (g x) ((flip h) p')) p)
= \p -> m ((flip (\x p' -> (g x) ((flip h) p')) p)
= \p -> m ((flip (\x -> (\p' -> (g x) ((flip h) p')))) p)
= \p -> m ((flip (\x -> g x >>= h)) p)
= m >>= (\x -> g x >>= h)

```