Generalising the 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, Department of ComputerScience, 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 -> r) -> 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 to the Selection Monad

This section introduces the selection monad, focusing on the `type J r a = (a -> r) -> a` for selection functions [1]. 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 the `pair` functions performance. This introductory overview sets the stage for a detailed exploration of the selection monad and subsequent discussion on optimizations.

1.1 Selection functions

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

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

Consider 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 the wall:

```
data Decision = Street | Wall deriving Show
```

The respective selection functions decide given a property function that tells them what decision is the correct one, select the correct 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 (Wall,Street) = True
pred (Street,Wall) = True
pred _             = False
```

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 -> 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.

```
ghci> 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.

1.2 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.

1.3 Selection monad

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' (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.

1.4 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 [2]: the task of cracking a secret password. In this hypothetical situation, a black box property function `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 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 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 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. 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 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 `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 (J to K Embedding). 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))))
{{Simplify }}
= 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 [3]:

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

```
g :: forall b. (a -> (r, b)) -> b
h :: b1 -> b2
p :: a -> (r, b1)
```

We have:

```
h (g p) = g (\x -> let (r,y) = (p x) in (r,h y))
```

The free theorem essentially asserts that a function `h :: y1 -> y2`, 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 `y1` 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. 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 . 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 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`.

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 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 :: a` within `f`, all `y :: 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.

3.1 Relationship to J and Special K

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

```
gk2k :: forall r a b. ((a -> (r,b)) -> (r,b)) -> ((a -> (r,b)) -> b)
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 theroem for the `K` type, it is also possible to derive the free theorem for the `GK` type:

Theorem 2 (Free Theorem for GK). *Given the following functions with thier corresponding types:*

```
g :: forall b. (\a -> (r,b)) -> (r,b)
f :: b1 -> b2
p :: a -> (r, b1)
```

We have:

```
((id *** f) . g) p = g ((id *** f) . p)
```

It is basically stating the same as the free Theorem for `K`, where given a function `f` that is applied to the result of a selection function, it dosent matter if this is done in the end to the final result, or inside the property function of the selection function. But it now needs to account for the fact that the `GK` type is also returning the `r` value.

With the free theorem for `GK` we can now proof that selection functions of type `K` can be embedded into `GK`:

Proof (K to GK Embedding). The equality (k2gk . gk2k) f = f is established through the following steps:

Assuming: f :: K r a

```
(gk2k . k2gk) f
{{ Definitions and rewrite }}
= (\p -> (snd . f) (\x -> let (r,y) = p x in (r, (r,y))))
{{ Free theorem of GK }}
= (\p -> f (\x -> let (r,y) = p x in (r, snd (r,y))))
{{ Simplify }}
= f
```

Embedding `K` selection functions into the new `GK` type is a little bit more tricky. We essentially need to make sure that `g` is not changing the `r` value after applying `p` to it's elements. Therefore

Proof (GK to K Embedding). The equality (k2gk . gk2k) g = g is established through the following steps:

Assuming that for g :: GK r a forall p :: forall b . (a -¿ (r,b)) exists x :: a such that: g p = p x

```
(k2gk . gk2k) g
{{ Definitions and rewrite }}
= \p -> snd (g(\x -> let (r,y) = p x in (r, (r,y))))
{{ Assumption }}
= \p -> snd (exist x -> let (r,y) = p x in (r, (r,y)))
{{ exists commuts }}
= \p -> exists x -> let (r,y) = p x in snd (r, (r,y))
{{ Assumption }}
= \p -> g (\x -> let (r,y) = p x in snd (r, (r,y)))
{{ Simplify }}
= g
```

- 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 GK forms a monad

The monad definition for `GK` is straightforward:

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

Given a selection function `e :: GK r a`, a function `f :: a -> GK r b`, and a property function
`p :: forall c. (b -> (r,c))`, the result of type `(r,c)` can be constructed by utilizing `e`. Each underlying element `x :: a` of `e` will be assessed based on the values produced by applying `f` to each element `x`. This process results in a pair comprising the `r` value by which the outcome is judged and the result value of type `c`. Since this pair is already of the correct type, it is sufficient to simply return it.

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

The proofs for the monad laws are attached in the appendix.

With these monad definitions, we'd like to investigate how they relate to the definitions for `J` or `K` respectively. We'd like the `GK` monad to behave in the same way as the `J` and `K` monad does.

In order to dirive we need to introduce the following two theorems:

Theorem 3 (Theorem 1).

```
f :: (r,a) -> (r,b)
g :: K r x
p :: x -> (r,a)
f (g p) = g (f . p)
```

iff $(fst \,.\, f \,.\, p) = fst \,.\, p$

Proof (Theorem 1).

```
Assuming that for
g :: GK r a
forall p :: forall b . (a -> (r,b))
exists x :: a
such that:
g p = p x

f (g p)
{{ Assumption }}
= exists x -> f (p x)
{{ rewrite as tuple }}
= ((fst . f . p) x, (snd . f . p) x)
{{ Theorem 1 condition }}
= ((fst . p ) x , (snd . f . p) x)
{{ rewrite as let }}
= let (r, y) = p x in (r, snd (f (r,y)))
{{ rewrite as *** }}
= let (r, y) = p x in (id *** (\y -> snd (f (r,y)))) (r, y)
{{ resolve *** }}
= let (r, y) = p x in (\(a,b) -> (a, (\y -> snd (f (r,y))) b)) (r, y)
{{ apply lambda }}
= let (r, y) = p x in (\(a,b) -> (a, snd (f (r,b)))) (r, y)
{{ expand let }}
= let r = fst (p x) in let y = snd (p x) in (\(a,b) -> (a, snd (f (r,b)))) (r, y)
{{ remove let }}
= (\(a,b) -> (a, snd (f (fst (p x), b)))) ((fst (p x)), (snd (p x)))
{{ simplify }}
= (\(a,b) -> (a, snd (f (fst (p x), b)))) p x
{{ remove patternmatch in lambda }}
= (\a -> (fst a, snd (f (fst (p x), snd a)))) p x
{{ replace (p x) with a within lambda }}
= (\a -> (fst a, snd (f (fst a, snd a)))) p x
{{ add patternmatch to lamvda }}
= (\(r,y) -> (r, snd (f (r, y)))) p x
{{ Assumption }}
= (\(r,y) -> (r, snd (f (r, y)))) g p
{{ free Theorem for GK }}
= g ((\(r,y) -> (r,  snd (f (r,y)))) . p)
{{ rewrite (.) }}
= g (\x -> (\(r,y) -> (r,  snd (f (r,y)))) (p x))
{{ pull (p x) into lambda }}
= g (\x -> (fst (p x),  snd (f (fst (p x) ,snd (p x)))))
{{ simplify tuple to p x }}
= g (\x -> (fst (p x),  snd (f (p x))))
{{ rewrite with (.) }}
= g (\x -> ((fst . p) x, (snd . f . p) x))
{{ expand first bit with theorem condition }}
= g (\x -> ((fst . f . p) x, (snd . f . p) x))
```

```
{{ simplify tuple to (f . p) x }}
= g (\x -> (f . p) x)
{{ remove lambda }}
= g (f . p)
```

To further simplify the calculation we aslso introduce the following theorem:

Theorem 4 (Theorem 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*

```
-- p :: x -> (r,y)
-- g :: K r x
-- p (snd (g q)) = g p
--     where q = (\x -> ((fst . p) x, x))
```

And we can proof Theorem 2 by utilising Theorem 1.

Proof (Theorem 1).

```
(p . snd) (g q)
= g (\x -> (p . snd) ((fst . p) x, x))
= g p

iff
(fst . p . snd) (\x -> ((fst . p) x, x))
= \y -> (fst ( p (snd ( (\x -> ((fst . p) x, x)) y))))
= \y -> (fst(p(snd ((fst . p) y, y) )))
= \x -> (fst . p) x
= fst . (\x -> ((fst . p) x, x))
```

– TODO: Give an intuition what these theorems mean

Now, 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)))
```

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).

```
j2gk ((>>=) (gk2j f) (\x -> gk2j (g x)))
{{ Definition of (>>=) }}
= j2gk ((\f g p -> g (f (p . flip g p)) p) (gk2j f) (\x -> gk2j (g x)))
{{ Simplify }}
```

```
= j2gk (\p -> gk2j (g (gk2j f (p . (\x -> gk2j (g x) p)))) p)
{{ Definition of j2gk and rewrite }}
= \p -> p (gk2j (g (gk2j f (\x -> fst ((p . snd) ((g x) (\x -> ((fst . p) x, x))))
{{ Theorem 1 }}
= \p -> p (gk2j (g (gk2j f (\x -> fst (((g x) (\x -> (p . snd) ((fst . p) x, x))))
{{Definition of j2gk and rewrite }}
= \p -> p (snd (g (snd (f (\x -> (fst (g x p), x)))) (\x -> ((fst . p) x, x))))
{{ Theorem 2 }}
= \p -> g (snd (f (\x -> (fst (g x p), x)))) p
{{ Rewrite }}
= \p -> (\y -> g (snd y) p) (f (\x -> (fst (g x p), x)))
{{ Theorem 1 }}
= \p -> f ((\y -> g (snd y) p) . (\x -> (fst (g x p), x)))
{{ Simplify }}
\p -> f (\x -> g x p)
```

This shows that all `GK` selection functions behave the same when transforemd to `K` or `J` selection functions.

– TODO: ilustrate how nice it is to deal with

5 Performance analisys

– give some perfomance analysis examples that ilustrate improvement
– Done by an example and use trace to count calls of P

6 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 algorythms, however this performance optimisation does not apply in this case
– But greedy algorythms can also be represented with the new generalised selection monad

7 Outlook and future work

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

8 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

References

1. Escardó, M., Oliva, P.: Selection functions, bar recursion and backward induction. Math. Struct. Comput. Sci. **20**(2), 127–168 (2010)
2. Hartmann, J., Gibbons, J.: Algorithm design with the selection monad. In: International Symposium on Trends in Functional Programming. pp. 126–143. Springer (2022)
3. 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)
```