Towards a more efficient 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] 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 generalised selection monad. The foundation is laid with the conventional selection monad J, defined as $(A \to R) \to A$, which employs a pair function to compute new selection functions. However, inefficiencies in the original pair function are identified. To address these issues, a specialised type K is introduced, and its isomorphism to J is demonstrated. The paper further generalises 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 emphasise the advantages of the generalised selection monad and its applicability in diverse scenarios, paving the way for further exploration and optimisation.

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 emphasising 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 scrutinised 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 generalisation 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: $J_{R,A} : (A \to R) \to A$ 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 optimisations.

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 either 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 optimal 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 *pred* 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 conceptualised 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 $(A \to R)$ to determine an optimal A. The *pair* function is endowed with a property function p of type $((A, B) \to R)$. Through the utilisation 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: $(\lambda x \to p(x, g(\lambda y \to p(x, y))))$. Upon the determination of an optimal A, a corresponding B can then be computed as $g(\lambda y \to p(a, y))$. In this case, the *pair* function can be conceptualised 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 generalisation 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 for 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 optimisation in subsequent sections of this 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
(>>=) e f p = f (e (p . flip f 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, here 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 specific to the J type. 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 emphasises 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 utilisation 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 utilised 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 scrutinised 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 generalisation 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. Utilising 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 special 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 $(A \to R) \to A$ and p of type $\forall B.(A \to (R, B))$, 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 $\forall B.(A \to (R, B)) \to B$ and a p of type $(A \to R) \to A$, an A can be directly extracted from f by constructing a property function that utilises 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).

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

The proof utilises the direct application of lambda expressions and the definitions of fst and snd for simplification. The facilitation of the proof for the second isomorphism involves the initial introduction of the free theorem for the special K type [5]:

Theorem 1 (Free Theorem for K).
Given the following functions with their corresponding types:

$g : K_{R,A}$
$h : B_1 \to B_2$
$p : A \to (R, B_1)$
$* * * : (A \to A') \to (B \to B') \to (A, B) \to (A', B')$

It follows:

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

The free theorem essentially asserts that a function h of type $B_1 \to B_2$, 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 generalised type of K, where the only means of generating B_1 values is through the application of p. Therefore, 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 $f : A \to A'$ and $g : B \to B'$ it returns a function of type $(A, B) \to (A', B')$, 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:

$$
\begin{aligned}
& (j2k \circ k2j)g \\
= \quad & \{ \text{ Apply definitions and simplify } \}
\end{aligned}
$$

$$
\begin{aligned}
& \lambda p \to snd(p(g(\lambda x \to ((fst \circ p)x, x)))) \\
= \quad & \{ \text{ Free Theorem for } K \ \} \\
& \lambda p \to g(\lambda x \to ((fst \circ p)x, (snd \circ p)x)) \\
= \quad & \{ \text{ Simplify } \} \\
& g
\end{aligned}
$$

The monad definitions and *sequence* definition for the new K type can be derived from this isomorphism. While the definition of K achieves the desired performance improvements, it necessitates significant copying of data structures, which are subsequently deconstructed and discarded at a higher layer. This necessity significantly complicates the associated definitions for *sequence* and *pair*, making them challenging to handle and less intuitive.
The introduction of another type, GK, which returns the entire tuple rather than merely the result value, appears more intuitive. This shift is elaborated upon in the following section, where GK is observed to facilitate similar performance improvements while simplifying the definitions. This method also removes the need for unnecessary data copying. Nevertheless, it is disclosed that GK is not isomorphic to J and K but rather can be embedded into GK. In contrast, an investigation into a specific precondition allowing for GK to be embedded into J or K is presented.

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, the definition of *pairGK* is significantly more concise, with the essence of the *pair* function being conveyed without unnecessary boilerplate code. Every element x of type A within f is inspected and evaluated by the given property function p for all y of type B within g. The optimal pair of (A, B) values is returned by the resulting pair selection function according to the provided property function. Furthermore, *sequenceGK* is defined as follows:

```
sequenceGK :: [GK r a] -> GK r [a]
sequenceGK [] p     = p []
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 of 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 [5] for the new GK type:

Theorem 2 (Free Theorem for GK).
Given the following functions with their corresponding types:
$g : GK_{R,A}$
$f : B_1 \to B_2$
$p : A \to (R, B_1)$
$* * * : (A \to A') \to (B \to B') \to (A, B) \to (A', B')$

It follows:

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

This theorem communicates a concept similar to the free theorem for K. It suggests that the outcome remains unchanged whether a function f is applied directly to the final result of a selection function or within the selection function's property function. This idea is now adapted to include the behavior of the GK type, which also returns the R value.
By using the free theorem for GK, it becomes clear that selection functions designed for the K type can be directly embedded into the GK structure:

Theorem 3 (K to GK Embedding).
Given:
$f : K_{R,A}$
The following embedding of f into GK 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:
$f : K_{R,A}$
It can be reasoned:
$(gk2k \circ k2gk)f$
$=$ { Definitions and rewrite }
$\lambda p \to (snd \circ f)(\lambda x \to \text{ let } (r, y) = p\, x \text{ in } (r, (r, y)))$

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

Further, to establish that selection functions of type GK can be embedded into the K type a specific precondition is introduced, under which this embedding is possible:

Theorem 4 (GK to K Embedding).
Assuming that for:
$g : GK_{R,A}$
$\forall p : \forall B.A \to (R, B), \exists x : A$ *such that:* $g\, p = p\, x$
It follows:

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

The essential condition is that the selection function g should not modify the R value after p has been applied to its elements. Given this precondition, the embedding can be proven as follows:

Proof (GK to K Embedding).
Assuming that for:
$g : GK_{R,A}$
$\forall p : \forall B.A \to (R, B), \exists x : A$ such that: $g\, p = p\, x$
It can be reasoned:

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

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 $GK_{R,A}$, a function f of type $A \to GK_{R,A}$, and a property function p of type $\forall C.B \to (R, C)$, the outcome of type (R, C) is assembled through the utilisation 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, it can be shown 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 embedding. This is achieved by proving the following theorem:

Theorem 5 (GK Monad Embedding).
Given:
$f : GK_{R,A}$
$g : a \to GK_{R,B}$
$\forall p : \forall B.A \to (R, B), \exists x : A$ *such that:* $g\, p = p\, x$ *It follows:*

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

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

Lemma 1. *Given:*
$f : (R, B_1) \to (R, B_2)$
$g : GK_{R,A}$
$p : A \to (R, B_1)$
It follows:

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

This lemma asserts that given a function f acting upon the result of a selection function of type $GK_{R,A}$, it is possible to apply f to each element of $GK_{R,A}$ within the property function, provided f solely transforms the B value without affecting the R value.

Proof (Lemma 1).
Assuming that for:
(1) $f : (R, B_1) \to (R, B_2), g : GK_{R,A}, p : A \to (R, B_1)$
(2) $\forall p : \forall B.A \to (R, B), \exists x : A$ such that $g\ p = p\ x$
(3) $fst \circ f \circ p = fst \circ p$
It can be reasoned:

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

To further simplify the calculation the following lemma is introduced:

Lemma 2. *Let q be a function that applies p to obtain the R value while preserving the original value. If this original value is subsequently utilised to compute the (R, Z) values using p, then g can be invoked directly with p.*
Given:
$p :: A \to (R, B)$
$g :: K_{R,A}$

It follows:

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

To prove Lemma 2, Lemma 1 is utilised:

Proof (Lemma 2).

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

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

Now it is possible calculate the *bindGK* implementation for GK with the $j2gk$ and $gk2j$ operators and the previously introduced theorems:

Proof (GK Monad behaves similar to J).

$$
\begin{aligned}
& j2gk(gk2j\ f >>= gk2j \circ g) \\
= \quad & \{ \text{ Definition of } J_{>>=} \ \} \\
& j2gk((\lambda f\ g\ p \to g(f(p \circ flip\ g\ p))p)(gk2j\ f)(gk2j \circ g)) \\
= \quad & \{ \text{ simplify } \} \\
& j2gk(\lambda p \to gk2j(g(gk2j\ f(p \circ (\lambda x \to gk2j(g\ x)p))))p) \\
= \quad & \{ \text{ Definition of j2k and rewrite } \} \\
& \lambda p \to p(gk2j(g(gk2jf(\lambda x \to fst((p \circ snd)((gx)(\lambda x \to ((fst \circ p)x, x)))))))(fst \circ \\
& p)) \\
= \quad & \{ \text{ Lemma 1 } \} \\
& \lambda p \to p(gk2j(g(gk2jf(\lambda x \to fst(((gx)(\lambda x \to (p \circ snd)((fst \circ p)x, x)))))))(fst \circ \\
& p)) \\
= \quad & \{ \text{ Definition of j2gk and rewrite } \} \\
& \lambda p \to p(snd(g(snd(f(\lambda x \to (fst(g\ x\ p), x))))(\lambda x \to ((fst \circ p)x, x)))) \\
= \quad & \{ \text{ Lemma 2 } \} \\
& \lambda p \to g(snd(f(\lambda x \to (fst(g\ x\ p), x))))p \\
= \quad & \{ \text{ Rewrite } \} \\
& \lambda p \to (\lambda y \to g(snd\ y)p)(f(\lambda x \to (fst(g\ x\ p), x))) \\
= \quad & \{ \text{ Lemma 1 } \} \\
& \lambda p \to f((\lambda y \to g(snd\ y)p) \circ (\lambda x \to (fst(g\ x\ p), x))) \\
= \quad & \{ \text{ Simplify } \} \\
& \lambda p \to f(\lambda x \to g\ x\ p)
\end{aligned}
$$

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

6 Performance Analysis

In this section, the performance of the J, K, and GK monads will be compared. All three are designed to perform an exhaustive search, exploring the complete problem space to select the best possible solution. The comparison will focus on the number of calls to the property function p, as well as the time taken for each of the monads to calculate a solution.
Given the following $maxWith$ functions for each of the monad types:

```
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)
```

6.1 Runtime Analysis

Initially, the runtime, while exploring a particular search space for each type, will be compared. For this purpose, the following basic property functions will be utilised. These functions simply sum up the elements of a given list of integers.

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

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

A list of selection functions for each type is further defined. A list of integers is searched by each individual selection function, which then selects the integer that will maximise a given property function.

```
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])
```

Considering a list of selection functions with a length of 6, where each selection function explores 10 possible elements, the search space size is 10^6. This search space can be conceptualised as a tree with a depth n of 6 and a branching factor K of 10. By employing the respective *sequence* function for each type, along with the corresponding property function, an initial analysis of runtime and space complexity was conducted within GHCI.

```
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)

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

The results obtained from GHCI already demonstrate 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, while further showing the significant memory overhead of the K type, that is due to the nested duplications of the final solution.
For a more robust performance analysis, the runtime of each type was tested with increasing depth of the search tree, i.e., longer lists of selection functions. This analysis was performed on the compiled version of the code to utilise any potential performance improvements the compiler offers.

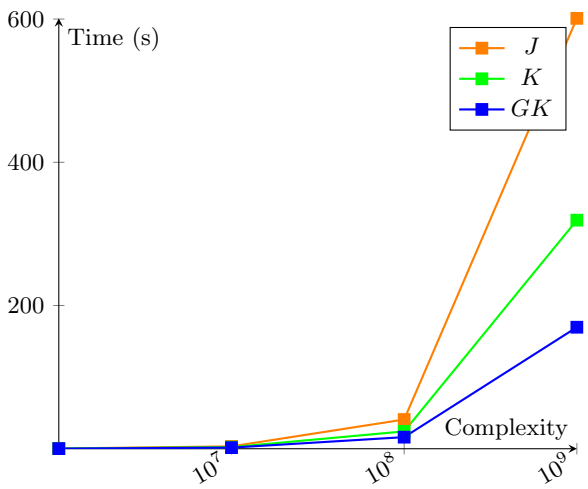

Fig. 1. Runtime of J, K, and GK with increasing complexity

Figure 1 plots the runtime of the compiled Haskell code for each monad J, K, and GK as they navigate an increasingly complex search space. The graph demonstrates a consistent trend where the GK monad outperforms the others J and K types. As the depth of the search tree increases, the gap in performance becomes

even more pronounced, clearly showcasing the efficiency and effectiveness of the generalised selection monad approach.
Additionally, by employing Haskell's trace debug option, the frequency with which the property function was invoked for each monad was tallied. Through this method, verification was achieved that for the J monad, its property function is indeed called $(K + 1)^n$ times. Conversely, owing to the performance enhancements, the K and GK monads necessitate only K^n calls to their property function, where K represents the branching factor and n the depth of the search tree being explored.

7 Related and Future Work

The exploration of the selection monad, particularly the J type, has been predominantly focused on sequential games with total information[2]. This line of research has primarily employed a minimax algorithm to calculate optimal strategies for these games, showcasing the utility of the J monad in navigating complex decision-making processes. Beyond this, the selection monad's applications have extended into logic, proof theory [2], and algorithm design. Notably, within algorithm design, the J monad can also be utilised in modeling greedy algorithms [3].
When considering the implementation of greedy algorithms via the J monad, it is important to acknowledge that the performance optimisations proposed in this paper do not extend to these algorithms. The reason is that the greedy algorithm approach is already optimal when applied through the J type, without unnecessarily duplicating any computations. However, this revelation paves the way for further research into the modeling of greedy algorithms using the new GK type. Investigating this could determine if the efficiencies intrinsic to the GK monad might present any benefits for greedy algorithms, given their already optimal performance under the J type.
Jules Hedges' contributions have laid a solid foundation in understanding the monad transformer for the conventional J selection monad [4]. This work has illuminated the potential for integrating selection functions into more intricate computational constructs. Future research should consider extending these insights to the GK type through the development of a corresponding monad transformer. Such endeavors could reveal new applications, potentially enhancing the computational efficiency and expressiveness of functional programming paradigms that leverage selection monads.
The advent of the GK type, marks a progression by mitigating redundant computations, thus yielding performance enhancements in specific scenarios. Nonetheless, it is imperative to maintain a measured perspective on the impact of these advancements. The efficiency improvements offered by the GK monad do not tackle the intrinsic challenge of exponential time complexity that is a hallmark of exhaustive search strategies. Although reducing unnecessary computations is a noteworthy optimisation, the broader issue of exhaustive search strategies' computational demands remains largely unchanged. Future research might explore

the feasibility of incorporating alpha-beta pruning into the minimax algorithm, potentially offering a strategy to mitigate the computational intensity of exhaustive searches.

8 Conclusion

This paper presents a compelling case for the adoption of the new general selection monad type GK over the conventional J type in the realm of functional programming, particularly within the context of selection functions. The introduction of the GK monad marks a significant advancement in the field, offering not only performance improvements but also a more intuitive and practical approach to monad, pair, and sequence implementations.

The core argument for transitioning to the GK monad stems from its utility and intuitive nature, which, though may require a slight learning curve, ultimately provides a more efficient and user-friendly programming experience. The performance enhancements associated with the GK monad are not merely theoretical but have practical implications for the execution of complex algorithms and the overall computational efficiency.

Furthermore, the GK monad's design facilitates a more intuitive understanding and implementation of monad, pair, and sequence constructs for selection functions, which are central to functional programming paradigms. This intuitiveness, coupled with the performance gains, makes the GK type an attractive alternative to the J type.

A pivotal finding of this research is that all GK constructs meeting the specific precondition can be seamlessly embedded into the J type. This implies that any model or algorithm previously framed within the J type can be transitioned to, or represented in, the GK framework without loss of functionality. Consequently, it is advocated that future research and development in the selection monad domain pivot towards the GK type. This shift is recommended not only because of the aforementioned performance and usability benefits but also to harness the full potential of the GK type's more advanced and efficient approach to handling selection functions.

In light of these findings, it is proposed that ongoing and future work in the selection monad sphere should utilise the advantages of the presented GK type. This involves translating existing work from the J framework to the GK framework, thus leveraging the GK type's advantages to foster a more efficient, intuitive, and robust functional programming environment. The transition to the GK type represents a forward-thinking approach to functional programming, promising improvements in both the development and execution of complex computational tasks.

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. Hedges, J.: Monad transformers for backtracking search. arXiv preprint arXiv:1406.2058 (2014)
5. 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)
```