Call-by-Push-Value with Algebraic Effects and Handlers

Stavros Alexandros Dimakos[1]

Supervisors: Casper Bach Poulsen[1]**, Jaro Reinders**[1]

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Stavros Alexandros Dimakos
Final project course: CSE3000 Research Project
Thesis committee: Casper Bach Poulsen, Jaro Reinders, Annibale Panichella

Abstract

Addressing the challenge of reasoning about programs across different evaluation strategies has long been a concern in functional programming. Levy's introduction of the call-by-push-value (CBPV) calculus represents a significant step forward in tackling this. His paradigm provided a more powerful approach that can encapsulate both call-by-value and call-by-name that was even later extended to include call-by-need. In this paper we present the development of an interface that integrates the theory of CBPV with algebraic effects and handlers. We demonstrate how this technique enables the definition and execution of programs, highlighting its capability to defer computations across different evaluation strategies and define operations in a modular fashion. We then define and prove a set of laws that can be used with our interface to reason about programs under varying evaluation regimes. This approach not only enhances the flexibility and modularity of language and library implementation but also allows for direct reasoning about these implementations, beyond the meta-level abstraction.

1 Introduction

As programming advances and software systems grow in complexity, the costs associated with maintaining and creating new systems also rise. Consequently, there is a growing demand for innovative programming methodologies to support developers throughout the development process.

Among the methods developed to address this challenge, the use of algebraic effects and handlers [11] in functional programming presents a promising approach for constructing and reasoning about sophisticated software abstractions. By using these tools, developers can implement interfaces that facilitate software development and define and validate specific laws that govern the behavior of these interfaces. This allows users to focus on functionality without concerning themselves with implementation details.

Wouter Swierstra's "Data Types à la Carte" [13] shows how to implement a modular system for defining and combining algebraic effects. This methodology allowed for the customization and extension of effects in a manner that had not been previously possible, setting the stage for subsequent innovations. It provided a solid foundation for modular functional programming and offered strategies for managing complex type systems and operations in Haskell. Following this, additional research was published to popularize the use of handlers in interpreting the effects of algebraic computations [2, 3].

Another popular topic in functional programming research is call-by-push-value (CBPV), proposed by Paul Blain Levy [4, 7]. CBPV aims to unify two predominant evaluation strategies, call-by-name (CBN) and call-by-value (CBV). Levy's work detailed the operational semantics of a CBPV language, illustrating how it serves as a foundational paradigm subsuming both CBV and CBN.

This paper aims to combine these two concepts, exploring **how algebraic effects and handlers can be used to construct an interface capable of achieving call-by-push-value in Haskell**.

Despite theoretical advancements, the practical adoption of CBPV in mainstream programming languages remains elusive. As of this writing, no well-known languages support this model, highlighting a gap between theoretical innovation and practical application. This paper aims to provide an interface for translating languages between evaluation strategies using the theory described in Levy's paper.

To achieve this, we will explore the following research questions:

RQ1 What are the benefits of call-by-push-value?

RQ2 How can we create an interface that can be used to translate programs to different evaluation strategies?

RQ3 How can we define the behavior of the implementation using mathematical laws?

RQ4 How can the implementation be proven correct with respect to its laws?

RQ5 How closely does the implementation and its laws align with existing theory found in the literature?

This implementation will be defined as an interface that other programs can use to change their evaluation strategies. As stated in research questions 3 and 4, we aim to describe this interface in terms of mathematical laws that it can be proved against. This not only validates the correctness of the interface but also empowers reasoning about programs written in different evaluation strategies.

Contributions We make the following contributions:

- We define an interface for postponing computations written in terms of algebraic effects (Section 3.1).
- We present a method for translating a lambda-calculus source language into either call-by-name or call-by-need evaluation (Section 3.2).
- We demonstrate how to use the implementation for running programs, handling effectful operations, and understanding how different translations impact the side effects of these operations (Section 3.3).
- We show how to modularly extend our implementation with new effects and operations, ensuring they are appropriately postponed when necessary (Section 3.4).
- We prove a set of laws describing the implementation for both call-by-name and call-by-need to reinforce reasoning about programs written against it (Section 4).

2 Methodology and Background

As mentioned in Section 1 call-by-push-value (CBPV) was introduced by Paul Blain Levy, who provided a formal framework that unifies and generalizes the operational semantics of both call-by-value and call-by-name. Levy's work [4, 6] demonstrated how CBPV could model various effects in programming languages, such as state, exceptions, and continuations, in a more granular and structured way. His paradigm aims to bridge the gap between call-by-name and call-by-value by treating values and computations as distinct entities, thereby simplifying the semantic understanding of different programming effects and enhancing language interoperability.

Building on Levy's foundational work, Downen et al. [8] extend the CBPV paradigm to include lazy evaluation. They also integrate an effect system to utilize known potential effects of expressions to improve reasoning about how programs behave under different evaluation orders.

In order to build on top of their work we will use similar techniques as covered in [12, 13] focusing around the use of the free monad. The free monad is a concept from category theory that allows us to represent algebraic effects as abstract syntax trees with nodes for operations (Op) and values (Pure):

```
data Free f a
       = Pure a
       | Op (f (Free f a))
```

Using the free monad, we can represent each operation of the language in a more modular way. Moreover, it provides a clearer way to handle effects and reason about them, which is crucial as the implementation of the interface needs to support both effectful and pure operations. For instance, consider the State functor which consists of the following two stateful operations:

```
data State s k
  = Put s k
  | Get (s -> k)
  deriving Functor
```

We can consider this signature functor as an interface that specifies Put and Get operations. With this interface, Free (State s) s represents the type for sequences of Put and Get operations that ultimately produce an s value. An example of such a program is one that doubles an integer state and returns the old value:

```
double :: Free (State Int) Int
double = Op (Get (\s -> Op (Put (s * 2) (Pure s))))
```

There are numerous techniques to improve the readability and writability of our code, including combining different interfaces, using smart constructors and using signature subtyping to name some. However, as large part of this paper is mainly focused on the denotation of syntax rather the implementation of effect interfaces we skip this part and focus on how to run our programs. If you want to learn more however, the following papers explore these techniques in depth: [2, 3, 12]

In order to use these interfaces we need to implement a structure for handling their effects. The main benefit of this approach is that we can define specific handlers completely independently from each other. Then by applying these in a nested fashion we run programs with different sets of interfaces. Essentially, a handler comprises two functions: one responsible for defining how effects are managed and another for returning the computed value. For example, consider the handler for our State operations described above:

```
hState :: Functor g => Handler_ (State s) a s g
     (a, s)
hState = Handler_
  { ret_ = \x s -> pure (x, s)
  , hdlr_ = \x s -> case x of
      Put s' k -> k s'
      Get k -> k s s }
```

In our return function, we return a pair with the current state and a value. For the handle case, we have two alternatives based on the Put and Get operations. In case of a Get, we pass the current state to the continuation while in case of a Put we pass a new value to the continuations [12].

3 Implementation

The core of our call-by-push-value (CBPV) interface implementation revolves around four primary functions: *thunk*, *force*, *lam*, and *apply*. In this section, we first explore how *thunk* and *force* are implemented to defer the evaluation of operations. Following this, we examine the *lam* and *apply* functions, detailing how they facilitate the translation of programs between different evaluation strategies. Finally, we show how we can use this interface to run programs and extend it with new effects.

3.1 Thunk

In call-by-push-value, terms can either be computations or values. This is based around the notion that "a value is, a computation does". However, in evaluation strategies such as call-by-name and call-by-need we often need to freeze computations and bind them to values. In order to achieve this while maintaining the idea that a value is, we use *thunks*. A *thunk* allows us to capture a computation as a value and delay its execution until it is needed. Consider the following example program, that uses a basic implementation of thunk:

```
print "Hello1";
let x be thunk (
    print "Hello2";
    return True;
   );
print "Hello3";
print x;
apply x to
   lambda y. (
      print "Hello4";
      y;
   );
```

We provide a step-by-step explanation of the execution. The program executes the commands in sequence. First, it prints "Hello1". Then, it binds x to a thunk. If the term *thunk* were omitted, the expression would be evaluated immediately and the resulting value would be bound to x.

Next, "Hello3" is printed, followed by a print call to x. This call triggers the execution of the thunked computation, where "Hello2" is printed and True is returned. As a result, "True" is also printed.

Finally, we end with an application of x to a lambda expression that takes y as a parameter. This will first bind the thunked computation to y, then it will print “Hello4” and return y. When returning y, the computation will run again, printing “Hello2” once more and then returning True.

In summary the program outputs as follows:

```
Hello1
Hello3
Hello2
True
Hello4
Hello2
```

and finally returns the value True.

In our implementation, we use two functions, *thunk* and *force*. The *thunk* function encapsulates computations in a data structure until they are needed, and the *force* function triggers immediate computation. The first challenge is that *forcing* a computation in a call-by-name manner (above example) is different from call-by-need, as the latter requires preventing reevaluation of the computation. To address this, we implement three versions of these functions, one for each evaluation strategy.

For a call-by-value strategy, a *thunk* operation is typically unnecessary. However, since all subsequent functions will use *thunks* and we want to enable straightforward transitions between evaluation strategies, we create a *thunk* structure for call-by-value. This structure simply runs the computations and captures the resulting value in a *thunk*. The *force* function then unwraps the *thunked* value and wraps it in the pure constructor:

```
newtype CBVal a = CBVal { unCBVal :: a }

thunkCBVal :: Functor f =>
    Free f a -> Free f (CBVal a)
thunkCBVal m = fmap CBVal m

forceCBVal :: CBVal a -> Free f a
forceCBVal (CBVal a) = Pure a
```

The call-by-name case is also straightforward due to our use of the free monad. The free monad represents our program and its continuations. By wrapping this in a new *thunk* data structure with the Pure constructor, the operations will not be executed until *force* is called. When *force* is called, it simply unwraps the computation to be executed:

```
newtype CBName f a = CBName (Free f a)

thunkCBName :: Free f a -> Free f (CBName f a)
thunkCBName m = Pure (CBName m)

forceCBName :: CBName f a -> Free f a
forceCBName (CBName m) = m
```

Finally for call-by-need we make use of the *State* functor we introduced in Section 2. Combining this with its handler, we can incorporate *memoization* into our interface, thus preventing the reevaluation of any *thunked* computation. Each call-by-need *thunk* holds an integer value that points to a store and the corresponding computation. The store is essentially a list consisting of a data structure called a pack, which either contains the computed value of the expression or Nothing. We maintain a list of packs within the state functor, updating it whenever a new computation is *thunked* or *forced*.

In the *thunk* function we begin by getting the current store and update it with a new entry that includes an additional *pack* for the newly created *thunk*. As the computation has not been run yet, the cell will hold Nothing. Then in *forcing* a computation we again retrieve the store and pattern match on the cell corresponding to the *thunk*. If the result is Nothing we run the computation, update the store and return the calculated value. If the result corresponds to a value, we can just return the value without any further actions.

```
newtype CBNeed f a = CBNeed (Int, Free f a)

data Pack = forall v. Pack (Maybe v)

thunkCBNeed :: (Functor f, (State [Pack]) < f) =>
    Free f a -> Free f (CBNeed f a)
thunkCBNeed m = do
  s <- get -- Smart constructor for injecting Get
      operations in our program
  put (s ++ [Pack Nothing])
  return (CBNeed (length s, m))

forceCBNeed :: (Functor f, State [Pack] < f) =>
    CBNeed f a -> Free f a
forceCBNeed (CBNeed (n, m)) = do
  s <- get
  case s !! n of
    Pack (Just v) -> return (unsafeCoerce v)
    Pack Nothing -> do
      v <- m -- Run computation
      -- Update store
      put (updateList n (Pack (Just v)) s)
      return v
```

In the above code snippet we use a *get* and a *put* function. These are called smart constructors for the State operations, which inject get and put operations respectively into our program.

3.2 Translation of types and terms

Using the implementation of *thunks* detailed in Section 3.1, we are now one step closer to being able to translate between evaluation strategies. At this stage we need a structure to represent functions. We create a new data structure “Fun” which takes a value of type t_1 and produces the continuation of our program with resulting type t_2. To interact with these types we use two additional functions, *lam* and *apply*:

```
newtype Fun f t1 t2 = Fun {app :: t1 -> Free f t2}

lam :: (a -> Free f b) -> Free f (Fun f a b)
lam f = Pure (Fun f)

apply :: Fun f a b -> a -> Free f b
apply = app
```

With these steps completed we can now translate the types and terms of one evaluation strategy to another. To showcase this capability and implement the translation mechanisms we create an example language with some operations. This includes boolean values, variables, strings, let expressions, if expressions, lambdas, applications, injections, pattern matching and an effectful operation, print:

```
data Lang
  = Var String
  | Let String Lang Lang
  | Truel
  | Falsel
  | Stringl String
  | If Lang Lang Lang
  | Print Lang Lang
  | Lambda String Lang
  | App Lang Lang
  | Inl Lang
  | Inr Lang
  | Pm Lang Lang Lang
```

We define a *denote* function that maps this syntax onto effectful (monadic [9]) operations. Through this we can later use algebraic effects and handlers to handle all our operations in isolation [14]. First we will show the translation for call-by-name of each expression. We create a new data structure to represent values and one for operations:

```
data Value f
  = VTrue
  | VFalse
  | VString String
  | VFun (Free f (Fun f (CBName f (Value f))
      (Value f)))
  | VInl (CBName f (Value f))
  | VInr (CBName f (Value f))

data Operation k = forall s. PrintOp s k
```

Then we use an environment to emulate memory for storing variables:

```
data Env f s = Env
    {cells :: [(String, CBNeed f s)]}

lookupEnv ::
    String -> Env f s -> Maybe (CBNeed f s)
lookupEnv x (Env env) = lookup x env

extendEnv ::
    String -> CBNeed f s -> Env f s -> Env f s
extendEnv x val (Env env) = Env ((x, val) : env)
```

Using these, the *denote* function takes an expression written in terms of the Lang data structure and produces a Free monad. This monad contains operations that will result in a value once they have been handled.

```
-- Type signature of denote
denote :: (Functor f, Operation < f) =>
    Env f (Value f) -> Lang -> Free f (Value f)
```

First for booleans and strings we directly produce their corresponding Values.

```
denote _ Truel = Pure VTrue
denote _ Falsel = Pure VFalse
denote _ (Stringl s) = Pure (VString s)
```

Accessing a variable is the same as *forcing* its computation.

```
denote env (Var x) = case (lookupEnv x env) of
  Just v -> forceCBName v
  Nothing -> undefined -- Variable name not found
```

Let expressions create new variables by *thunking* their computations and adding them to the environment. After the computations are *thunked* we denote the rest of the program with the updated environment.

```
denote env (Let x e1 e2) = do
  v <- thunkCBName (denote env e1)
  denote (extendEnv x v env) e2
```

Lambda expressions produce a function value that can be used in function application. When applying a value to function we first *thunk* the value to be applied and bind it to the variable name. After adding the variable to the environment we *denote* the body of the function.

```
denote env (Lambda x body) =
  Pure $ VFun (lam (\v ->
    denote (extendEnv x v env) body))
denote env (App e1 e2) = do
  v1 <- thunkCBName (denote env e1)
  v2 <- denoteC env e2
  case v2 of
    (VFun (Pure f)) -> apply f v1
    _ -> undefined
```

We use injections to enable pattern matching in our language. A standalone injection will *thunk* a computation and produce an injection *value*. When used with a pattern match expression that includes left and right injections of lambda expressions, it will pass the *thunked* computation into the corresponding lambda expression.

```
denote env (Inl e) = do
  v <- thunkCBName (denote env e)
  return (VInl v)
denote env (Inr e) = do
  v <- thunkCBName (denote env e)
  return (VInr v)
denote env (Pm e (Inl (Lambda x body1)) (Inr
     (Lambda y body2))) = do
  v <- denote env e
  case v of
    VInl val -> denote (extendEnv x val env) body1
    VInr val -> denote (extendEnv y val env) body2
    _ -> undefined
```

Finally, effectful operations like print can be represented directly using the free monad. We can use the Print operation that will later need a Print handler to execute the expression:

```
denote env (Print e1 e2) = do
  v1 <- denote env e1
  printf v1 -- Smart constructor for Print
  denote env e2
```

With each case handled, we have effectively translated any language using the above operations into a call-by-name evaluation. This approach allows us to extend our language with any necessary operations. Additionally, by utilizing the Free monad, users of the interface can execute their programs in a completely modular fashion. They can select any set of individual handlers for their desired effects and operations. This not only provides a powerful way to execute and reason about programs, but it also allows for flexibility in evaluation strategies. By simply replacing the *thunk* and *force* functions described in Section 3.1, we can adopt any of the three evaluation strategies (call-by-name, call-by-value, or call-by-need).

3.3 Executing programs

Using this syntax with the *denote* function, we can create and execute programs. In the language described in Section 3.2, the only operation included is *Print*. Let's refer back to the example showcased in Section 3.1 that uses the print operation and will yield different results depending on the evaluation strategy. To achieve this, we first need to create a handler for our *Print* operation. As mentioned in Section 2 a handler comprises two functions: one responsible for defining how effects are managed and another for returning the computed value. In our scenario, we define *hPrint*, which concatenates the value to be printed within a string composed of all previously printed values and returns the value when no further operations remain.

```
hPrint :: Functor g =>
    Handler_ Operation a String g (a, String)
hPrint = Handler_
  { ret_ = \x s -> pure (x, s)
  , hdlr_ = \x s1 -> case x of
      PrintOp s2 k -> k (s1 ++ " " ++ (show s2))}
```

We describe the example program in terms of our Lang Expressions:

```
Print (StringI "Hello1") (
    Let "x" (
        Print (StringI "Hello2") TrueI
    ) (
        Print (StringI "Hello3") (
            Print (Var "x") (
                App (Var "x") (
                    Lambda "y" (
                        Print (StringI "Hello4") (
                            Var "y"
                        )
                    )
                )
            )
        )
    )
)
```

Running this with our print handler will yield different results for each evaluation strategy. For call-by-name, the results will match those explained in Section 3.1.

For call-by-value, the computation will be executed directly, assigning True to x. As a result, it prints the "Hello" messages in order.

Finally, for call-by-need, the results will be the same as call-by-name until we reach the function application. By that point, the variable x has already been evaluated once. Therefore, when binding the value of x to y, only True will be bound, resulting in "Hello2" being printed only once.

Table 1: Print results for each evaluation strategy

Call-by-Value	Call-by-Name	Call-by-Need
Hello1	Hello1	Hello1
Hello2	Hello3	Hello3
Hello3	Hello2	Hello2
True	True	True
Hello4	Hello4	Hello4
	Hello2	

3.4 Language extension

This interface provides a powerful method for reasoning about languages with different evaluation strategies. But what if we need to reason about other effects not included in the current implementation? The strength of our approach lies in its ability to extend the set of operations included, without interfering with existing effects. We showcase this ability through an example: Suppose we want to add a new error effect representing the abrupt termination of our programs. To accomplish this, we start by adding a new constructor in *Lang* and we define a new signature functor for it:

```
data Lang
  = Var String
  | ...
  | Error String -- New Lang constructor

data Error k = ErrorOp String
  deriving Functor
```

Then we add a new case in our *denote* function for the Error constructor:

```
-- We add Error to the signature of denote
denote :: (Functor f, Operation < f, Error < f) =>
    Env f (Value f) -> Lang -> Free f (Value f)
denote _ TrueI = Pure VTrue
...
denote _ (Error e) = errf e -- We use errf to
     inject an Error operation into our program
```

These are the only updates needed for our existing implementation; the rest is completely independent. We can now implement our handler to behave as desired and nest it with our other handlers when running our program. Simple as that, we have successfully incorporated errors into our language, which properly work with different evaluation strategies, allowing us to write programs such as:

```
exampleError :: Lang
exampleError = Let "x" (Error "Error!") TrueI)
```

Running this will terminate early using our call-by-value *thunk*, as it would evaluate the error immediately. Conversely, it would run until the end using our call-by-name or call-by-need *thunk*, as *x* is never evaluated, meaning the error is never thrown.

4 Laws

To describe the behavior of our implementation, we use mathematical laws from existing research in the field [5, 8]. Proving these laws is crucial for numerous reasons, one of which is ensuring the correctness of our interface. Verification through these laws confirms that our implementation behaves as expected under all defined conditions, accurately translating theoretical foundations into practical applications.

Moreover, these help with facilitating reproducibility. Clear proofs of the underlying laws allow others to reproduce our work more accurately. This transparency helps other researchers understand, replicate, and build upon our implementation, contributing to the overall advancement of the field (Section 5).

Finally, having these proofs allows us to reason about programs written against our implementation with greater confidence and clarity. By relying on established mathematical laws, we can predict how these programs will behave in various scenarios, identify potential issues, and ensure that they adhere to the intended design principles. Consequently, other developers can interact with it without the need to get into the specifics of the implementation.

We begin with the most fundamental law of the interface:

$$\text{thunk}\ m \gg\!= \text{force} \quad \equiv \quad m$$

The law states that binding the result of any of the three thunk functions in our implementation to its corresponding force function should yield the original computation represented by the monadic value m. This demonstrates that the combined effect of wrapping and immediately unwrapping a computation does not alter the original value. The proofs for call-by-name and call-by-need are displayed bellow:

Call by name:

```
thunkCBName m >>= forceCBName
 =
Pure (CBName m) >>= forceCBName
 =
fold forceCBName Op (Pure (CBName m))
 =
forceCBName (CBName m)
 =
m
```

Call by need:

```
thunkCBNeed m >>= forceCBNeed
 =
(do
  s ← get
  put (s ++ [Pack Nothing])
  return (CBNeed (length s, m))) >>= forceCBNeed
 =
(do
  put (s_0 ++ [Pack Nothing])
  return (CBNeed (length s_0, m))) >>= forceCBNeed
 =
Pure (CBNeed (length s_0, m)) >>= forceCBNeed
 =
forceCBNeed (CBNeed (length s_0, m))
 =
do
  s ← get
  case s !! (length s_0) of
    Pack (Just v)  →  return (unsafeCoerce v)
    Pack Nothing →  do
      v ← m
      put (updateList n (Pack (Just v)) s)
      return v
 =
case (s_0 ++ [Pack Nothing]) !! (length s_0) of
  Pack (Just v)  →  return (unsafeCoerce v)
  Pack Nothing →  do
    v ← m
    put (updateList n (Pack (Just v)) s)
    return v
 =
v ← m
put (updateList n (Pack (Just v)) s)
return v
```

In the call-by-need scenario, the final step involves executing the computation *m*, updating the store to avoid reevaluation, and returning the result of the computation *m*. Even though this is not exactly equivalent to just executing the computation it will behave identically if we only evaluate the same expression once. We can then presume that upon future evaluations, the value will be directly returned. Using this sequential reasoning technique showcased in the above example we prove the following laws:

$$
\begin{array}{rcll}
\text{thunk}\ m \gg\!= \text{force} & \equiv & m & (1)\\
\text{Let}\ x\ v\ m & \equiv & \text{App}\ v\ (\text{Lam}\ x\ m) & (2)\\
m & \equiv & \text{Let}\ x\ m\ (\text{Var}\ x) & (3)\\
\text{Pm}\ (\text{Inl}\ v)\ (\text{Inl}\ (\text{Lam}\ x\ m_1))\ (\text{Inr}\ (\text{Lam}\ x\ m_2)) & \equiv & \text{App}\ v\ (\text{Lam}\ x\ m_1) & (4)\\
\text{Pm}\ (\text{Inr}\ v)\ (\text{Inl}\ (\text{Lam}\ x\ m_1))\ (\text{Inr}\ (\text{Lam}\ x\ m_2)) & \equiv & \text{App}\ v\ (\text{Lam}\ x\ m_2) & (5)
\end{array}
$$

1. The first law describes that forcing a thunked computation *m* is equivalent to directly running the computation *m*.
2. The second law describes that binding a variable *x* to a value *v* and then running a computation *m* is equivalent to replacing all instances of *x* with *v* within *m* and running the computation *m*.
3. The third law describes that running a computation *m* is equivalent to binding the computation to a variable and calling the variable.
4. The fourth law describes that pattern matching a left injection holding a value *v* onto left and right injections each holding a lambda function, is equivalent to applying the *v* to the left injections' lambda.
5. The fifth law describes that pattern matching a right injection holding a value *v* onto left and right injections each holding a lambda function, is equivalent to applying the *v* to the right injections' lambda.

In addition to these, we have proven two sequencing laws that enable us to reason about the order of operations in our computations. The first law states:

$$\text{Let } y\ (\text{Let } x\ m\ n)\ p \quad \equiv \quad \text{Let } x\ m\ (\text{Let } y\ n\ p)$$

This law ensures that the nested sequencing of computations maintains consistent behavior, allowing us to reframe the order of "let" expressions without changing the program's semantics. The second law further enhances our understanding of computation sequencing over functions. It states:

$$\text{Let } x\ m\ (\text{Lambda } y\ n) \quad \equiv \quad \text{Lambda } y\ (\text{Let } x\ m\ n)$$

This law emphasizes the interchangeability of "let" expressions and lambda abstractions. It asserts that the order of binding and abstraction can be freely interchanged without altering the program's behavior. Evaluating these two expressions results in merely a different order of storing in our environment. However, this alteration does not affect the evaluation of expressions since the environment is accessed by the names of the variables bound to the expressions. This means that as long as $x \neq y$ the law holds.

It should be noted that this is not a completely exhaustive set of laws, but it covers most of the behaviours the implementation should adhere to. A complete set of these proofs for both the call-by-name and the call-by-need cases can be found in our Github repository[1].

5 Responsible Research

In this section, we outline the measures taken to ensure the reproducibility and transparency of this research. All code related to the implementation of the Call-By-Push-Value (CBPV) interface is available on GitHub[1]. This repository includes the version of the Free monad used for the implementation, the State functor and its handlers, as well as the thunk, force, lam, and apply functions described in Section 3.

Moreover, Section 2 and Section 3 provide a clear, step-by-step description of the procedures used, ensuring that others can replicate the experiments. Additionally, we use the most popular Haskell compiler (GHC) that offers the most features. For all code in this research we used version 9.4.8[2].

To validate the theoretical aspects of the CBPV interface and ensure its correctness, mathematical proofs were constructed. Due to space constraints, not all proofs could be included in Section 4, but all of them are mentioned. For this reason, all proofs are included in our GitHub[1] repository, ensuring they are readily accessible for verification by others.

By following responsible research practices such as transparency, reproducibility, and avoiding divisive methods, this study aims to contribute to the collective progress of the field. These principles not only uphold the integrity of the research but also inspire additional exploration and refinement of our proposed approach.

6 Related Work

Algebraic Effects and Handlers Algebraic effects [10] have become a crucial area of research in programming languages. Moggi's introduction of monads [9] provided a foundation for structuring computational effects. Algebraic effects built on top of their work to introduce a more structured and composable [1] approach. Recent advancements of effect handlers [11], offer a flexible way to define and control effects, enabling practical language and library implementations.

Call-by-Push-Value Levy's call-by-push-value [4, 6] calculus introduced a new framework for expressing both call-by-name and call-by-value. This has been the inspiration for the interface defined in this paper. We have demonstrated how to use the theory that Levy proposed in order to translate the types and terms of our language. However, it would be inaccurate to describe our interface as a replica of CBPV, as we do not adhere to the exact same distinctions between computations and values that CBPV employs.

Instead, we have focused on using algebraic effects and handlers to modularly isolate our operations. We also used the concept of *thunking* to achieve the postponement of effects that is found in CBPV. This approach has shown similar benefits to those of the CBPV language. Similarly, to CBPV our interface not only allows us to optimize our programs but to also reason about more properties of them and their side effects. In our model we can choose how we want to handle each effect and operation individually which can then be combined by nesting their handlers.

Extended Call-by-Push-Value In this research, we have also touched upon extended CBPV [8]. This notion introduces lazy evaluation to CBPV by adding a new construct, M *need* x.N, which evaluates x only the first time it is used and directly uses the produced value for subsequent uses. In our implementation, we used the *thunkCBNeed* and *forceCBNeed* functions to achieve similar results. Our *denote* function applies this directly without the option to evaluate different parts of the same program differently, something that is possible in

[1]Github repository: `https://github.com/AlexDimakos/CBPV_using_algebraic_effects_and_handlers`

[2]GHC version: `https://www.haskell.org/ghc/download_ghc_9_4_8.html`

extended CBPV. However, it is also possible in our implementation by working with these functions directly.

7 Discussion

This paper introduces an interface for translating lambda-calculus source languages into various evaluation strategies based on call-by-push-value (CBPV) theory. Our implementation (Section 3) demonstrates how to delay the execution of certain computations and effects, enabling the translation of entire expressions into different evaluation strategies.

Moreover, we can now compare programs written using different evaluation strategies. Traditionally, this comparison would require translating the programs into the same language to ensure a uniform evaluation strategy. Even then, tracing the execution of call-by-need or call-by-name strategies can be challenging and often leads to unexpected behavior. Our interface, however, offers a straightforward solution that is easier to reason about, further supported through the laws we have proven (Section 4).

In essence, this interface offers a powerful framework for composing programs from isolated components with interchangeable evaluation strategies. However, our approach comes with certain limitations. One is that it is not always feasible to create modular components for operations described in terms of algebraic effects. While the approach holds promise in terms of scalability—allowing for the seamless integration of new operations without disrupting existing ones—the actual implementation of these modular components can present challenges and complexities. Specifically, algebraic effects can interact in complex ways that are difficult to modularize, requiring global knowledge about their composition which undermines modularity. Additionally, effect handlers need to be correctly scoped to manage where and how effects are handled, and ensuring proper scoping in a modular system can be tricky, potentially leading to unintended behavior or performance issues.

Furthermore, we highlighted that by utilizing our *thunk* and *force* functions directly, comparable outcomes to those outlined in existing literature [6, 8] can be attained. However, this approach necessitates programmers to have a deeper understanding of the implementation, thereby decreasing their capacity to reason about programs developed with it.

8 Conclusions and Future Work

This paper aims to develop an interface that integrates algebraic effects and handlers [11] with call-by-push-value [4, 6] (CBPV). Implemented in Haskell, the interface was constructed by first developing a *thunking* mechanism to delay computations. This was afterwards used to translate types and terms of a lambda calculus-based language into different evaluation strategies. Then we demonstrated how this interface can be used to write and run programs. Finally, we defined and proved a set of laws that our implementation adheres to in order to help with reasoning about programs.

With these steps completed, we have successfully addressed all of our research questions. We demonstrated the benefits of call-by-push-value, created an interface capable of translating programs to different evaluation strategies, defined the behavior of the implementation through mathematical laws, proved the correctness of the implementation with respect to these laws, and aligned our work with existing theoretical frameworks.

The groundwork laid by this study opens up numerous avenues for future research. Our primary focus was to establish a foundational framework for understanding how different evaluation orders impact program behavior and side effects. Consequently, many opportunities remain for further exploration.

One intriguing area for future research is the application of extended CBPV, as demonstrated in [8]. This includes the proof of equivalence of call-by-name and call-by-need evaluation strategies when the only effect in the source language is nontermination. Although this example showcases the proof capabilities of their language, it would be worthwhile to investigate whether this equivalence holds within our interface as well. Additionally, if this equivalence is confirmed, further research could explore what other equivalences can be proven within this framework.

Another promising area for exploration is the optimization of performance within the CBPV framework. While lazy evaluation often performs best for many programs, it comes with higher overhead and memory usage due to the need to store computed values. By using an interface that simplifies the translation of programs into different evaluation strategies, we can analyze which programs perform best under which evaluation strategies trivially. It could be valuable to investigate whether there is a way to easily identify the optimal evaluation strategy for a given program.

Such investigations could enhance our understanding of evaluation strategies and their interactions with different computational effects, potentially leading to new theoretical insights and practical applications.

Acknowledgements

We would like to thank our supervisor and responsible professor, for their guidance and support throughout this research. Their feedback and engagement were invaluable, and their willingness to spend time answering questions and discuss various aspects of our research greatly helped in the completion of this work.

References

[1] Martin Hyland, Gordon Plotkin, and John Power. Combining effects: Sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006.

[2] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’13, page 145–158, New York, NY, USA, 2013. Association for Computing Machinery.

[3] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *ACM SIGPLAN Symposium/Workshop on Haskell*, 2013.

[4] Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 228–243, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[5] Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis (Semantics Structures in Computation, V. 2).* Kluwer Academic Publishers, USA, 2004.

[6] Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006.

[7] Paul Blain Levy. Call-by-push-value. *ACM SIGLOG News*, 9(2):7–29, may 2022.

[8] Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In Luís Caires, editor, *Programming Languages and Systems*, pages 235–262, Cham, 2019. Springer International Publishing.

[9] Eugenio Moggi. An abstract view of programming languages. *Computer Languages*, 1989.

[10] Gordon Plotkin and John Power. Notions of computation determine monads. In *Foundations of Software Science and Computation Structures*, pages 342–356. Springer, 2002.

[11] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[12] Casper Bach Poulsen. Algebraic effects and handlers in haskell, 2023.

[13] WOUTER SWIERSTRA. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[14] Birthe van den Berg, Tom Schrijvers, Casper Bach-Poulsen, and Nicolas Wu. Latent effects for reusable language components. 08 2021.

A AI Usage During the Research

In this paper, we utilized AI tools such as ChatGPT and Gemini to enhance the clarity and coherence of our writing. By providing prompts such as "Make this well written: ...", we received suggestions from these tools, which we then adapted to maintain our own voice and style. Additionally, Claude AI assisted in understanding complex concepts relevant to our research, thereby enabling deeper analysis of theoretical frameworks and methodologies. It's important to note that while AI played a significant role in refining our writing and enhancing conceptual understanding, the coding aspect of our research was conducted without AI intervention, following traditional programming practices.