Summary

Compilers are used countless times every day as part of software development. They serve to fix many errors before the code is even run on the machine, such as with type checking, but as with all software programs, there is the concern of introducing bugs. Depending on where the bug is introduced it can be benign, but it can also lead to the flawed code generation. That is, even following the semantics of the programming languages and correctly implementing a well known algorithm might still result in bugs, since the code generated by the compiler, the code that actually runs on the machine, might not reflect the higher-level implementation because of miscompilation of the compiler. This is the motivation behind the verification of compilers, i.e. ensuring that the compiler is faithful to the semantics of the language, such that the generated code is accurate. A well known verified compiler project is CompCert, which is working towards a formally verified C compiler. A large part of the challenges they are facing is of course the size of the C language, but also the lack of a formal semantics. Resulting in a lot of work to bridge the gap between the implied semantics and the verification effort.

What can be done for making proof easier, is first and foremost the presence of a sound formal semantics. Additionally abstract machines can be a tool to prove compiler correctness.

This report seeks to develop and implement an abstract machine for the call-by-push-value language described by Levy [16], and prove its correctness by showing that it is faithful to the semantics described in [16].

The report starts by introducing call-by-push-value and its syntax and semantics, then proceeds to present the notion of binding in lambda calculus, and how these tie into the call-by-push-value interpreter. Next is the presentation of functional abstract machines, where the well known SECD and Krivine machines are presented, culminating in the presentation of the call-by-push-value abstract machine. Finally, correctness of abstract machines is discussed and the call-by-push-value abstract machine is proven correct using bisimulation. This report demonstrates the concerns involved in implementing and proving correctness of an abstract machine.

Implementing and proving correctness of a call-by-push-value abstract machine

Dion Sean Hansen

Department of Computer Science, Aalborg University

June 14, 2024

Abstract machines are a cheap method of implementation, that offers better performance compared to interpreters, while being a tool to prove compiler correctness. This report presents the implementation and proof of correctness for a call-by-push-value abstract machine. The implemented abstract machine is based on the SECD machine, and was supported by the implementation of a call-by-push-value interpreter. We provide a proof of correctness using a bisimulation of the call-by-push-value semantics and the abstract machine's transition rules and configuration.

Contents

1 Introduction

Software is at the heart of modern society and one of the essential tools that enables that is compilers. They *compile* a source language, typically a high-level programming language, into a target language, which can then be further compiled, run on an abstract machine (AM) or run natively on the machine, if the target language is machine code. Compilers are one of the tools that enable developers to write in languages closer to human intuition and create ambitious programs and systems, however, not all compilers are created equal.

Say you implement a well known algorithm, such as a sorting algorithm, compile it and run it, but the result is wrong. That is, the code generated by the compilers was erroneous, and the behaviour and meaning of the program has been changed during compilation. A compiler that produces minor or rare errors, can be considered a nuisance in simple systems, but for safety-critical systems it is a major concern, which is the motivation behind proving compilers correct. That is to say, ensuring that compilers do not change the meaning of a program during compilation. A well known example of this is CompCert, which is working towards a formally verified C compiler [5]. Where formally verified means that it is exempt from *miscompilation* issues, which they use the Coq theorem prover to prove. Compcert mentions several empirical studies that have found issues in popular production compilers, such as [9, 26].

The challenges that CompCert face are in large part the size of the C language, but also that initial C implementations were not based on a formal semantics. CompCert therefore has to bridge the gap by reverse engineering existing implementations.

So what can be done to enable easier verification of a compiler, is to base the language on a sound formal semantics. This was the case with Standard ML,

which sought to standardise the language [19]. The efforts of standardisation meant that implementations such as ML Kit could be a direct translation of the 1990 definition [19, 20]. Another example is CakeML, which is an implementation of a significant subset of Standard ML, that has been formally verified using the HOL4 theorem prover [4]. Having a well established semantics and a implementation that closely mirrors the semantics makes it much simpler to prove correctness of the implementation.

The two main ways of implementing a language, is to either implement the semantics, i.e. interpretation, or to compile the language to an intermediate form that can be evaluated, such as with an abstract machine. Both ways would be syntax-directed in that an implementation of the semantics would evaluate a term by constructing derivation trees according to the reduction rules, and that an AM would evaluate the instructions as they appear on the control stack. The key difference is that the transition rules for AMs do not have conditions, and simply evaluate the top-most instruction and move the stacks around. AMs therefore offer a means of evaluation on a lower level of abstraction, but above that of real hardware machines, as discussed by Diehl et al. in [6]. Furthermore, as presented by [14], abstract machines offer many benefits, such as better performance than that of interpreters, functions as a cheap implementation device, and as a tool to prove compiler correctness.

Diehl et al. present an extensive bibliography of AMs and an insight into the research field of AMs [6]. AMs serve as an intermediate stage of compilation in order to bridge the gap of high level programming languages and real hardware machines. By utilising an intermediate stage, it becomes simpler to reason from higher-level language to the intermediate representation, and also to reason about the representation with respect to the underlying computer architecture. AMs are a formal description or implementation capable of executing a specific set of instructions, where those instructions are tailored to the specific operational needs of the source language, or class of languages.

Diehl et al. also highlights that AMs are appealing with respect to proving correctness of code generation, along with various analyses and transformations [6]. Lastly, by leveraging a common abstract machine, or platform, the reach of the source language is significantly wider. A well known example of utilising abstract machines, and the concept of a common compilation target, is Microsoft's .NET platform, which ties together multiple languages, such as C#, F# and Visual Basic, using the intermediate language *Common Intermediate Language* (CIL) [7]. CIL enables optimisations to it, and its abstract machine, to be exploited by all the source languages. Furthermore, by consolidating all the target languages into one language and abstract machine, it becomes simpler to achieve and maintain cross-platform support since it is only necessary to support CIL on the respective platforms.

In this report I wish to develop, and prove correctness of, an abstract machine that uses call-by-push-value (CBPV), a parameter mechanism first described by Paul Levy [16]. CBPV is an active area of research [10, 23, 17, 8] and a subsuming paradigm of call-by-value and call-by-name, which is to say, that the two evaluation strategies can be expressed with simply CBPV. This means CBPV can be used to implement both languages such as Standard ML, which is

call-by-value, and Haskell, which is call-by-name. Lastly, by using the semantics provided by Levy [16] to direct the implementation, it should enable a simpler proof of correctness.

Section 2 will present call-by-push-value (CBPV), notions of binding and the implementation of a translator and interpreter for CBPV. This will be followed by Section 3, which will present well known AMs and the definition of the developed CBPV AM. Section 4 will present the proof of correctness for the CBPV AM, and finally Section 5 will conclude the report.

2 Call-by-push-value and its implementation

This section will present call-by-push-value (CBPV), its syntax and semantics, as well as translations of CBPV based on Levy [16]. Whereafter it will discuss the different notions of binding variables and the considerations required in developing an implementation of the CBPV language, and finally present parts of the implementation.

2.1 Introducing call-by-push-value

Call-by-push-value is an idealised calculus first discovered by Paul Levy [16]. It is a variant of simply typed lambda calculus, which incorporates computational effects, and therefore combines functional and imperative programming. A noteworthy aspect of call-by-push-value (CBPV) is that it subsumes call-by-value (CBV) and call-by-name (CBN). CBV and CBN can be translated to and from CBPV, and the transformation preserves the semantics, which suggests that, from a semantic viewpoint, CBV and CBN are sub-systems of CBPV [16].

CBPV has many variants, and since this report wishes to develop an implementation of CBPV it would be natural to consider the classic variant as presented by Levy [16]. The classic CBPV is typed, but since types are not central to the challenges tackled by this report we consider an untyped setting. Furthermore, only a core subset of CBPV will be considered and implemented.

2.2 Call-by-push-value syntax and semantics

The syntax can be seen in Definition 1 and 2, which present the set of values **Val** and the set of computation **Comp** respectively.

Definition 1 (Values)**.** *The formation rules for values in call-by-push-value is as follows*

$$V \in \textbf{\textit{Val}} ::= x \mid \mathit{thunk}\ M$$

Definition 2 (Computations)**.** *The formation rules for computations in call-by-push-value is as follows*

$$M, N \in \textbf{\textit{Comp}} ::= \lambda x.M \mid MV \mid \mathit{let}\ x = V\ \mathit{in}\ M \mid \mathit{force}\ V \mid \mathit{return}\ V \mid M\ \mathit{to}\ x\ \mathit{in}\ N$$

A value is:

- a variable x
- or a thunk M, where a *thunk* is a suspended computation, such as a λ-abstraction $\lambda x.M$.

Computations are::

- λ-abstraction $\lambda x.M$, and application MV
- a let $x = V$ in M, x is bound to the value V in M, i.e. $M[V/x]$
- a force V, which runs a suspended computation, i.e. a thunk
- M to x in N, where M is first evaluated to a computation of the form return V, and then V is bound to x in N, as so $N[V/x]$.

Definition 3 presents the set of *terminals* **Term**, where **Term** $\subset$ **Comp**, and Table 1 contains the semantics of CBPV.

Definition 3 (Terminals)**.**

$$T \in \textbf{\textit{Term}} ::= \lambda x.M \mid \textit{return } V$$

(CBPV-LET)	$\dfrac{}{\text{let } x = V \text{ in } M \to M[V/x]}$
(CBPV-TO)	$\dfrac{M \to M'}{M \text{ to } x \text{ in } N \to M' \text{ to } x \text{ in } N}$
(CBPV-RETURN)	$\dfrac{}{(\text{return } V) \text{ to } x \text{ in } M \to M[V/x]}$
(CBPV-FORCE)	$\dfrac{}{\text{force thunk } M \to M}$
(CBPV-APP)	$\dfrac{M \to M'}{MV \to M'V}$
(CBPV-BETA)	$\dfrac{}{(\lambda x.M)V \to M[V/x]}$

Table 1: Small-step semantics of CBPV [16]

Tables 2 and 3 present the translations for CBV and CBN to CBPV, and vice versa. The use of these translations can be seen in Section 3.5, with Examples 8 and 9.

CBV term	CBPV computation
x	return x
λx.M	return thunk λx.M
MN	M to f in (N to g in (force f)g)

Table 2: Decomposition of CBV into CBPV [16]

CBN term	CBPV computation
x	force x
$\lambda x.M$	$\lambda x.M$
MN	M(thunk N)

Table 3: Decomposition of CBN into CBPV [16]

2.3 Notions of binding

In the usual semantics of λ-calculus, bindings are explained as substitutions. In this section we will present the concepts that make this possible. The following section is in large part based on Barendregt [1], except modifications made with respect to CBPV.

There are two primary binding models in λ-calculus, and those are *syntactic binding* (substitution) and *semantic binding* (environments). Both models define how values are bound to *free* variables. In syntactic binding, the variable in a β-reduction is immediately substituted for the value, whereas in semantic binding the binding is stored in the environment. Then, when a variable is reached during reduction, the value of the variable is looked up in the environment and substituted for the binding. This section will present syntactic binding, and Section 3.2 will present semantic binding.

Essential concepts for both binding models is that of *free* and *bound* variables. Those are presented in Definitions 4 and 5, which inductively define the set of free and bound variables in a term M. Using `FV`() and `BV`(), you can derive `AV`(), as seen in Definition 6, which defines the set of all variables in a term M. Furthermore, Definition 7, presents the notion of open and closed lambda terms.

Definition 4 (Free and bound variables)**.** *The set of free variables for a term*

M, denoted $\texttt{FV}(M)$ is defined inductively as follows.

$$\begin{aligned}
\texttt{FV}(x) &= \{x\} \\
\texttt{FV}(MV) &= \texttt{FV}(M) \cup \texttt{FV}(V) \\
\texttt{FV}(\lambda x.M) &= \texttt{FV}(M) - \{x\} \\
\texttt{FV}(\mathit{let}\ x = V\ \mathit{in}\ M) &= (\texttt{FV}(V) \cup \texttt{FV}(M)) - \{x\} \\
\texttt{FV}(M\ \mathit{to}\ x\ \mathit{in}\ N) &= (\texttt{FV}(M) \cup \texttt{FV}(N)) - \{x\} \\
\texttt{FV}(\mathit{return}\ V) &= \texttt{FV}(V) \\
\texttt{FV}(\mathit{force}\ V) &= \texttt{FV}(V) \\
\texttt{FV}(\mathit{thunk}\ M) &= \texttt{FV}(M)
\end{aligned}$$

Definition 5 (Bound variables)**.** *The set of bound variables for a term M, denoted $\texttt{BV}(M)$ is defined inductively as follows.*

$$\begin{aligned}
\texttt{BV}(x) &= \{\}; \\
\texttt{BV}(MV) &= \{\}; \\
\texttt{BV}(\lambda x.M) &= \{x\}; \\
\texttt{BV}(\mathit{let}\ x = V\ \mathit{in}\ M) &= \{x\}; \\
\texttt{BV}(M\ \mathit{to}\ x\ \mathit{in}\ N) &= \{x\}; \\
\texttt{BV}(\mathit{return}\ V) &= \{\}; \\
\texttt{BV}(\mathit{force}\ V) &= \{\}; \\
\texttt{BV}(\mathit{thunk}\ M) &= \{\}.
\end{aligned}$$

Definition 6 (All variables)**.** *The set of all variables for a term M, denoted $\texttt{AV}(M)$ is defined as follows.*

$$\texttt{AV}(M) = \texttt{FV}(M) \cup \texttt{BV}(M).$$

Definition 7 (Open and closed terms)**.** *M is a closed λ-term if $\texttt{FV}(M) = \emptyset$.*

Definition 8, presents how free variables are substituted for their binding. It is worth noting the need for a *fresh* variable y', in the cases that $y \in N$, to ensure that the substitutions remain capture-avoiding, since the substitutions could otherwise change the behaviour of the term, as seen in Example 1. Additionally, since circular definitions are not allowed, y cannot be a free variable of V in a term let $y = V$ in M.

Definition 8 (Substitution in CBPV)**.** *The result of substituting N for the free occurrences of x in M, denoted $M[N/x]$. In case of name collisions a fresh*

variable y' is introduced. Substitution is defined as follows

$$
\begin{aligned}
x[N/x] &\equiv N;\\
y[N/x] &\equiv y, \textit{if } x \not\equiv y;\\
(MV)[N/x] &\equiv (M[N/x])(V[N/x]);\\
(\lambda y.M)[N/x] &\equiv \lambda y.(M[N/x]), \textit{if } y \notin N;\\
(\lambda y.M)[N/x] &\equiv \lambda y'.((M[y'/y])[N/x]), \textit{if } y \in N;\\
(\textit{let } y = V \textit{ in } M)[N/x] &\equiv (\textit{let } y = (V[N/x]) \textit{ in } (M[N/x])), \textit{if } y \notin N;\\
(\textit{let } y = V \textit{ in } M)[N/x] &\equiv (\textit{let } y' = (V[N/x]) \textit{ in } ((M[y'/y])[N/x])), \textit{if } y \in N;\\
(M_1 \textit{ to } y \textit{ in } M_2)[N/x] &\equiv (M_1[N/x]) \textit{ to } x \textit{ in } (M_2[N/x]), \textit{if } y \notin N;\\
(M_1 \textit{ to } y \textit{ in } M_2)[N/x] &\equiv (M_1[N/x]) \textit{ to } x \textit{ in } ((M_2[y'/y])[N/x]), \textit{if } y \in N;\\
(\textit{return } V)[N/x] &\equiv \textit{return } (V[N/x]);\\
(\textit{force } V)[N/x] &\equiv \textit{force } (V[N/x]);\\
(\textit{thunk } M)[N/x] &\equiv \textit{thunk } (M[N/x]).
\end{aligned}
$$

Example 1 (Capture-avoiding substitution)**.** *The following substitution is not capture-avoiding, since behaviour is changed as a result of substitution.*

$$
\begin{aligned}
&(\lambda x.y)[x/y] \rightarrow\\
&(\lambda x.y[x/y]) \rightarrow\\
&(\lambda x.x)
\end{aligned}
$$

The behaviour is changed from the function $\lambda x.y$ to the identity function $\lambda x.x$.

To avoid capturing variables, we need to ensure that there are no name collisions during substitution. That is, in a substitution

$$(\lambda x.M)[N/y]$$

we have that if $x \neq y$ and x is not a free variable of N, then the substitution can be said to be capture-avoiding.

The renaming of bound variables and their free occurrences, as seen in Definition 8 for the binding constructs, is known as α-conversion, and can be seen in Definition 9. An example of the use of α-conversion to solve a name collision, can be seen in Example 2. α-conversion also serves as a means to assert equivalence between two terms, which can be seen in Example 3.

Definition 9 (α-conversion)**.** *The renaming of bound variables without modifying behaviour, such that they are different from free ones, is denoted by $=_\alpha$, and defined as follows*

$$\lambda x.M =_\alpha \lambda y.M[y/x], \textit{ provided that } y \textit{ does not occur in } M.$$

Example 2 (Name collision and α-conversion)**.** *In the term below, it is unclear what the application of z will substitute. The inner abstraction λx is shadowing the bound variable x, which is bound from the outer abstraction. Thus it is unclear, which abstraction the variable is bound to.*

$$(\lambda x.(\lambda x.x)y)z$$

Here, α-conversion can be used to rename the outer lambda. Thereby making it clear, which variable will be substituted in the application and dispel the ambiguity. An α-conversion could be as follows

$$(\lambda w.(\lambda x.x)y)z \rightarrow$$
$$(\lambda x.x)y) \rightarrow$$
$$y$$

Example 3 (Equivalence and α-conversion)**.** *Due to the compatibility rules (presented by Barendregt [1]), we can replace terms and sub-terms by equivalent terms in any term context. That is, say we have the following term*

$$\lambda x.x(xx)x$$

If we have, $(\lambda y.yy)x = xx$, then we can replace the sub-term xx and have the following term and equivalence.

$$(\lambda x.x((\lambda y.yy)x)x = \lambda x.x(xx)x$$

Likewise with α-conversion, two terms differing only in the names of bound variables are said to be α-equivalent. Such as with

$$(\lambda x.x)z = z = (\lambda y.y)z$$

In the presentation of lambda calculus in [1], Barendregt states a convention, that has since been referred to as *Barendregt's convention*, which can be seen in Convention 1. In simple terms, the convention states that, in a lambda term no two variables can share the same name, which can be understood to mean that variable names are implicitly α-converted to avoid variable capturing and name collisions. Nonetheless, an implementation would have to perform these conversions explicitly, and the interpreter presented in Section 2.4 does this.

Convention 1 (Barendregt's convention)**.** *If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables [1].*

However, there is a way to make Barendregt's convention hold, thereby simplifying the required bookkeeping, and that is to use De Bruijn notation, which can be seen in Definition 10 and Example 4. De Bruijn notation is a well known way of avoiding name variable capturing, collisions, and thereby eliminate the need for α-conversion.

Definition 10 (De Bruijn notation)**.** *When using De Bruijn notation, every variable is replaced by a number, or index, denoting how much further up in its syntax tree the binding construct occurs, counting from the inside out [3].*

Example 4 (Transformation to De Bruijn notation)**.** *As seen in Definition 10, De Bruijn notation is a method of representing variables in a way that avoids names and instead uses numerical indices. Transforming the following term to De Bruijn notation is as follows*

$$\lambda x.\lambda y.x\ y \rightarrow \lambda.\lambda.\#1\ \#0$$

The variable name x has been replaced by $\#1$, as there is one abstraction between the binding occurrence and the free occurrence, whereas y has been replaced by $\#0$.

2.4 Implementation of the interpreters and translator

This section will present the interpreter for call-by-push-value (CBPV) and a translator to, and from, CBPV. For the purpose of aiding the development of the interpreter for CBPV, interpreters for call-by-value (CBV) and call-by-name (CBN) were implemented as well, where the interpreters implement the definitions of CBV and CBN seen in Definition 11 and Definition 12 respectively. Similarly, the translator made it possible to test translated terms from CBV and CBN, and assert that the interpreter for CBPV preserved the behaviour as expected. The three interpreters have in common, that they are implementations of the semantics, and therefore compute terms by constructing derivation trees according to the semantics. Furthermore, they all employ syntactic binding and use α-conversion to avoid variable capturing.

Definition 11 (CBV interpreter lambda calculus)**.** *The formation rules for untyped lambda calculus are as follows*

$$M ::= x \mid \lambda x.M \mid MN$$

Values have the following form

$$V ::= x \mid \lambda x.M$$

$$\text{(CBV-1)} \qquad \frac{M \to M'}{MN \to M'N}$$

$$\text{(CBV-2)} \qquad \frac{N \to N'}{(\lambda x.M)N \to (\lambda x.M)N'}$$

$$\text{(CBV-3)} \qquad \frac{}{(\lambda x.M)V \to M[V/x]}$$

Table 4: Call-by-value semantics

Definition 12 (CBN interpreter lambda calculus)**.** *The formation rules for untyped lambda calculus are as follows*

$$M ::= x \mid \lambda x.M \mid MN$$

The goal of implementing the CBPV interpreter is for it to serve as a basis for the later design and implementation of an abstract machine for CBPV, which

$$\text{(CBN-1)} \qquad \frac{M \to M'}{MN \to M'N}$$

$$\text{(CBN-2)} \qquad \frac{}{(\lambda x.M)N \to M[N/x]}$$

Table 5: Call-by-name semantics

is introduced in Section 3.5. The focus will be on implementing the syntax and semantics, and not error handling of malformed terms etc. The interpreters and translator were implemented in Haskell.

This section will first present the data structures used in representing lambda terms. Following it will be a presentation of translator and the CBPV interpreter. Lastly, the implementation of topics discussed in Section 2.3 will be presented, namely the implementation of `FV()` and `AV()`, substitution, and α-conversion.

Listing 1 presents the data structure that represents terms in the implementation. Listing 2 presents the implementation for translation between CBV and CBN to CBPV, and vice versa, as seen in Tables 2 and 3. Listing 3 presents the CBPV interpreter, which demonstrates the implementation of the semantics seen in Table 1.

```
type Name = String

data Term = Let Name Term Term |
            To Name Term Term |
            Lam Name Term |
            App Term Term |
            Return Term |
            Force Term |
            Thunk Term |
            Var Name
```

Listing 1: CBPV Syntax

```
cbvCBPV :: Term -> Term
cbvCBPV t@(Var _)    = Return t
cbvCBPV (Lam x t)    = Return (Thunk (Lam x (cbvCBPV t)))
cbvCBPV (App t1 t2) = To f t1' (To g t2' app)
    where f = newvar $ Set.fromList [t1', t2']
          g = newvar $ Set.insert (Var f) $ Set.fromList [t1', t2']
          t1' = cbvCBPV t1
          t2' = cbvCBPV t2
          app = App (Force $ Var f) (Var g)

cbpvCBV :: Term -> Term
cbpvCBV (Return t@(Var _)) = t
cbpvCBV (Return (Thunk (Lam x t))) = Lam x (cbpvCBV t)
```

```
cbpvCBV (To f t1 (To g t2 app)) = case app of
    App (Force (Var f)) (Var g) -> App (cbpvCBV t1) (cbpvCBV t2)
    _ -> error (...)

cbnCBPV :: Term -> Term
cbnCBPV t@(Var _) = Force t
cbnCBPV (Lam x t) = Lam x (cbnCBPV t)
cbnCBPV (App t1 t2) = App (cbnCBPV t1) (Thunk (cbnCBPV t2))

cbpvCBN :: Term -> Term
cbpvCBN (Force t@(Var _)) = t
cbpvCBN (Lam x t) = Lam x (cbpvCBN t)
cbpvCBN (App t1 (Thunk t2)) = App (cbpvCBN t1) (cbpvCBN t2)
```

Listing 2: Implementation of CBN-CBPV translation

```
reduce :: Term -> Term
reduce t@(Let x t1 t2)
    | value t1                  = sub' x t1 t2
    | otherwise                 = error (...)
reduce t@(To x (Return t1) t2)
    | value t1                  = sub' x t1 t2
    | otherwise                 = error (...)
reduce (To x t1 t2)             = To x (reduce t1) t2
reduce (Force (Thunk t))        = reduce t
reduce t@(App (Lam x t1) t2)
    | value t2                  = sub' x t2 t1
    | otherwise                 = error (...)
reduce (App t1 t2)              = App (reduce t1) t2
```

Listing 3: Implementation of CBPV semantics

As seen in Definition 8, substitution in binders is conditional to prevent capturing of variables. This can be seen in Listing 4, where `freeIn` is used to check if the bound variable y is free in the substitution, if it is, `newvar` is used to produce an unused variable name z, and α-conversion is then performing by substituting the variable name z. Listing 5 and 6 are implementations of the functions `FV()` and `AV()`, and Listing 7 presents the function `newvar`, which produces unused variable names.

```
sub' :: Name -> Term -> Term -> Term
sub' x s t@(Lam y t1)
    | not $ freeIn x t  = t
    | freeIn y s        = Lam z $ sub' x s $ sub' y (Var z) t1
    | otherwise         = Lam y $ sub' x s t1
    where z = newvar $ Set.singleton t

...

.
.
```

.

```
sub' x s t                = sub x s t
```

Listing 4: Substitution for nested binders

```
free :: Term -> Set.Set Name
free (Var x)        = Set.singleton x
free (App t1 t2)    = Set.union (free t1) (free t2)
free (Lam x t)      = Set.delete x $ free t
free (Let x t1 t2) = Set.delete x $ Set.union (free t1) (free t2)
free (To x t1 t2)  = Set.delete x $ Set.union (free t1) (free t2)
free (Return t)     = free t
free (Force t)      = free t
free (Thunk t)      = free t
```

Listing 5: Implementation of FV(M)

```
vars :: Term -> Set.Set Name
vars (Var x)        = Set.singleton x
vars (App t1 t2)    = Set.union (vars t1) (vars t2)
vars (Lam x t)      = Set.insert x $ vars t
vars (Let x t1 t2) = Set.insert x $ Set.union (vars t1) (vars t2)
vars (To x t1 t2)  = Set.insert x $ Set.union (vars t1) (vars t2)
vars (Return t)     = vars t
vars (Force t)      = vars t
vars (Thunk t)      = vars t
```

Listing 6: Implementation of AV(M)

```
newvar :: Set.Set Term -> Name
newvar ts = head $ dropWhile (`Set.member` vs) xs
    where vs = varss ts
          xs = iterate ('x':) "x"
```

Listing 7: Function for finding unused variable name

The implementation presents a compelling case for the use of De Bruijn notation, as that would simplify aspects of the substitution, which is why De Bruijn notation is often preferred for implementations.

3 Implementing an abstract machine for call-by-push-value

This section will present well known abstract machines (AM), namely the SECD and Krivine machines, and it will present the abstract machine for the CBPV language, the CBPV AM. The presentation of the SECD and Krivine machines is based on Xavier Leroy's presentation in [15].

The section will first present the structure of a functional AM, followed by presentations of the SECD machine, the Krivine machine, and lastly the CBPV AM.

3.1 Implementation of the abstract machines

The transition rules of the SECD, Krivine, and CBPV AM machines, have been the basis for their respective implementation in Haskell. The implementations are a straightforward representation of the respective rules.

The computation examples of the AMs, seen in Examples 6-9, have been created using the implementations and a pretty printer for LaTeX.

3.2 Functional abstract machines

Common for all the abstract machines presented in this report is that they use semantic binding, which was briefly mentioned in Section 2.3, as well as De Bruijn notation, which was presented in Section 2.3, in Definition 10 and Example 4. Example 5 presents how substitution is performed in semantic binding.

Example 5 (Substitution using semantic binding and De Bruijn notation)**.** *Variable bindings are stored in the environment and substituted when needed. The environment is typically implemented as a stack, such that look-up can be performed according to the De Bruijn index, and the stack grows as each binding β-reduction pushes a new value to the stack, as so*

$$(\lambda.\#0)(\lambda.\#0\ \#0)[\ ] \rightarrow_{\beta} \#0[(\lambda.\#0\ \#0)] \rightarrow_{lookup} (\lambda.\#0\ \#0)[(\lambda.\#0\ \#0)]$$

Definition 13, presents a general formalisation of a functional abstract machine.

Definition 13 (Abstract machine)**.**

An abstract machine is a 4-tuple $(\mathcal{C}, \mathcal{E}, \mathcal{S}, \rightarrow)$, where

$C \in \mathcal{C}$ *is the set of the possible control instructions*
$E \in \mathcal{E}$ *is the set of the possible variable bindings*
$S \in \mathcal{S}$ *is the set of the possible evaluations*
$\langle C, E, S\rangle \rightarrow \langle C', E', S'\rangle$

C, E, and S are stacks, where C holds the control instructions, E is the environment holding variable bindings, and lastly S holds the intermediate evaluations. If xs is a stack and x is an element, then $x :: xs$ and $x : xs$ denote that the element x is placed on top of the stack xs.

Variable bindings are stored in E, such that look-up of variables is denoted as $E(i)$, where i is the De Bruijn index of the variable in question, and so $E(i)$ is the i-th entry in the environment E.

An initial configuration for an abstract machine is of the form $\langle C, \epsilon, \epsilon \rangle$*, where* $C \neq \epsilon$*.*

It is necessary to be able to compile terms into instructions for the machine, hence we need a definition for compiling terms of the source language, lambda calculus, into the target language, instructions of the respective abstract machines.

3.3 The SECD machine

The SECD machine is a highly influential abstract machine, which was designed by Peter Landin [13] and evaluates call-by-value lambda calculus. The lambda calculus recognised by the presented machine can be seen in Definition 14.

Definition 14 (SECD lambda calculus)**.** *The formation rules for untyped lambda calculus are as follows*

$$M ::= x \mid \lambda x.M \mid MN \mid \mathit{let}\ x = V\ \mathit{in}\ N \mid n \mid M + N \mid M - N$$

Values have the following form

$$V ::= x \mid \lambda x.M \mid n$$

$$\text{(CBV-1)} \qquad \frac{M \to M'}{MN \to M'N}$$

$$\text{(CBV-2)} \qquad \frac{N \to N'}{(\lambda x.M)N \to (\lambda x.M)N'}$$

$$\text{(CBV-3)} \qquad \frac{}{(\lambda x.M)V \to M[V/x]}$$

$$\text{(CBV-4)} \qquad \frac{}{\mathit{let}\ x = V\ \mathit{in}\ N \to N[V/x]}$$

Table 6: Call-by-value semantics

The machine gets its name from the configuration, which uses four stacks: an evaluation stack S, an environment E, a control C, and a dump D. The modern variant of SECD uses De Bruijn notation to look into the environment, which enables the removal of the dump D. The formal introduction of the modern SECD machine can be seen in Definition 15.

Definition 15 (Modern SECD machine)**.**

A configuration of the SECD machine is of the form $\langle C, E, S \rangle$*, where*

- *C is the remaining control instructions*
- *E is the environment of variable bindings*
- *S is the stack of values*

An initial configuration for the SECD machine is $\langle C, \epsilon, \epsilon \rangle$, where $C \neq \epsilon$, as seen in Definition 13, and the final configuration is $\langle \epsilon, \epsilon, v :: S \rangle$, where $v \in V$.

The formation rules of the control instructions for the SECD machine are as follows

$$
\begin{aligned}
C &::= I :: C \mid \epsilon \\
I &::= \mathsf{access}(i) \mid \mathsf{closure}(C) \mid \mathsf{return} \mid \mathsf{apply} \\
&\quad \mid \mathsf{let} \mid \mathsf{endlet} \mid \mathsf{const}(n) \mid \mathsf{add} \mid \mathsf{sub}
\end{aligned}
$$

The instructions describe the following behaviour

- *access(i), push the i-th field of the environment*
- *closure(C), push the closure with the current environment*
- *return, terminate current function, and jump back to caller*
- *apply, pop function closure and argument, and perform application*
- *let, pop value and add it to environment*
- *endlet, discard first entry in environment*
- *const(n), push integer n on the stack*
- *add, pop two integers, and push their sum*
- *sub, pop two integers, and push their difference*

The transition rules of the SECD machine can be seen in Table 7. The transition rules are determined by the instruction on top of the control stack C. Here we have that there are only values placed in the evaluation stack S and in the environment E. Thus the arguments for application are always values, as expected for call-by-value.

As stated in Section 3.2, we need a definition for compiling terms of the source language to the target language. Definition 16, presents the rules for compilation from the lambda calculus, seen in Definition 14, to the SECD instructions, seen in Definition 15. Lastly Example 6, presents the steps involved with making use of the abstract machine, compiling from terms to instructions and finally computing the compiled instructions using the machine.

Definition 16 (Compilation from lambda calculus to SECD instructions)**.** *The compilation rules for lambda calculus to SECD instructions are*

(SECD-1) $\langle \mathsf{access}(i) :: C, E, S\rangle \to \langle C, E, E(i) :: S\rangle$

(SECD-2) $\langle \mathsf{closure}(C') :: C, E, S\rangle \to \langle C, E, \mathsf{clo}(C', E) :: S\rangle$

(SECD-3) $\langle \mathsf{return} :: C, E, v :: \mathsf{clo}(C', E') :: S\rangle \to \langle C', E', v :: S\rangle$

(SECD-4) $\langle \mathsf{apply} :: C, E, v :: \mathsf{clo}(C', E') :: S\rangle \to \langle C', v :: E', \mathsf{clo}(C, E) :: S\rangle$

(SECD-5) $\langle \mathsf{let} :: C, E, v :: S\rangle \to \langle C, v :: E, S\rangle$

(SECD-6) $\langle \mathsf{endlet} :: C, v :: E, S\rangle \to \langle C, E, S\rangle$

(SECD-7) $\langle \mathsf{const}(n) :: C, E, S\rangle \to \langle C, E, n :: S\rangle$

(SECD-8) $\langle \mathsf{add} :: C, E, n :: m :: S\rangle \to \langle C, E, (m+n) :: S\rangle$

(SECD-9) $\langle \mathsf{sub} :: C, E, n :: m :: S\rangle \to \langle C, E, (m-n) :: S\rangle$

Table 7: Transition rules of SECD

$$
\begin{aligned}
[\![\#i]\!] &= \mathit{access}(i) \\
[\![\lambda.M]\!] &= \mathit{closure}([\![M]\!] : \mathit{return}) \\
[\![\mathit{let}\ V\ \mathit{in}\ N]\!] &= [\![M]\!] : \mathit{let} : [\![N]\!] : \mathit{endlet} \\
[\![MN]\!] &= [\![M]\!] : [\![N]\!] : \mathit{apply} \\
[\![n]\!] &= \mathit{const}(n) \\
[\![M+N]\!] &= [\![M]\!] : [\![N]\!] : \mathit{add} \\
[\![M-N]\!] &= [\![M]\!] : [\![N]\!] : \mathit{sub}
\end{aligned}
$$

Example 6 (Compiling and computing call-by-value lambda calculus with the SECD machine)**.** *The following term has been chosen to demonstrate the call-by-value nature of the SECD machine.*

$$(\lambda x.x)((\lambda x.x)(\lambda x.x))$$

Before it can be compiled using the rules from Definition 16, it needs to be transformed into De Bruijn notation, as seen below

$$(\lambda.\#0)((\lambda.\#0)(\lambda.\#0))$$

Then using the compilation rules for the SECD machine, it results in the following instructions

$$M, M, M, \mathsf{apply}, \mathsf{apply}$$

Where M is the identity function, $[\![\lambda x.x]\!]$.

Following the transition rules from Table 7, and the initial configuration

presented in Definition 13, $\langle C, [\,], [\,]\rangle$*, results in the following computation.*

$$\begin{aligned}
&\langle [M, M, M, \mathsf{apply}, \mathsf{apply}], [\,], [\,]\rangle \rightarrow \\
&\langle [M, M, \mathsf{apply}, \mathsf{apply}], [\,], [N]\rangle \rightarrow \\
&\langle [M, \mathsf{apply}, \mathsf{apply}], [\,], [N, N]\rangle \rightarrow \\
&\langle [\mathsf{apply}, \mathsf{apply}], [\,], [N, N, N]\rangle \rightarrow \\
&\langle [\mathsf{access}(0), \mathsf{return}], [N], [\mathsf{clo}([\mathsf{apply}], [\,]), N]\rangle \rightarrow \\
&\langle [\mathsf{return}], [N], [N, \mathsf{clo}([\mathsf{apply}], [\,]), N]\rangle \rightarrow \\
&\langle [\mathsf{apply}], [\,], [N, N]\rangle \rightarrow \\
&\langle [\mathsf{access}(0), \mathsf{return}], [N], [\mathsf{clo}([\,], [\,])]\rangle \rightarrow \\
&\langle [\mathsf{return}], [N], [N, \mathsf{clo}([\,], [\,])]\rangle \rightarrow \\
&\langle [\,], [\,], [N]\rangle
\end{aligned}$$

Where M *is the identity function,* $[\![(\lambda x.x)]\!]$*, and* N *the closure* $[\![(\lambda x.x)[\,]]\!]$*.*

Looking at the final configuration in Definition 15 and the compilation rules in Definition 16. It is clear that the final configuration is in the correct form, and that it is equivalent to $\lambda x.x$*, which is the expected result.*

3.4 The Krivine machine

The Krivine machine is a call-by-name machine described by Jean-Louis Krivine in [12]. The lambda calculus recognised by the presented machine can be seen in Definition 17.

Definition 17 (Krivine lambda calculus)**.** *The formation rules for untyped lambda calculus are as follows*

$$M ::= x \mid \lambda x.M \mid MN$$

$$\text{(CBN-1)} \qquad \frac{M \rightarrow M'}{MN \rightarrow M'N}$$

$$\text{(CBN-2)} \qquad \frac{}{(\lambda x.M)N \rightarrow M[N/x]}$$

Table 8: Call-by-name semantics

Similar to the SECD machine, it utilises De Bruijn notation, a control C, an environment E, and an intermediate evaluation stack S. However, unlike the SECD machine, the stacks S and E do not contain values, but instead *thunks*, where a thunk is a delayed evaluation in the form of a closure $\mathsf{clo}(C, E)$.

Definition 18 (Krivine machine)**.**

A configuration of the Krivine machine is of the form $\langle C, E, S\rangle$*, where*

- *C is the remaining control instructions*
- *E is the environment of variable bindings*
- *S is the stack holding thunks*

An initial configuration for the Krivine machine is $\langle C, \epsilon, \epsilon \rangle$, where $C \neq \epsilon$, as seen in Definition 13, and the final configuration is $\langle \mathit{grab} :: C, E, \epsilon \rangle$.

The formation rules of the control instructions for the Krivine machine, are as follows

$$C ::= I :: C \mid \epsilon$$
$$I ::= \mathit{access}(i) \mid \mathit{grab} \mid \mathit{push}(C)$$

The instructions describe the following behaviour

- *$\mathit{access}(i)$, start evaluating the i-th thunk in the environment*
- *grab, pop one argument and add it to the environment*
- *$\mathit{push}(C)$, push a thunk for code C*

The transition rules for the Krivine machine can be seen in Table 9. The transition rules are determined by the instruction on top of the control stack C. Here we have that there are only thunks placed in the evaluation stack S and the environment E. Thus, applications are always performed with thunks, hence the parameter-mechanism is call-by-name.

(KRIV-1)	$\langle \mathsf{access}(i) :: C, E, S \rangle \to \langle C', E', S \rangle$, if $E(i) = \mathsf{clo}(C', E')$
(KRIV-2)	$\langle \mathsf{grab} :: C, E, \mathsf{clo}(C', E') :: S \rangle \to \langle C, \mathsf{clo}(C', E') :: E, S \rangle$
(KRIV-3)	$\langle \mathsf{push}(C') :: C, E, S \rangle \to \langle C, E, \mathsf{clo}(C', E) :: S \rangle$

Table 9: Transition rules of the Krivine machine

As stated in Section 3.2, we need a definition for compiling terms of the source language to the target language. Definition 19, presents the rules for compilation from the lambda calculus, seen in Definition 17, to the Krivine instructions, seen in Definition 18. Lastly Example 7, presents the steps involved with making use of the abstract machine, compiling from terms to instructions and finally computing the compiled instructions using the machine.

Definition 19 (Compilation from lambda calculus to Krivine instructions)**.** *The compilation rules for lambda calculus to Krivine instructions are*

$$[\![\#i]\!] = \mathit{access}(i)$$
$$[\![\lambda M]\!] = \mathit{grab} : [\![M]\!]$$
$$[\![MN]\!] = \mathit{push}([\![N]\!]) : [\![M]\!]$$

Example 7 (Compiling and computing call-by-name lambda calculus with the Krivine machine)**.** *The following term is non-terminating in call-by-value, but terminating in call-by-name. Thus it has been chosen to demonstrate the call-by-name nature of the Krivine machine.*

$$(\lambda y.(\lambda x.y)((\lambda x.xx)(\lambda x.xx)))(\lambda z.z)$$

Before it can be compiled using the rules from Definition 19, it needs to be transformed into De Bruijn notation, as seen below

$$(\lambda.(\lambda.\#1)((\lambda.\#0\ \#0)(\lambda.\#0\ \#0)))(\lambda.\#0)$$

Then using the compilation rules for the Krivine machine, it results in the following instructions

$$\mathsf{push}([\mathsf{grab}, \mathsf{access}(0)]), \mathsf{grab}, \mathsf{push}(M), \mathsf{grab}, \mathsf{access}(1)$$

Where M is the recursive argument, $[\![(\lambda x.xx)(\lambda x.xx)]\!]$.

Following the transition rules from Table 9, and the initial configuration presented in Definition 13, $\langle C, [\ ], [\ ]\rangle$, results in the following computation.

$\langle[\mathsf{push}([\mathsf{grab}, \mathsf{access}(0)]), \mathsf{grab}, \mathsf{push}(M), \mathsf{grab}, \mathsf{access}(1)], [\ ], [\ ]\rangle \rightarrow$
$\langle[\mathsf{grab}, \mathsf{push}(M), \mathsf{grab}, \mathsf{access}(1)], [\ ], [\mathsf{clo}([\mathsf{grab}, \mathsf{access}(0)], [\ ])]\rangle \rightarrow$
$\langle[\mathsf{push}(M), \mathsf{grab}, \mathsf{access}(1)], [\mathsf{clo}([\mathsf{grab}, \mathsf{access}(0)], [\ ])], [\ ]\rangle \rightarrow$
$\langle[\mathsf{grab}, \mathsf{access}(1)], [\mathsf{clo}([\mathsf{grab}, \mathsf{access}(0)], [\ ])], [\mathsf{clo}(M, [\mathsf{clo}([\mathsf{grab}, \mathsf{access}(0)], [\ ])])]\rangle \rightarrow$
$\langle[\mathsf{access}(1)], [\mathsf{clo}(M, [\mathsf{clo}([\mathsf{grab}, \mathsf{access}(0)], [\ ])]), \mathsf{clo}([\mathsf{grab}, \mathsf{access}(0)], [\ ])], [\ ]\rangle \rightarrow$
$\langle[\mathsf{grab}, \mathsf{access}(0)], [\ ], [\ ]\rangle$

Where M is the recursive argument, $[\![(\lambda x.xx)(\lambda x.xx)]\!]$.

Looking at the final configuration in Definition 18 and the compilation rules in Definition 19. It is clear that the final configuration is in the correct form, and that it is equivalent to $\lambda z.z$, which is the expected result.

3.5 An abstract machine for call-by-push-value

The abstract machine for call-by-push-value is based on the Modern SECD machine, and as such it has the same configuration consisting of three stacks: an evaluation stack S, an environment E, a control C. The SECD machine performs call-by-value, by ensuring that there are only values placed in S and E, unlike the Krivine machine where the stacks S and E contain thunks rather than values, resulting in call-by-name.

However, in CBPV the definition of values is different from that of CBV, such as thunks being considered values in CBPV, as seen in Definition 1 in Section 2.2. Furthermore, evaluation of CBPV does not result in values, but rather terminals, as presented in Definition 3 in Section 2.2. The stack S therefore contains elements in the set $\mathbf{Val} \cup \mathbf{Term}$. Definition 20 presents the abstract machine for CBPV.

Definition 20 (CBPV AM)**.**

A configuration of the CBPV machine is of the form $\langle C, E, S\rangle$*, where*

- C *is the remaining control instructions*
- E *is the environment of variable bindings*
- S *is the stack of values and terminals*

An initial configuration for the CBPV AM is $\langle C, \epsilon, \epsilon\rangle$*, where* $C \neq \epsilon$*, as seen in Definition 13, and the final configuration is* $\langle \epsilon, \epsilon, T :: S\rangle$*, where* $T \in \textbf{\textit{Term}}$*.*

The formation rules of the control instructions for the CBPV AM are as follows

$$
\begin{aligned}
&C ::= I :: C \mid \epsilon \\
&I ::= \mathit{access}(i) \mid \mathit{closure}(C) \mid \mathit{return} \mid \mathit{apply} \mid \mathit{begin} \mid \mathit{end} \\
&\quad \mid \mathit{thunk}(C) \mid \mathit{force} \mid \mathit{return}(V)
\end{aligned}
$$

The instructions describe the following behaviour

- $\mathit{access}(i)$*, push the i-th field of the environment*
- $\mathit{closure}(C)$*, push the closure with the current environment*
- *return, terminate current function, and jump back to caller*
- *apply, pop function closure and argument, and perform application*
- *begin, pop value and add it to environment*
- *end, discard first entry in environment*
- $\mathit{thunk}(C)$*, pushes thunk to stack*
- *force, pops thunk of stack and evaluates* C
- $\mathit{return}(V)$*, pushes terminal to stack*

As mentioned, the CBPV AM is based on the SECD machine, and as such the transition rules concerning abstractions, applications, and variable lookup remain unchanged. It has however been extended with additional rules in order to handle the additional constructs of CBPV. Table 10 presents the transition rules of the CBPV AM.

As stated in Section 3.2, we need a definition for compiling terms of the source language to the target language. Definition 21, presents the rules for compilation from CBPV, presented in Section 2.2, to the CBPV machine instructions, seen in Definition 20.

In addition, as presented in Section 2.2, call-by-value (CBV) and call-by-name (CBN) can be translated to CBPV. As such, the SECD and Krivine

$$\text{(AM-1)} \quad \langle \mathsf{access}(i) :: C, E, S\rangle \to \langle C, E, E(i) :: S\rangle$$

$$\text{(AM-2)} \quad \langle \mathsf{closure}(C') :: C, E, S\rangle \to \langle C, E, \mathsf{clo}(C', E) :: S\rangle$$

$$\text{(AM-3)} \quad \langle \mathsf{return} :: C, E, t :: \mathsf{clo}(C', E') :: S\rangle \to \langle C', E', t :: S\rangle$$

$$\text{(AM-4)} \quad \langle \mathsf{apply} :: C, E, v :: \mathsf{clo}(C', E') :: S\rangle \to \langle C', v :: E', \mathsf{clo}(C, E) :: S\rangle$$

$$\text{(AM-5)} \quad \langle \mathsf{begin} :: C, E, \mathsf{return}(v) :: S\rangle \to \langle C, v :: E, S\rangle$$

$$\text{(AM-6)} \quad \langle \mathsf{begin} :: C, E, v :: S\rangle \to \langle C, v :: E, S\rangle$$

$$\text{(AM-7)} \quad \langle \mathsf{end} :: C, v :: E, S\rangle \to \langle C, E, S\rangle$$

$$\text{(AM-8)} \quad \langle \mathsf{thunk}(C') :: C, E, S\rangle \to \langle C, E, \mathsf{thunk}(C') :: S\rangle$$

$$\text{(AM-9)} \quad \langle \mathsf{force} :: C, E, \mathsf{thunk}(C') :: S\rangle \to \langle C' :: C, E, S\rangle$$

$$\text{(AM-10)} \quad \langle \mathsf{return}(\mathsf{thunk}(C')) :: C, E, S\rangle \to \langle C, E, \mathsf{return}(\mathsf{thunk}(C')) :: S\rangle$$

$$\text{(AM-11)} \quad \langle \mathsf{return}(\mathsf{access}(i)) :: C, E, S\rangle \to \langle C, E, \mathsf{return}(E(i)) :: S\rangle$$

Table 10: Transition rules of the CBPV AM

example terms, presented in Example 6 and 7 respectively, can be translated into CBPV, from there into CBPV machine instructions, and finally be evaluated and reach the same result all while preserving the semantics. Therefore, Example 8 will present the translation, transformation, compilation, and execution of the SECD example term in the CBPV machine, and Example 9 likewise for the Krivine example.

Definition 21 (Compilation from call-by-push-value to CBPV machine instructions)**.** *The compilation rules for CBPV lambda calculus to CBPV machine instructions are*

$$\begin{aligned}
[\![\#i]\!] &= \mathit{access}(i) \\
[\![\lambda.M]\!] &= \mathit{closure}([\![M]\!] : \mathit{return}) \\
[\![MV]\!] &= [\![M]\!] : [\![V]\!] : \mathit{apply} \\
[\![\mathit{let}\ V\ \mathit{in}\ M]\!] &= [\![V]\!] : \mathit{begin} : [\![M]\!] : \mathit{end} \\
[\![M\ \mathit{to}\ \mathit{in}\ N]\!] &= [\![M]\!] : \mathit{begin} : [\![N]\!] : \mathit{end} \\
[\![\mathit{return}\ V]\!] &= \mathit{return}([\![V]\!]) \\
[\![\mathit{thunk}\ M]\!] &= \mathit{thunk}([\![M]\!]) \\
[\![\mathit{force}\ (\mathit{thunk}\ M)]\!] &= [\![M]\!] \\
[\![\mathit{force}\ V]\!] &= [\![V]\!] : \mathit{force}
\end{aligned}$$

Example 8 (Compiling and computing call-by-push-value lambda calculus with the CBPV AM: CBV example term)**.** *The following term, is the term used in Example 6 for the SECD machine.*

$$(\lambda x.x)((\lambda x.x)(\lambda x.x))$$

Since the SECD machine evaluates using the CBV semantics, the term will be translated as a CBV term to CBPV. Translating the term into CBPV is done

according to the rules presented in Table 2 in Section 2.2. Using those rules we get the following CBPV term

$$M\ to\ f\ in\ (M\ to\ p\ in\ (M\ to\ q\ in\ (force\ p)q))\ to\ g\ in\ (force\ f)g$$

Where M *is the identity function,* $return\ (thunk\ (\lambda x.return\ x))$.

Before it can be compiled into CBPV machine instruction, it needs to be transformed into De Bruijn notation, as seen below

$$M\ to\ in\ (M\ to\ in\ (M\ to\ in\ (force\ \#1)\#0))\ to\ in\ (force\ \#1)\#0$$

Where M *is the translated identity function,* $return\ (thunk\ (\lambda.return\ \#0))$.

Then using the compilation rules for the CBPV machine, it results in the following instructions

$$M, \mathsf{begin}, M, \mathsf{begin}, M, \mathsf{begin}, P, \mathsf{end}, \mathsf{end}, \mathsf{begin}, P, \mathsf{end}, \mathsf{end}$$

Where M *is the identity function,* $[\![return\ (thunk\ (\lambda.return\ \#0))]\!]$, *and* P *is the application* $[\![(force\ \#1)\#0]\!]$.

Finally, following the transition rules from Table 10 and the initial configuration presented in Definition 20, $\langle C, [\,], [\,]\rangle$, *it results in the following computation*

$\langle[\mathsf{return}(M), \mathsf{begin}, \mathsf{return}(M), \mathsf{begin}, \mathsf{return}(M), \mathsf{begin}, P, Q], [\ ], [\ ]\rangle \rightarrow$
$\langle[\mathsf{begin}, \mathsf{return}(M), \mathsf{begin}, \mathsf{return}(M), \mathsf{begin}, P, Q], [\ ], [\mathsf{return}(M)]\rangle \rightarrow$
$\langle[\mathsf{return}(M), \mathsf{begin}, \mathsf{return}(M), \mathsf{begin}, P, Q], [M], [\ ]\rangle \rightarrow$
$\langle[\mathsf{begin}, \mathsf{return}(M), \mathsf{begin}, P, Q], [M], [\mathsf{return}(M)]\rangle \rightarrow$
$\langle[\mathsf{return}(M), \mathsf{begin}, P, Q], [M, M], [\ ]\rangle \rightarrow$
$\langle[\mathsf{begin}, P, Q], [M, M], [\mathsf{return}(M)]\rangle \rightarrow$
$\langle[P, Q], [M, M, M], [\ ]\rangle \rightarrow$
$\langle[\mathsf{force}, \mathsf{access}(0), \mathsf{apply}, Q], [M, M, M], [M]\rangle \rightarrow$
$\langle[\mathsf{closure}([\mathsf{return}(\mathsf{access}(0)), \mathsf{return}]), \mathsf{access}(0), \mathsf{apply}, Q], [M, M, M], [\ ]\rangle \rightarrow$
$\langle[\mathsf{access}(0), \mathsf{apply}, Q], [M, M, M], [\mathsf{clo}([\mathsf{return}(\mathsf{access}(0)), \mathsf{return}], [M, M, M])]\rangle \rightarrow$
$\langle[\mathsf{apply}, Q], [M, M, M], [M, \mathsf{clo}([\mathsf{return}(\mathsf{access}(0)), \mathsf{return}], [M, M, M])]\rangle \rightarrow$
$\langle[\mathsf{return}(\mathsf{access}(0)), \mathsf{return}], [M, M, M, M], [\mathsf{clo}([Q], [M, M, M])]\rangle \rightarrow$
$\langle[\mathsf{return}], [M, M, M, M], [\mathsf{return}(M), \mathsf{clo}([Q], [M, M, M])]\rangle \rightarrow$
$\langle[Q], [M, M, M], [\mathsf{return}(M)]\rangle \rightarrow$
$\langle[\mathsf{end}, \mathsf{begin}, P, \mathsf{end}, \mathsf{end}], [M, M], [\mathsf{return}(M)]\rangle \rightarrow$
$\langle[\mathsf{begin}, P, \mathsf{end}, \mathsf{end}], [M], [\mathsf{return}(M)]\rangle \rightarrow$
$\langle[P, \mathsf{end}, \mathsf{end}], [M, M], [\ ]\rangle \rightarrow$
$\langle[\mathsf{force}, \mathsf{access}(0), \mathsf{apply}, \mathsf{end}, \mathsf{end}], [M, M], [M]\rangle \rightarrow$
$\langle[\mathsf{closure}([\mathsf{return}(\mathsf{access}(0)), \mathsf{return}]), \mathsf{access}(0), \mathsf{apply}, \mathsf{end}, \mathsf{end}], [M, M], [\ ]\rangle \rightarrow$
$\langle[\mathsf{access}(0), \mathsf{apply}, \mathsf{end}, \mathsf{end}], [M, M], [\mathsf{clo}([\mathsf{return}(\mathsf{access}(0)), \mathsf{return}], [M, M])]\rangle \rightarrow$
$\langle[\mathsf{apply}, \mathsf{end}, \mathsf{end}], [M, M], [M, \mathsf{clo}([\mathsf{return}(\mathsf{access}(0)), \mathsf{return}], [M, M])]\rangle \rightarrow$
$\langle[\mathsf{return}(\mathsf{access}(0)), \mathsf{return}], [M, M, M], [\mathsf{clo}([\mathsf{end}, \mathsf{end}], [M, M])]\rangle \rightarrow$
$\langle[\mathsf{return}], [M, M, M], [\mathsf{return}(M), \mathsf{clo}([\mathsf{end}, \mathsf{end}], [M, M])]\rangle \rightarrow$
$\langle[\mathsf{end}, \mathsf{end}], [M, M], [\mathsf{return}(M)]\rangle \rightarrow$
$\langle[\mathsf{end}], [M], [\mathsf{return}(M)]\rangle \rightarrow$
$\langle[\ ], [\ ], [\mathsf{return}(M)]\rangle$

Where M is the identity function $[\![thunk\ (\lambda.return\ \#0)]\!]$, R the body of the identity function $[\![return\ x]\!]$: return*, Q the instructions* $\mathsf{end}, \mathsf{end}, \mathsf{begin}, \mathsf{access}(1), \mathsf{force}, \mathsf{access}(0), \mathsf{apply}, \mathsf{end}, \mathsf{end}$*, and P the application $[\![(force\ \#1)\#0]\!]$.*

Looking at the final configuration in Definition 20 and the compilation rules in Definition 21, the resulting CBPV term is $return\ (thunk\ (\lambda.return\ \#0))$. Transforming the term back to named variables we get $return\ (thunk(\lambda x.return\ x))$, and finally translating the term from CBPV to CBV, we get $\lambda x.x$, which is the result from the SECD machine. The CBPV AM therefore corroborates the results of SECD machine in Example 6, and the translations described by Levy [16].

Example 9 (Compiling and computing call-by-push-value lambda calculus with the CBPV machine: CBN example term)**.** *The following term, is the term used*

in Example 7 for the Krivine machine.

$$(\lambda y.(\lambda x.y)((\lambda x.xx)(\lambda x.xx)))(\lambda z.z)$$

Since the Krivine machine evaluates using the CBN semantics, the term will be translated as a CBN term to CBPV. Translating the term into CBPV is done according to the rules presented in Table 3 in Section 2.2. Using those rules we get the following CBPV term

$$(\lambda y.(\lambda x.force\ y)(thunk\ (M\ (thunk\ M))))(thunk\ (\lambda z.force\ z))$$

Where M *is the abstraction* $\lambda x.(force\ x)(thunk\ (force\ x))$.

Before it can be compiled into CBPV machine instructions, it needs to be transformed into De Bruijn notation, as seen below

$$(\lambda.(\lambda.force\ \#1)(thunk\ (M\ (thunk\ M))))(thunk\ (\lambda.force\ \#0))$$

Where M *is the abstraction* $\lambda.(force\ \#0)(thunk\ (force\ \#0))$.

Then using the compilation rules for the CBPV machine, it results in the following instructions

$$\mathsf{closure}([\mathsf{closure}([\mathsf{access}(1), \mathsf{force}, \mathsf{return}]), R, \mathsf{apply}, \mathsf{return}]), N, \mathsf{apply}$$

Where N *is the argument* $[\![thunk\ (\lambda.force\ \#0)]\!]$, R *is the recursive argument* $[\![thunk\ (M\ (thunk\ M))]\!]$, *where* M *is the abstraction* $[\![\lambda.(force\ \#0)(thunk\ (force\ \#0))]\!]$.

Finally, following the transition rules from Table 10 and the initial configuration presented in Definition 20, $\langle C, [\,], [\,]\rangle$, *it results in the following computation*

$\langle[\mathsf{closure}([\mathsf{closure}([\mathsf{access}(1), \mathsf{force}, \mathsf{return}]), R, \mathsf{apply}, \mathsf{return}]), N, \mathsf{apply}], [\,], [\,]\rangle \rightarrow$
$\langle[N, \mathsf{apply}], [\,], [\mathsf{clo}([\mathsf{closure}([\mathsf{access}(1), \mathsf{force}, \mathsf{return}]), R, \mathsf{apply}, \mathsf{return}], [\,])]\rangle \rightarrow$
$\langle[\mathsf{apply}], [\,], [N, \mathsf{clo}([\mathsf{closure}([\mathsf{access}(1), \mathsf{force}, \mathsf{return}]), R, \mathsf{apply}, \mathsf{return}], [\,])]\rangle \rightarrow$
$\langle[\mathsf{closure}([\mathsf{access}(1), \mathsf{force}, \mathsf{return}]), R, \mathsf{apply}, \mathsf{return}], [N], [\mathsf{clo}([\,], [\,])]\rangle \rightarrow$
$\langle[R, \mathsf{apply}, \mathsf{return}], [N], [\mathsf{clo}([\mathsf{access}(1), \mathsf{force}, \mathsf{return}], [N]), \mathsf{clo}([\,], [\,])]\rangle \rightarrow$
$\langle[\mathsf{apply}, \mathsf{return}], [N], [R, \mathsf{clo}([\mathsf{access}(1), \mathsf{force}, \mathsf{return}], [N]), \mathsf{clo}([\,], [\,])]\rangle \rightarrow$
$\langle[\mathsf{access}(1), \mathsf{force}, \mathsf{return}], [R, N], [\mathsf{clo}([\mathsf{return}], [N]), \mathsf{clo}([\,], [\,])]\rangle \rightarrow$
$\langle[\mathsf{force}, \mathsf{return}], [R, N], [N, \mathsf{clo}([\mathsf{return}], [N]), \mathsf{clo}([\,], [\,])]\rangle \rightarrow$
$\langle[\mathsf{closure}([\mathsf{access}(0), \mathsf{force}, \mathsf{return}]), \mathsf{return}], [R, N], [\mathsf{clo}([\mathsf{return}], [N]), \mathsf{clo}([\,], [\,])]\rangle \rightarrow$
$\langle[\mathsf{return}], [R, N], [\mathsf{clo}([\mathsf{access}(0), \mathsf{force}, \mathsf{return}], [R, N]), \mathsf{clo}([\mathsf{return}], [N]), \mathsf{clo}([\,], [\,])]\rangle \rightarrow$
$\langle[\mathsf{return}], [N], [\mathsf{clo}([\mathsf{access}(0), \mathsf{force}, \mathsf{return}], [R, N]), \mathsf{clo}([\,], [\,])]\rangle \rightarrow$
$\langle[\,], [\,], [\mathsf{clo}([\mathsf{access}(0), \mathsf{force}, \mathsf{return}], [R, N])]\rangle$

Where N *is the argument* $[\![thunk\ (\lambda.force\ \#0]\!]$, R *is the recursive argument* $[\![thunk\ (M\ thunk\ M)]\!]$, *where* M *is the abstraction* $[\![\lambda.(force\ \#0)(thunk\ (force\ \#0))]\!]$.

Looking at the final configuration in Definition 20 and the compilation rules in Definition 21, the resulting CBPV term is a closure $\lambda.force\ \#0$. *Transforming the term back to named variables we get* $\lambda z.force\ z$, *and finally translating*

the term from CBPV to CBN, we get $\lambda z.z$, which is the result from the Krivine machine. The CBPV AM therefore corroborates the results of Krivine machine in Example 7, and the translations described by Levy [16].

3.6 Analysis of the call-by-push-value abstract machine

This section will discuss the performance of the CBPV AM relative to the SECD and Krivine machines when run on the same term, as seen in Examples 6-9. While the section will investigate and argue for some of the performance deficits in the CBPV AM, it will not be an exhaustive or in-depth performance analysis of the CBPV AM, but instead provide simply insights into aspects of the performance.

Looking first at the call-by-value example, seen in Example 6 and 8. We see that the CBPV AM performs around 2.9 times more reductions than the SECD machine, as seen in Table 11. These can in large part be explained by the initial translation from call-by-value to call-by-push-value. Looking at the translation rules of call-by-value to call-by-push-value, seen in Table 2, we see that an application MN is translated to the following

$$MN \rightarrow_{trans} M \; to \; f \; in \; (N \; to \; g \; in \; (force \; f)g)$$

Then by looking at the compilation rules for the SECD machine, seen in Table 16, we have that an application is compiled to 3 instructions, $[\![M]\!] : [\![V]\!] : \mathsf{apply}$. Whereas, looking at the compilation rules for the CBPV AM, as seen in Table 21, the translated equivalent results in 8 instructions, $[\![M]\!] : \mathsf{begin} : [\![N]\!] : \mathsf{begin} : \mathsf{access}(g) : \mathsf{access}(f) : \mathsf{force} : \mathsf{apply}$.

In Example 8, this leads to 19 CBPV AM instructions, compared to just 5 SECD instructions. Note that the discrepancy with respect to Table 11 is largely due to nested instructions.

SECD reductions	9
CBPV AM reductions	25

Table 11: Performance on the call-by-value example term

The comparison of the CBPV AM and the Krivine machine, is however not as straightforward. While Table 12 presents that the CBPV AM performs around 2.2 times more reductions compared to the Krivine machine, they are difficult to directly compare in contrast to SECD comparison, since the CBPV AM is based on the SECD machine. However, there is a fundamental difference between the SECD and Krivine machines, as noted by Leroy [15], the SECD and Krivine machines illustrate two subtle different ways of evaluating function applications, namely *eval-apply* and *push-enter* respectively. Thereby making it difficult to make a direct comparison.

In [15], Leroy goes on to present the ZAM machine, which is a call-by-value push-enter model, and also the underlying model of the bytecode interpreters of OCaml. Leroy presents that the eval-apply approach performs extra work

Krivine machine reductions	5
CBPV reductions	11

Table 12: Performance on the call-by-name example term

compared to the push-enter approach. It would therefore be interesting to see if the CBPV AM could leverage the ZAM machine to improve performance.

4 Proving correctness of abstract machines

This section will present the work behind proving the correctness of the CBPV AM. First, Section 4.1 will discuss what correctness means in the context of an abstract machine, as well as the methods. Finally, Section 4.2 will present the proof of correctness for the CBPV AM.

4.1 Correctness of abstract machines

This section will discuss the meaning of correctness with respect to abstract machines and their respective semantics. The following presentation is largely based on Xavier Leroy's presentation in [15].

In order to define correctness in the context of abstract machines and their semantics. It is necessary at this point to understand, that we have two ways of executing a given source term. First is by evaluating the term using the semantics, i.e. $M \to^* T$, and the other is by compiling it to machine instructions and then executing the instructions using an abstract machine, such as $\langle [\![M]\!], \epsilon, \epsilon \rangle \to^* \langle \epsilon, \epsilon, T :: \epsilon \rangle$ in the case of the CBPV AM.

The question then becomes, whether the two execution paths agree, which is to say, does the abstract machine adhere to the semantics of the source language. The notion of observational equivalence can be used as a means to assert equivalence based on observable behaviours. So for the execution of a term M and a configuration $\langle [\![M]\!], \epsilon, \epsilon \rangle$, we have that, if they exhibit the same behaviour, they can be said to be equivalent. The behaviours are

- Termination on (a state representing) a terminal $T \in \mathbf{Term}$. If the semantics terminate, then so does the machine, and vice versa for the machine.

$$M \to^* T \iff \langle [\![M]\!], \epsilon, \epsilon \rangle \to^* \langle \epsilon, \epsilon, [\![T]\!] :: \epsilon \rangle \tag{1}$$

- Divergence, never terminating. If the semantics reduces forever, then so does the machine, and vice versa for the machine.

$$M \to^* \infty \iff \langle [\![M]\!], \epsilon, \epsilon \rangle \to^* \infty \tag{2}$$

- Error, getting stuck. If the semantics get stuck, then so does the machine, and vice versa for the machine.

$$M \to^* M' \not\to \iff \langle [\![M]\!], \epsilon, \epsilon \rangle \to^* \ldots \not\to \tag{3}$$

Leroy presents two ways of proving correctness of the SECD machine, which are relevant since the CBPV AM is based on the SECD machine, using small-step semantics and big-step semantics. However, he finds that the proofs using small-step are heavy, and that while the proofs using big-step are more convenient, they do not account for all of the behaviours seen in Equations 1-3.

Like Leroy, Nielson & Nielson [21] used big-step semantics to prove correctness of their implementation. They did it first by proof of induction on the derivation trees, that for each derivation tree in the semantics there was a corresponding finite computation sequence on the abstract machine. Secondly, by induction on the length of the computation sequences, that there was a corresponding derivation tree in the big-step semantics. The proof is also quite heavy, and following it Nielson & Nielson present an alternative proof technique, which is to use bisimulation as seen in Definition 22.

Definition 22 (Correctness of Implementation using Bisimulation [21])**.**

1. *Prove that one step in the structural operational semantics can be simulated by a non-empty sequence of steps on the abstract machine. Show that this extends to sequences of steps in the structural operational semantics.*
2. *Prove that a carefully selected non-empty sequence of steps on the abstract machine can be simulated by a step in the structural operational semantics. Show that this extends to more general sequences of steps on the abstract machine.*

Bisimulation was discovered by Park [22], used and refined by Milner [18] to prove his translation from λ-calculus to π-calculus, and lastly expanded upon by Sangiorgi [24].

By using bisimulation, it is possible to demonstrate that the compilation respects Equations 1-3 all at once, without the proof becoming unmanageable or heavy. Furthermore, it becomes straightforward to perform a resource analysis, since the steps in the machine are directly tied to reductions in the semantics. Hence the proof of correctness for the CBPV AM will be using bisimulation.

4.2 Proof of correctness for CBPV AM

The proof uses bisimulation to prove the correctness of the CBPV AM. The definition of bisimulation employed in this proof can be seen in Definition 23, and Definition 24, presents the notion of an abstract machine implementing a term.

Definition 23 (Bisimulation)**.** *A relation $R \subseteq CBPV \times CBPV\ AM$ is called a bisimulation if it holds that when $(M, \langle C, E, S\rangle) \in R$ then*

1. $M \to M' \Rightarrow \exists \langle C', E', S'\rangle.$
   $\langle C, E, S\rangle \to^* \langle C', E', S'\rangle,$ *such that* $(M', \langle C', E', S'\rangle) \in R$

2. $\langle C, E, S\rangle \to \langle C', E', S'\rangle \Rightarrow \exists \langle C'', E'', S''\rangle.$
$\langle C', E', S'\rangle \to^* \langle C'', E'', S''\rangle$ *and* $M \to M'$, *such that* $(M', \langle C'', E'', S''\rangle) \in R$

Definition 24 (Implementable)**.** $\langle C, E, S\rangle$ *implements* M*, if there exists a bisimulation* R*, such that* $(M, \langle C, E, S\rangle) \in R$*. We write* $\langle C, E, S\rangle$ imp M*.*

Definition 25 and 26 introduce notation necessary for reasoning about the decompilation of machine configuration to terms. Definition 25, allows us to disregard trailing end and return instructions, while Definition 26 provides the translation between semantic binding and syntactic binding, by performing substitutions in the term with respect to the bindings in the environment, E. The use of Definition 25 and 26 can be seen in the machine-equivalent of $(\lambda x.M)V \to M[V/x]$ below

$$\langle [\![V]\!] : \mathsf{closure}([\![M]\!] : \mathsf{return}) : \mathsf{apply}, \epsilon, \epsilon\rangle \to^3 \langle [\![M']\!] : \mathsf{return}, [\![V]\!], S\rangle$$

In order to match the term $M[V/x]$, it is necessary to disregard the trailing return and substitute the binding from E into the compiled term $[\![M']\!]$, to produce $[\![M'\sigma_E]\!] = M[V/x]$. Otherwise the machine configuration and term wouldn't match, despite the two being equivalent.

Definition 25 (Garbage collection)**.** *We write* $C \approx C'$*, if* $C = C' : (\mathsf{end}, \mathsf{return})^*$

Definition 26 (Translating semantic binding to syntactic binding)**.** *For an* E *we have substitution* σ_E *given by* $\sigma_E(i) = E(i)$*.*

Definition 27 is the bisimulation relation proposed to prove the correctness of the implementation. The membership constraints for the relation clearly encapsulate the three sets of terms in CBPV, namely the first constraint matches **Val** and **Term**, while the second constraint matches **Comp**. Worth noting is the condition with respect to the top element of the stack S, seen in the second constraint. If $S \neq \mathsf{clo}(\_, \_) :: S'$, then we know the term is not a nested application, however, if $C = [\![V]\!] : \mathsf{apply}$ and $S = \mathsf{clo}(C', E') :: S'$ then we know the reduction is of the form $((\lambda x.\lambda y.N)V')V \to (\lambda y.N[V'/x])V$, which results in the following machine reduction

$$\langle \mathsf{closure}(\mathsf{closure}([\![N]\!] : \mathsf{return}) : \mathsf{return}) : [\![V']\!] : \mathsf{apply} : [\![V]\!] : \mathsf{apply}, \epsilon, \epsilon\rangle \to^5$$
$$\langle [\![V]\!] : \mathsf{apply}, E', \mathsf{clo}([\![N]\!] : \mathsf{return}, [[\![V']\!]]) :: S\rangle$$

The case enables R_H to capture the otherwise excluded pair $((\lambda y.N[V'/x])V, \langle [\![V]\!] : \mathsf{apply}, E', \mathsf{clo}([\![N]\!] : \mathsf{return}, [[\![V']\!]]) :: S\rangle) \in R_H$.

Definition 27 (A bisimulation for the implementation)**.**

$$\begin{aligned} R_H = \{&(Q, \langle \epsilon, \epsilon, [\![Q]\!]\rangle) \mid Q \in (\boldsymbol{Val} \cup \boldsymbol{Term})\} \\ \cup \{&(M, \langle C, E, S\rangle) \mid M \in \boldsymbol{Comp}. \\ &\quad \textit{If } S \neq \mathit{clo}(\_, \_) :: S' \textit{ then } \exists M'.C \approx [\![M']\!] \textit{ and } M'\sigma_E = M. \\ &\quad \textit{If } C = [\![V]\!] : \mathit{apply} \textit{ and } S = \mathit{clo}(C', E') :: S' \textit{ then} \\ &\quad \mathit{clo}(C', E') = [\![\lambda.M'\sigma_{E'}]\!] \textit{ and } M = (\lambda.M'\sigma_{E'})V\} \end{aligned}$$

Lastly, the proof of correctness can be seen in Theorem 1, followed by its proof.

Theorem 1 (Correctness of CBPV AM)**.**

$\forall M \in \boldsymbol{Comp} \cup \boldsymbol{Val}.\ \langle C, E, S\rangle\ \mathit{imp}\ M$, *where* $(M, \langle C, E, S\rangle) \in R_H$.

Proof. We have that, for each term M it can be compiled ($[\![\ ]\!]$) to a machine configuration $\langle C, E, S\rangle$.

The proof idea involves proving that R_H is in fact a bisimulation using Definition 23, and to do it via a case-analysis on the respective pairs in R_H.

The induction hypothesis is therefore, if $(M, \langle C, E, S\rangle)$ is a pair in R_H, then the immediate components of M are also represented in R_H, and therefore has matching transitions.

We now show that the relation R_H, seen in Definition 27, is a bisimulation correlating terms of the CBPV language and machine configurations of the CBPV AM.

$(\mathit{thunk}\ M, \langle \mathsf{thunk}([\![M]\!]), \epsilon, \epsilon\rangle) \in R_H$.

- $\mathit{thunk}\ M \not\to$, and so there is nothing to match.
- $\langle \mathsf{thunk}([\![M]\!]), \epsilon, \epsilon\rangle \to \langle \epsilon, \epsilon, \mathsf{thunk}([\![M]\!])\rangle \not\to$, and $\mathit{thunk}\ M \not\to$. In the definition of R_H we see that the pair $(\mathit{thunk}\ M, \langle \epsilon, \epsilon, \mathsf{thunk}([\![M]\!])\rangle) \in R_H$.

$(\lambda x.M, \langle \mathsf{closure}([\![M]\!] : \mathsf{return}), \epsilon, \epsilon\rangle) \in R_H$.

- $\lambda x.M \not\to$, and so there is nothing to match.
- $\langle \mathsf{closure}([\![M]\!] : \mathsf{return}), \epsilon, \epsilon\rangle \to \langle \epsilon, \epsilon, \mathsf{clo}([\![M]\!] : \mathsf{return}, [\ ])\rangle \not\to$, and $\lambda x.M \not\to$. In the definition of R_H we see that the pair $(\lambda x.M, \langle \epsilon, \epsilon, \mathsf{clo}([\![M]\!] : \mathsf{return}, [\ ])\rangle) \in R_H$.

$(MV, \langle [\![M]\!] : [\![V]\!] : \mathsf{apply}, \epsilon, \epsilon\rangle) \in R_H$.

- $MV \to M'V$, using rule (CBPV-APP). If M' is a component of M in the pair $(M, \langle C, E, S\rangle) \in R_H$. We have that, per the the induction hypothesis, there exists a $\langle C, E, S\rangle \to^* \langle C', E', S'\rangle$, such that $(M', \langle C', E', S'\rangle) \in R_H$.
- $\langle [\![M]\!] : [\![V]\!] : \mathsf{apply}, \epsilon, \epsilon\rangle \to \langle [\![M'']\!] : [\![V]\!] : \mathsf{apply}, E', S'\rangle$, and $M \to M'$, where $M''\sigma_E = M'$. Then, per the induction hypothesis, there exists a continuation $\langle [\![M'']\!] : [\![V]\!] : \mathsf{apply}, E', S'\rangle \to^* \langle [\![M''']\!] : [\![V]\!] : \mathsf{apply}, E'', S''\rangle$, such that $(M'', \langle [\![M''']\!], E'', S''\rangle) \in R_H$.

$((\lambda x.M)V, \langle \mathsf{closure}([\![M]\!] : \mathsf{return}) : [\![V]\!] : \mathsf{apply}, \epsilon, \epsilon\rangle) \in R_H$.

- $(\lambda x.M)V \to M[V/x]$, using rule (CBPV-BETA). That is matched by $\langle \mathsf{closure}([\![M]\!] : \mathsf{return}) : [\![V]\!] : \mathsf{apply}, \epsilon, \epsilon\rangle \to^3 \langle [\![M]\!] : \mathsf{return}, [\![V]\!], \mathsf{clo}([\,], [\,])\rangle$. In the definition of R_H we see that the pair $(M[V/x], \langle [\![M]\!] : \mathsf{return}, [\![V]\!], \mathsf{clo}([\,], [\,])\rangle) \in R_H$.

- $\langle \mathsf{closure}(\llbracket M \rrbracket : \mathsf{return}) : \llbracket V \rrbracket : \mathsf{apply}, \epsilon, \epsilon \rangle \rightarrow^3 \langle \llbracket M \rrbracket : \mathsf{return}, \llbracket V \rrbracket, \mathsf{clo}([\,], [\,]) \rangle$, and $(\lambda x.M)V \rightarrow M[V/x]$. In the definition R_H we see that the pair $(M[V/x], \langle \llbracket M \rrbracket : \mathsf{return}, \llbracket V \rrbracket, \mathsf{clo}([\,], [\,]) \rangle) \in R_H$.

$(force\ V, \langle \llbracket V \rrbracket : \mathsf{force}, \epsilon, \epsilon \rangle) \in R_H$.

- For $force\ V$ to reduce, V needs to be a *thunk* M as per (CBPV-FORCE), as such $force\ thunk\ M \rightarrow M$. This is matched by $\langle \mathsf{thunk}(\llbracket M \rrbracket) : \mathsf{force}, \epsilon, \epsilon \rangle \rightarrow^2 \langle \llbracket M \rrbracket, \epsilon, \epsilon \rangle$. In the definition of R_H we see that the pair $(M, \langle \llbracket M \rrbracket, \epsilon, \epsilon \rangle) \in R_H$.
- Establishing again, that for a reduction to take place, V must be a *thunk* M, and as such $\langle \mathsf{thunk}(\llbracket M \rrbracket) : \mathsf{force}, \epsilon, \epsilon \rangle \rightarrow^2 \langle \llbracket M \rrbracket, \epsilon, \epsilon \rangle$. This is matched by $force\ thunk\ M \rightarrow M$. In the definition of R_H we see that the pair $(M, \langle \llbracket M \rrbracket, \epsilon, \epsilon \rangle) \in R_H$.

$(return\ V, \langle \mathsf{return}(\llbracket V \rrbracket), \epsilon, \epsilon \rangle) \in R_H$.

- $return\ V \not\rightarrow$, and so there is nothing to match.
- $\langle \mathsf{return}(\llbracket V \rrbracket), \epsilon, \epsilon \rangle \rightarrow \langle \epsilon, \epsilon, \mathsf{return}(\llbracket V \rrbracket) \rangle \not\rightarrow$, and $return\ V \not\rightarrow$. In the definition of R_H we see that the pair $(return\ V, \langle \epsilon, \epsilon, \mathsf{return}(\llbracket V \rrbracket) \rangle) \in R_H$.

$(let\ x = V\ in\ M, \langle \llbracket V \rrbracket : \mathsf{begin} : \llbracket M \rrbracket : \mathsf{end}, \epsilon, \epsilon \rangle) \in R_H$.

- $let\ x = V\ in\ M \rightarrow M[V/x]$, using rule (CBPV-LET). That is matched by $\langle \llbracket V \rrbracket : \mathsf{begin} : \llbracket M \rrbracket : \mathsf{end}, \epsilon, \epsilon \rangle \rightarrow^2 \langle \llbracket M \rrbracket : \mathsf{end}, \llbracket V \rrbracket, \epsilon \rangle$. In the definition of R_H we see that the pair $(M[V/x], \langle \llbracket M \rrbracket : \mathsf{end}, \llbracket V \rrbracket, \epsilon \rangle) \in R_H$.
- $\langle \llbracket V \rrbracket : \mathsf{begin} : \llbracket M \rrbracket : \mathsf{end}, \epsilon, \epsilon \rangle \rightarrow^2 \langle \llbracket M \rrbracket : \mathsf{end}, \llbracket V \rrbracket, \epsilon \rangle$, and $let\ x = V\ in\ M \rightarrow M[V/x]$. In the definition of R_H we see that the pair $(M[V/x], \langle \llbracket M \rrbracket : \mathsf{end}, \llbracket V \rrbracket, \epsilon \rangle) \in R_H$.

$(M\ to\ x\ in\ N, \langle \llbracket M \rrbracket : \mathsf{begin} : \llbracket N \rrbracket : \mathsf{end}, \epsilon, \epsilon \rangle) \in R_H$.

- $M\ to\ x\ in\ N \rightarrow M'\ to\ x\ in\ N$, using rule (CBPV-TO). If M' is a component of M in the pair $(M, \langle C, E, S \rangle) \in R_H$. We have that, per the induction hypothesis, there exists a $\langle C, E, S \rangle \rightarrow^* \langle C', E', S' \rangle$, such that $(M', \langle C', E', S' \rangle) \in R_H$.
- $\langle \llbracket M \rrbracket : \mathsf{begin} : \llbracket N \rrbracket : \mathsf{end}, \epsilon, \epsilon \rangle \rightarrow \langle \llbracket M'' \rrbracket : \mathsf{begin} : \llbracket N \rrbracket : \mathsf{end}, E, S \rangle$, and $M \rightarrow M'$ where $M''\sigma_E = M'$. Then, per the induction hypothesis, there exists a continuation $\langle \llbracket M'' \rrbracket : \llbracket V \rrbracket : \mathsf{apply}, E', S' \rangle \rightarrow^* \langle \llbracket M''' \rrbracket : \llbracket V \rrbracket : \mathsf{apply}, E'', S'' \rangle$, such that $(M', \langle \llbracket M''' \rrbracket : \llbracket V \rrbracket : \mathsf{apply}, E'', S'' \rangle) \in R_H$.

$((return\ V)\ to\ x\ in\ M, \langle \mathsf{return}(\llbracket V \rrbracket) : \mathsf{begin} : \llbracket M \rrbracket : \mathsf{end}, \epsilon, \epsilon \rangle) \in R_H$.

- $(return\ V)\ to\ x\ in\ M \rightarrow M[V/x]$, using rule (CBPV-RETURN). That is matched by $\langle \mathsf{return}(\llbracket V \rrbracket) : \mathsf{begin} : \llbracket M \rrbracket : \mathsf{end}, \epsilon, \epsilon \rangle \rightarrow^2 \langle \llbracket M \rrbracket : \mathsf{end}, \llbracket V \rrbracket, \epsilon \rangle$. In the definition of R_H we see that the pair $(M[V/x], \langle \llbracket M \rrbracket : \mathsf{end}, \llbracket V \rrbracket, \epsilon \rangle) \in R_H$.

- $\langle \mathsf{return}(\llbracket V \rrbracket) : \mathsf{begin} : \llbracket M \rrbracket : \mathsf{end}, \epsilon, \epsilon \rangle \rightarrow^2 \langle \llbracket M \rrbracket : \mathsf{end}, \llbracket V \rrbracket, \epsilon \rangle$, and $(return\ V)\ to\ x\ in\ M \rightarrow M[V/x]$. In the definition of R_H we see that the pair $(M[V/x], \langle \llbracket M \rrbracket : \mathsf{end}, \llbracket V \rrbracket, \epsilon \rangle) \in R_H$.

□

5 Conclusion

In this report, we have described, implemented, and proved the correctness of an abstract machine that recognises a core subset of the CBPV language described in [16]. As a part of developing the call-by-push-value abstract machine (CBPV AM) various implementations have taken place, such as the implementation of an interpreter and translator for call-by-push-value, call-by-value and call-by-name, as well as the implementation of the SECD machine and Krivine machine. This was done in order to support the development of the CBPV AM. Correctness of the CBPV AM was proved using bisimulation.

5.1 Discussion

As mentioned, the implementation focused on a core subset of the CBPV language, and therefore does not cover all the construct described in [16], such as pairs, match etc. The extension of the implementation to include these constructs would likely be a reasonably straightforward affair, since the implementation is in Haskell, the source code closely mirrors that of the semantics, transition rules, compilation etc.

A limitation of the proof is that it is a pen and paper proof, in the sense that there could remain ambiguity, and so by formalising the proof in a theorem prover, such as Coq, it is possible to eliminate any ambiguity and assert the correctness of the proof, and thereby the implementation. Formalising the proof would also enable extensions of the implementation to be machine-checked, and since the implementation is in Haskell, the path to formal verification using Coq shouldn't be too arduous, considering projects such as *hs-to-coq* exist and have been successful in translating Haskell code into Coq [25, 2, 11].

5.2 Future work

As mentioned in Section 3.6, and evident in Examples 6-9, the CBPV AM performs about twice as many reductions as the Krivine machine, and around 2.9 times more reductions compared to the SECD machine. It would be worth investigating this by performing a proper resource analysis of the CBPV AM, using the bisimulation proof as a basis, to identify the reason for the relatively poor performance and to illuminate potential avenues that it could be alleviated. One of these could be the ZAM machine presented in [15], which adapts the *push-enter* method from the Krivine machine and applies it to call-by-value,

thereby reducing the number of reductions compared to the *eval-apply* method of the SECD machine.

References

[1] Henk (Hendrik) Barendregt and E. Barendsen. "Introduction to lambda calculus". In: *Nieuw archief voor wisenkunde* 4 (Jan. 1984), pp. 337–372.

[2] Joachim Breitner et al. "Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code". In: *J. Funct. Program.* 31 (2021), e5. DOI: 10.1017/S0956796820000283. URL: https://doi.org/10.1017/S0956796820000283.

[3] N. G. de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". English. In: *Nederl. Akad. Wet., Proc., Ser. A* 75 (1972), pp. 381–392. ISSN: 0023-3358.

[4] *CakeML*. 2024. URL: https://cakeml.org/.

[5] *CompCert*. 2024. URL: https://compcert.org/.

[6] Stephan Diehl, Pieter H. Hartel, and Peter Sestoft. "Abstract machines for programming language implementation". In: *Future Gener. Comput. Syst.* 16.7 (2000), pp. 739–751. DOI: 10.1016/S0167-739X(99)00088-6. URL: https://doi.org/10.1016/S0167-739X(99)00088-6.

[7] *dotNET*. 2024. URL: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework.

[8] Paul Downen et al. "Kinds are calling conventions". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 104:1–104:29. DOI: 10.1145/3408986. URL: https://doi.org/10.1145/3408986.

[9] Eric Eide and John Regehr. "Volatiles are miscompiled, and what to do about it". In: *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*. Ed. by Luca de Alfaro and Jens Palsberg. ACM, 2008, pp. 255–264. DOI: 10.1145/1450058.1450093. URL: https://doi.org/10.1145/1450058.1450093.

[10] Yannick Forster et al. "Call-by-push-value in coq: operational, equational, and denotational theory". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi and Magnus O. Myreen. ACM, 2019, pp. 118–131. DOI: 10.1145/3293880.3294097. URL: https://doi.org/10.1145/3293880.3294097.

[11] *hs-to-coq*. 2024. URL: https://github.com/plclub/hs-to-coq?tab=readme-ov-file.

[12] Jean-Louis Krivine. "A call-by-name lambda-calculus machine". In: *High. Order Symb. Comput.* 20.3 (2007), pp. 199–207. DOI: 10.1007/S10990-007-9018-9. URL: https://doi.org/10.1007/s10990-007-9018-9.

[13] P. J. Landin. "The Mechanical Evaluation of Expressions". In: *Comput. J.* 6.4 (1964), pp. 308–320. DOI: 10.1093/COMJNL/6.4.308. URL: https://doi.org/10.1093/comjnl/6.4.308.

[14] Xavier Leroy. *From Krivine's machine to the Caml implementations*. 2005. URL: https://xavierleroy.org/talks/zam-kazam05.pdf.

[15] Xavier Leroy. *Functional programming languages. Part II abstract machines*. 2015. URL: https://xavierleroy.org/mpri/2-4/machines.pdf.

[16] Paul Blain Levy. "Call-by-push-value". In: *ACM SIGLOG News* 9.2 (2022), pp. 7–29. DOI: 10.1145/3537668.3537670. URL: https://doi.org/10.1145/3537668.3537670.

[17] Henry Mercer, Cameron Ramsay, and Neel Krishnaswami. "Implicit Polarized F: local type inference for impredicativity". In: *CoRR* abs/2203.01835 (2022). DOI: 10.48550/ARXIV.2203.01835. arXiv: 2203.01835. URL: https://doi.org/10.48550/arXiv.2203.01835.

[18] Robin Milner. "Functions as Processes". In: *Math. Struct. Comput. Sci.* 2.2 (1992), pp. 119–141. DOI: 10.1017/S0960129500001407. URL: https://doi.org/10.1017/S0960129500001407.

[19] Robin Milner et al. *The Definition of Standard ML*. The MIT Press, May 1997. ISBN: 9780262287005. DOI: 10.7551/mitpress/2319.001.0001. URL: https://doi.org/10.7551/mitpress/2319.001.0001.

[20] *MLKit*. 2024. URL: https://elsman.com/mlkit/.

[21] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction*. USA: John Wiley & Sons, Inc., 1992, pp. 73–83. ISBN: 0471929808.

[22] David Michael Ritchie Park. "Concurrency and Automata on Infinite Sequences". In: *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*. Ed. by Peter Deussen. Vol. 104. Lecture Notes in Computer Science. Springer, 1981, pp. 167–183. DOI: 10.1007/BFB0017309. URL: https://doi.org/10.1007/BFb0017309.

[23] Christine Rizkallah, Dmitri Garbuzov, and Steve Zdancewic. "A Formal Equational Theory for Call-By-Push-Value". In: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Jeremy Avigad and Assia Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 523–541. DOI: 10.1007/978-3-319-94821-8_31. URL: https://doi.org/10.1007/978-3-319-94821-8_31.

[24] Davide Sangiorgi. "On the origins of bisimulation and coinduction". In: *ACM Trans. Program. Lang. Syst.* 31.4 (2009), 15:1–15:41. DOI: 10.1145/1516507.1516510. URL: https://doi.org/10.1145/1516507.1516510.

[25] Antal Spector-Zabusky et al. "Total Haskell is Reasonable Coq". In: *CoRR* abs/1711.09286 (2017). arXiv: 1711.09286. URL: http://arxiv.org/abs/1711.09286.

[26] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532. URL: https://doi.org/10.1145/1993498.1993532.