The selection monad as a CPS translation

Jules Hedges

February 28, 2015

Abstract

A computation in the continuation monad returns a final result given a continuation, ie. it is a function with type $(X \to R) \to R$. If we instead return the intermediate result at X then our computation is called a selection function. Selection functions appear in diverse areas of mathematics and computer science (especially game theory, proof theory and topology) but the existing literature does not heavily emphasise the fact that the selection monad is a CPS translation. In particular it has so far gone unnoticed that the selection monad has a call/cc-like operator with interesting similarities and differences to the usual call/cc, which we explore experimentally using Haskell.

Selection functions can be used whenever we find the intermediate result more interesting than the final result. For example a SAT solver computes an assignment to a boolean function, and then its continuation decides whether it is a satisfying assignment, and we find the assignment itself more interesting than the fact that it is or is not satisfying. In game theory we find the move chosen by a player more interesting than the outcome that results from that move. The author and collaborators are developing a theory of games in which selection functions are viewed as generalised notions of rationality, used to model players. By realising that strategic contexts in game theory are examples of continuations we can see that classical game theory narrowly misses being in CPS, and that a small change of viewpoint yields a theory of games that is better behaved, and especially more compositional.

Contents

1   Introduction

A selection function is a type-2 function $\varepsilon : \mathscr{J} X$ where

$$\mathscr{J} X = (X \to R) \to X$$

We can view the input $k : X \to R$ to such a function in several ways: as a generalised predicate, as the context of a decision, or as a continuation. These are emphasised respectively in topology [4], game theory [5] and proof theory [7]. The earliest selection functions considered were computable instances of Hilbert's ε-operator $(X \to \mathbb{B}) \to X$, which witness a computational form of topological compactness. An operation $\otimes : \mathscr{J} X \times \mathscr{J} Y \to \mathscr{J} (X \times Y)$, which is the (left-leaning) monoidal product of the strong monad $\mathscr{J}$, witnesses the fact that the product of two compact spaces is compact. Remarkably this extends to countable products, leading to the derivation of 'seemingly impossible functional programs' [3] that search set-theoretically infinite but topologically compact types such as $\mathbb{N} \to \mathbb{B}$ (the Cantor space) in a finite amount of time.

It was noticed by Paulo Oliva that the product of selection functions is equivalent to Spector's bar recursion [8], a notoriously obscure computational feature used to realise the axiom of countable choice via a double negative translation and Gödel's Dialectica interpretation. Bar recursion is important in the proof mining programme because it can be used to interpret proofs of classical analysis (including differential equations and ergodic theory) [14], but the computer programs arising from such proofs via bar recursion, while provably correct, are not well suited to human understanding.

The next step made by Escardó and Oliva was the connection with game theory, by realising that economic rationality is modelled by the selection function

$$\arg\max : (X \to \mathbb{R}) \to X$$

which finds a point maximising a real-valued function (say, on a finite set X). By generalising selection functions by replacing the booleans with $\mathbb{R}$ or an arbitrary type R, the product of selection functions is seen to correspond to an well-known and intuitive algorithm in game theory known as backward induction [5]. Since proof interpretations using bar recursion can be rewritten using the product of selection functions, we therefore obtain a computational interpretation of proofs in classical analysis that is amenable to human understanding via game theory [16].

The relationship to the continuation monad is immediately obvious: the Hilbert ε-operator is a refined form of the quantifier

$$\exists : (X \to \mathbb{B}) \to \mathbb{B}$$

and the operator $\arg\max$ is a refined form of

$$\max : (X \to \mathbb{R}) \to \mathbb{R}$$

These functions are both computations in the continuation monad

$$\mathscr{K} X = (X \to R) \to R$$

Escardó and Oliva call any function with a type of this form a *quantifier*. Moreover the relationships

$$\exists p = p(\varepsilon p)$$

and

$$\max p = p(\arg \max p)$$

define a monad morphism from the selection to the continuation monad. Thus in proof theory the J-translation (or Peirce translation) [7] can be seen as a refined form of the usual (generalised) double negative translation, and it would be tempting to view the selection monad as a refinement of the continuation monad. However things are not so clear, because call/cc of the selection monad has a more specific type, and behaves differently.

In section 2 we will explain the (quite complex) bind operation of the selection monad using intuition from CPS, which previously has only been derived in a purely formal way. In sections 3 and 4 we will discuss the call/cc-like operator that exists for the selection monad, and see its behaviour experimentally. Finally in section 5 we will informally discuss ongoing work by the author and others on applications of selection functions and CPS in game theory.

2 Two monads for CPS

We begin by recalling the usual intuition for the continuation monad, which can be used to implement (delimited) continuations in a pure language such as Haskell. We view all computation as being done relative to a *continuation*, which takes the return value and chains it to the future of the computation. Thus our computation has the shape[1]

$$\cdots \longrightarrow X \xrightarrow{\quad k \quad} R$$

We will call X the type of *intermediate values*, and R the type of *final values*. The key idea of the continuation monad is that we always work relative to an *unknown* continuation k, although we may fix the type R. (The ability to fix R may have begun as a quirk due to Hindley-Milner typing, see [13], but it is vital to many applications of selection functions.) We allow our functions to have side-effects, modelled by functions $X \to MR$ for a suitable monad M. A computation in the continuation monad is a function with type

$$\mathcal{K}_R^M = (X \to MR) \to MR$$

which computes a final result given a continuation.

---

[1]The diagrams in this paper are not intended to be formal, but rather as a possible aid to intuition, which not every reader will find helpful. 'Plain' arrows are intended to live in the Kleisli category of the base monad M, while arrows with quote marks around the name live in the Kleisli category of either $\mathcal{K}_R^M$ or $\mathscr{J}_R^M$.

To embed a pure value $x : X$ we simply view it as the intermediate value, and apply the continuation to it immediately:

$$\eta_{\mathscr{K}_R^M} x = \lambda k^{X \to MR}.kx$$

For the bind, suppose we have a computation $\varphi : \mathscr{K}_R^M X$ and a family of computations $F : X \to \mathscr{K}_R^M Y$. Equivalently, F is a computation $X \to Y$, which we can run by supplying it with a continuation from Y. We form a computation that has the shape

$$\cdots \xrightarrow{\hspace{2cm}} X \xrightarrow{\text{``}F\text{''}} Y \xrightarrow{\hspace{1cm}k\hspace{1cm}} R$$

The key to understanding the bind of both the continuation and selection monads is that we have two views of a computation like this: we can either view the intermediate result as being at X or Y. The 'external' view of $\varphi \gg\!\!=_{\mathscr{K}_R^M} F : \mathscr{K}_R^M Y$ is that it is a computation with the intermediate result being at Y. Suppose we are running this computation, so we have a continuation $k : Y \to MR$. To find the final result we move to the other view, and run the computation φ by building it a longer continuation $k' : X \to MR$. Given an intermediate value $x : X$ we can now find the final result $k'x$ because we have the continuation k from Y: it is $k'x = Fxk$. Therefore the bind operator for the continuation monad is given by

$$\varphi \gg\!\!=_{\mathscr{K}_R^M} F = \lambda k^{Y \to MR}.\varphi \lambda x^X.Fxk$$

This intuition for the continuation monad transfers directly to the selection monad. Whereas a computation in the continuation monad computes the final result given a continuation, a computation in the selection monad computes the intermediate result instead. Thus the selection monad is

$$\mathscr{J}_R^M X = (X \to MR) \to MX$$

First, suppose we want to embed a pure value $x : X$ as the intermediate result. Given a continuation $k : X \to MR$, we ignore the continuation and simply return x, embedded as a computation in M:

$$\eta_{\mathscr{J}_R^M} x = \lambda k^{X \to MR}.\eta_M x$$

For the bind operator of the selection monad we must again move between the two views of the computation

$$\cdots \xrightarrow{\hspace{2cm}} X \xrightarrow{\text{``}F\text{''}} Y \xrightarrow{\hspace{1cm}k\hspace{1cm}} R$$

Suppose we have a computation $\varepsilon : \mathscr{J}_R^M X$ and a family of computations $F : X \to \mathscr{J}_R^M Y$. To run $\varepsilon \gg\!\!=_{\mathscr{J}_R^M} F : \mathscr{J}_R^M Y$ we are given a continuation $k : Y \to MR$, and we must return an intermediate result at Y. Our first task is to find a way to turn an intermediate result at X into one at Y. Suppose

3

we have an intermediate value $x : X$, so we have a computation $Fx : \mathscr{J}_R^M Y$ with intermediate value at Y. We can now run this computation with our continuation k, which yields an intermediate value at Y. Thus we have a function $f : X \to MY$ given by

$$fx = Fxk$$

Now we can extend our continuation k with f to give a continuation k' from X, namely

$$k'x = fx \gg\!\!=_M k = Fxk \gg\!\!=_M k$$

By running the computation ε with this continuation, we obtain an intermediate result at X. Finally, we must apply the function f again to obtain an intermediate result at Y, which is what we need. Written out explicitly, the operator is

$$\varepsilon \gg\!\!=_{\mathscr{J}_R^M} F = \lambda k^{Y \to MR}.(\varepsilon\lambda x^X.Fxk \gg\!\!=_M k) \gg\!\!=_M \lambda x^X.Fxk$$

Obviously, the sheer complexity of this formula makes reasoning about programs in the selection monad very difficult. In practice, Haskell is a very useful tool for qualitatively understanding the behaviour of such programs.

No published proof of the monad laws for the selection monad exists. The original proof (in the absence of other side-effects) was generated by an equational reasoning tool written by Martin Escardó specifically for this purpose, resulting in several pages of formal manipulations. For an arbitrary side-effects the unit laws were checked by the author with several pages of equational reasoning, including use of functional extensionality (η-expansion). The associative law seems to be impractical to check by hand, but the proof is found by Coq's tactic for intuitionistic logic. However previously the unit and bind operators were considered purely formal objects (derived simply by proving them as theorems of intuitionistic logic), and the intuitions developed in this section suggest for the first time that it might be possible to find a human-readable proof, by reducing to the monad laws for the continuation monad.

3 Call/cc for the selection monad

For building computations in the continuation monad we can use the call-with-current-continuation operator. This has type

$$\mathfrak{cc} : ((X \to \mathscr{K}_R^M Y) \to \mathscr{K}_R^M X) \to \mathscr{K}_R^M X$$

The input Φ to $\mathfrak{cc}$ is called a *continuation handler*, which is a computation that has access to the *current continuation* $\Bbbk : X \to \mathscr{K}_R^M Y$ (we will use the letter k to refer to an actual continuation, and $\Bbbk$ to refer to a continuation reified as a computation in the continuation or selection monad). The current continuation is just a computation $X \to Y$ in the continuation monad, and the purpose of $\mathfrak{cc}\Phi$ is to call $\Phi\Bbbk$ where $\Bbbk$ is bound to the continuation of $\mathfrak{cc}\Phi$. The implementation of $\mathfrak{cc}$ is given by

$$\mathfrak{cc}\Phi = \lambda k_1^{X \to MR}.\Phi(\lambda x^X, k_2^{Y \to MR}.k_1 x)k_1$$

4

Given our description it is quite easy to read this formula. The parameter given to Φ should be the continuation k_1 reified as a function $X \to K_R^M Y$. We have exactly that: given an input, we make the computation which ignores its own continuation and uses k_1 instead.

Put another way, we want to build a computation $\Bbbk$ to which we can apply Φ. Consider the diagram
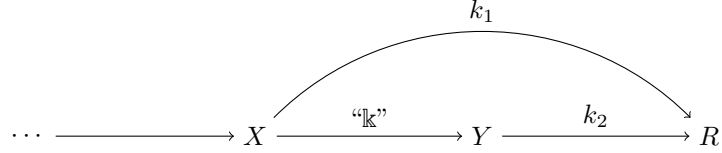
In this diagram, X is the intermediate type at the point in the handler at which the current continuation is invoked, and Y is the intermediate result when the handler ends. We do not have a pure function $X \to Y$, but we can build a CPS computation $\Bbbk : X \to \mathscr{K}_R^M Y$ by short-cutting with the continuation k_1, that is,

$$\Bbbk x = \lambda k_2^{Y \to MR}.k_1 x$$

However, the principal type of the term ∞ is the far more general

$$((A \to B \to C) \to (A \to C) \to D) \to (A \to C) \to D$$

We obtain the specific type $((X \to \mathscr{K}_R^M Y) \to \mathscr{K}_R^M X) \to \mathscr{K}_R^M X$ by setting $A = X$, $B = Y \to MR$ and $C = D = MR$. In order that our computations are in the selection rather than the continuation monad, we instead set $C = MR$ and $D = MX$. This implies that $Y = R$, and so $B = R \to MR$. Thus ∞ has the type

$$((X \to \mathscr{J}_R^M R) \to \mathscr{J}_R^M X) \to \mathscr{J}_R^M X$$

This is the call-with-current-continuation operator for the selection monad. It is equal (as an untyped λ-term) to the ordinary call/cc, and the difference in behaviour comes from the different bind operator with which it is composed.

If we draw a similar diagram we get
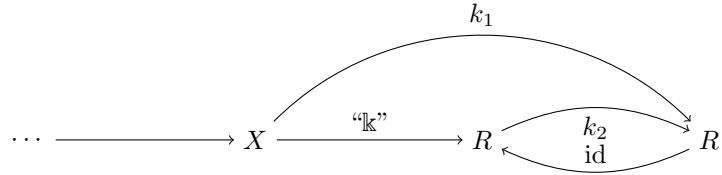
We have the continuation k_1 which gives us a final result given an intermediate result at X, which is also the point at which the continuation is invoked in the handler. The computation $\Bbbk$, given its continuation k_2, should return the intermediate result. However instead we simply return $k_1 x$, which is really the final result, which we arranged to have the same type. That is, we are implicitly

assuming that k_2 is always the trivial continuation, even though it might not be. We could change the type to

$$((X \to \mathscr{K}_R^M R) \to \mathscr{J}_R^M X) \to \mathscr{J}_R^M X$$

to emphasise that $\Bbbk$ returns a final rather than an intermediate result, but that only moves our dishonesty elsewhere because to actually invoke the current continuation in the handler we would need an 'invoke-continuation' function with type $\mathscr{K}_R^M R \to \mathscr{J}_R^M R$, which treats its final result as intermediate.

In the next section we will see experimental results about the operational behaviour of this function, including when the whole computation is run with a nontrivial continuation.

4 Programming with selection call/cc

This section uses the author's implementation of the selection monad transformer from [10], and assumes some familiarity with monad transformers and the syntax of Haskell. As an example we will take the simple CPS Haskell program

```haskell
trace :: (MonadIO m) => String -> m ()
trace = liftIO . putStrLn

foo :: ContT r IO Int
foo = do trace "In foo"
         n <- callCC $ \k -> do trace "In handler"
                                m <- k 0
                                trace "Still in handler"
                                return (m + 1)
         trace "In continuation"
         return (n + 1)
```

When run interactively we obtain

```
ghci> runContT foo return
In foo
In handler
In continuation
1
```

This program demonstrates an important fact about call/cc: once the continuation is run, control is never returned to the handler. Thus while the continuation logically returns a value and we can bind it to a variable, any code after the continuation is called is unreachable. This is in contrast to the selection monad, which we will see next.

The same program written in the selection monad is

```
bar :: SelT Int IO Int
bar = do trace "In bar"
         n <- callCC' $ \k -> do trace "In handler"
                                 m <- k 0
                                 trace "Still in handler"
                                 return (m + 1)
         trace "In continuation"
         return (n + 1)
```

By running this program we can see the execution trace:

```
ghci> runSelT bar return
In bar
In handler
In continuation
Still in handler
In continuation
3
```

As can be seen, there is a coroutine-like dialogue between the handler and the
continuation, with data flowing back and forth.

Like in the continuation monad, the continuation can be called from arbi-
trarily far inside a call stack. In particular we can write a product of selection
functions that has access to its current continuation. For example, we can write
a SAT solver that calls the current continuation with a dummy input once per
iteration:

```
sat :: Int -> SelT Bool IO [Bool]
sat n = do bs <- callCC' $ \k -> sequence $ replicate n $
                 do b <- SelT ($ True)
                    liftIO $ putStr $ "b = " ++ show b ++ ", "
                    k []
                    return b
           trace $ "Continuation called with " ++ show bs
           return bs
```

This program is based on the verbose SAT solver in [10], and is designed to
exhibit as many unexplained patterns as possible. Given a particular formula
like

```
f :: [Bool] -> IO Bool
f bs = return $ bs!!0 && not(bs!!1) && bs!!2
```

we can run it by

```
ghci> runSelT (sat 3) f
b = True, Continuation called with []
b = True, Continuation called with []
b = True, Continuation called with []
```

```
Continuation called with [True,True,True]
b = False, Continuation called with []
Continuation called with [True,True,False]
b = False, Continuation called with []
b = True, Continuation called with []
Continuation called with [True,False,True]
b = True, Continuation called with []
Continuation called with [True,False,True]
b = True, Continuation called with []
b = True, Continuation called with []
b = True, Continuation called with []
Continuation called with [True,True,True]
b = False, Continuation called with []
Continuation called with [True,True,False]
b = False, Continuation called with []
b = True, Continuation called with []
Continuation called with [True,False,True]
b = True, Continuation called with []
Continuation called with [True,False,True]
[True,False,True]
```

(Note that each time the current continuation is called with the empty list it also calls the formula f, but the expressions which index the empty list, which would result in a runtime exception, are not evaluated due to the lazy semantics of Haskell.) It is already an open question how to explain patterns such as $TTT, TTF, TFT, TFT, TTT, TTF, TFT, TFT$ resulting from the operational behaviour of the product of selection functions. The positions of the empty list in this trace only seem to deepen the mystery.

5 Strategic contexts are continuations

The author and collaborators are developing a theory of games based on selection functions, which will be informally described in this section. In a computable game theory, the computations done by players are naturally CPS computations, in the sense that the computation divides into two parts: first the player computes a move, and then the rules of the game (and the other players) use the move to compute the outcome. However the player's computation of a move is done with knowledge of how that move will be used to compute an outcome, which makes it a CPS computation. In classical game theory a player may have a finite set of choices X, and each choice determines a real number $kx : \mathbb{R}$ called a *utility*. A fundamental assumption of classical game theory is that rational players act so as to maximise their utility. We can view the continuation of the player's decision as $k : X \to \mathbb{R}$, and the computation done by the player is $\arg\max : \mathscr{J}_\mathbb{R} X$. One reason that this is interesting is that the product of selection functions computes Nash equilibria of sequential games of

8

perfect information, by implementing an algorithm known as backward induction [6]. Thus what was apparently a fact of applied mathematics, concerning the behaviour of interacting rational agents, in fact arises very naturally in a theoretical setting.

An immediate application of these ideas is that we can easily generalise some concepts in classical game theory to arbitrary selection functions, which allows us to work with players who are not classically rational without having to build a new game theory from scratch. This is applied to 'context-dependent choice' in voting games in [11], where it is shown that the fixpoint operator fix : $\mathscr{J}_X^{\mathcal{P}} X$ models a 'Keynes agent' who would like to vote for the winner of an election, or a so-called Keynes beauty contest. Similarly the operator that selects non-fixpoints models a 'punk' who would like to vote for anybody but the winner, which corresponds to a one-shot minority game [12]. In general, a 'context-dependent agent' may have preferences not just over outcomes, but over the ways in which those outcomes are achieved. This formally gives no new expressive power because the type of outcomes could be extended to include all relevant information, but selection games are more convenient, and in particular seem to be more scalable. To give a silly example, a million dollars earned is not equivalent to a million dollars stolen. To model such 'social concerns' classically, an explicit conversion rate between morality and dollars must be given globally and encoded into the outcome structure of the game. With selection functions this can be done instead on a per-agent basis.

We conclude by giving several research directions currently being explored by the author and several collaborators. In each of these, the slogan 'strategic contexts are continuations' is an important part of the author's intuitive understanding.

- Most importantly, the author has recently developed a graphical calculus for game theory by extending string diagrams for monoidal categories [17]. The semantics of these string diagrams uses CPS very heavily: a diagram generally represents some group of interacting agents, which is equivalent to a 'generalised agent' with preferences over both strategies and continuations from computations of strategies. The categorical composition and tensor product (which are primitive forms of sequential and parallel composition, respectively) are both defined using (delimited) continuation transformers, and cannot be written in a direct style.

- The way in which this graphical calculus differs from the ones used in quantum theory is the way in which backward-causality is treated. Game theory contains a quite restricted form of backward-causality due to agents reasoning about future events. An agent is graphically connected to a relevant future value by a feedback-like operation. Intriguingly there is an extremely close analogy with shift/reset operators [1] here: a decision made by an agent is like 'shift' (it captures a continuation), and the point in the future at which a value is designated as the outcome for an agent is like 'reset' (it delimits a continuation). In the graphical calculus

these also appear strongly analogous to the unit and counit of a compact structure in category theory, which correspond to pair-production and pair-annihilation in quantum mechanics.

- The first two points relate to defining games and solution concepts, but not to computing solutions in general. In this direction, the author has used various monad transformer stacks with the selection monad at the top. For sequential games of perfect information this can be done in generality using the product of selection functions, but to extend to simultaneous and other games must be done on a per-effect basis. The most progress has been made for nondeterministic games, in which agents can make nondeterministic choices between several moves. (True nondeterminism is essentially unknown in economic game theory; for a use in game theory in computer scientist see [15, chapter 9].) The idea is to define a new 'sum of selection functions' operator

$$\oplus : \mathscr{J}_R^M X \times \mathscr{J}_R^M Y \to \mathscr{J}_R^M (X \times Y)$$

which is analogous to the product of selection functions, but for simultaneous games. Games with nondeterministic strategies are noticeably better behaved than either pure or mixed strategies, with solution spaces carrying more structure.

- For infinite games, and games with mixed strategies, things are more interesting and difficult. The author is bringing together many ideas from functional programming, topology and category theory to attack these problems. One starting point is that probability distributions form a monad [9, 2] which carries additional topological structure.

References

[1] Olivier Danvy and Andrzej Filinski. Abstracting control. *Proceedings of the 1990 ACM conference on Lisp and functional programming*, pages 151–160, 1990. doi: 10.1145/91556.91622. 9

[2] Martin Erwig and Steve Kollmansberger. Probabilistic functional programming in Haskell. *Journal of functional programming*, 16(1):21–34, 2006. doi: 10.1017/S0956796805005721. 10

[3] Martin Escardó. Seemingly impossible functional programs, 2007. URL http://math.andrej.com/2007/09/28/seemingly-impossible-functional-programs/. 1

[4] Martin Escardó. Exhaustible sets in higher-type computation. *Logical methods in computer science*, 4(3:3):1–37, 2008. doi: 10.2168/lmcs-4(3:3)2008. 1

[5] Martin Escardó and Paulo Oliva. Sequential games and optimal strategies. *Proc R Soc A*, 467:1519–1545, 2011. doi: 10.1098/rspa.2010.0471. 1

[6] Martin Escardó and Paulo Oliva. Computing Nash equilibria of unbounded games. *Proceedings of the Turing centenary conference*, 2012. 9

[7] Martin Escardó and Paulo Oliva. The Pierce translation. *Annals of pure and applied logic*, 163(6):681–692, 2012. 1, 2

[8] Martin Escardó and Paulo Oliva. Bar recursion and products of selection functions. To appear, 2014. 1

[9] Michèlle Giry. A categorical approach to probability theory. *Categorical aspects of topology and analysis*, pages 68–85, 1982. doi: 10.1007/BFb0092872. 10

[10] Jules Hedges. Monad transformers for backtracking search. In Paul Levy and Neel Krishnaswami, editors, *Proceedings of the 5th workshop on mathematically structured functional programming*, volume 153 of *Electronic proceedings in theoretical computer science*, pages 31–50. Open Publishing Association, 2014. doi: 10.4204/EPTCS.153.3. 6, 7

[11] Jules Hedges, Paulo Oliva, Evguenia Winschel, Viktor Winschel, and Philipp Zahn. A higher-order framework for decision problems and games. Submitted, 2014. 9

[12] Willemien Kets. Learning with fixed rules: the minority game. *Journal of economic surveys*, 26(5):865–878, 2011. doi: 10.1111/j.1467-6419.2011.00686.x. 9

[13] Oleg Kiselyov. Undelimited continuations are co-values rather than functions. URL http://okmij.org/ftp/continuations/undelimited.html. 2

[14] Ulrich Kohlenbach. *Applied proof theory: proof interpretations and their use in mathematics*. Springer, 2008. 1

[15] Steven M. LaValle. *Planning algorithms*. Cambridge University Press, 2006. 10

[16] Paulo Oliva and Thomas Powell. A game-theoretic computational interpretation of proofs in classical analysis. To appear, 2012. 1

[17] Peter Selinger. A survey of graphical languages for monoidal categories. In Bob Coecke, editor, *New structures for physics*, pages 289–355. Springer, 2011. 9