Standardisation and Call-By-Push-Value

Dion Sean Hansen

Department of Computer Science, Aalborg University

January 12, 2024

Standardisation of lambda calculus has many benefits, from proving an equational theory to implementing an interpreter for the language. This paper presents an overview of simple untyped lambda calculus, the evaluation strategies call-by-value (CBV) and call-by-name (CBN), and an in-depth look at a call-by-value standardisation proof by Karl Crary [2]. Followed is an introduction of call-by-push-value (CBPV), its many properties and features, such as translation to and from CBV and CBN, as well as the work that has gone into proving equational theories for CBPV.

Acknowledgment

This work was done as a study abroad postgraduate at Melbourne University. I would like to express gratitude to my supervisors Christine Rizkallah and Hans Hüttel.

1 Introduction

Lambda calculus has many variants, that vary in the available constructs, the addition of types, and the semantics or evaluation strategy. In this paper we will consider the untyped lambda calculus, as seen in Definition 2, with the purpose of discussing and reasoning about the different evaluation strategies and their respective expressiveness. The general introduction to lambda calculus can be found in [1].

This paper will specifically consider call-by-value (CBV), call-by-name (CBN) and the subsuming paradigm call-by-push-value (CBPV). Call-by-value and call-by-name have been the subject of much research [6, 8, 2] and also form the basis of very influential programming languages, such as C and Algol 60 respectively. Call-by-push-value is an idealised calculus for functional and imperative programming first described by Levy [5]. It enables the translation of call-by-value and call-by-name to call-by-push-value and vice versa, while preserving every known form of semantics [5]. This makes it possible to work only with call-by-push-value, and not need to replicate work for call-by-value and call-by-name, while achieving the same results [5].

1.1 Related work

The work of Plotkin [6], Takahashi [8], as well as Crary [2] is largely centred around proving two main theorems of lambda-calculus [4]. One is the Church-Rosser theorem, also referred to as the property of confluence, seen in Definition 1. The second is the standardisation theorem, which expresses that left-most outermost reductions are a terminating strategy, which will be defined later in Theorem 1.

Definition 1 (Confluence). *A reduction relation $\rightarrow$, is confluent if for every M there exists an N, such that if $M \rightarrow^* P$ and $M \rightarrow^* Q$ then $P \rightarrow^* N$ and $Q \rightarrow^* N$.*

Plotkin proved standardisation for call-by-value and Takahashi proved it for call-by-name [6, 8]. Crary provided a simplified proof for call-by-value using Takahashi's method, namely the notion of parallel reduction [2], which Takahashi had refined from Tait-Martin-Löf [8].

Rizkallah et al. provided and proved soundness of an equational theory for a variant of call-by-push-value with the goal of studying verification of compiler optimisations [7]. Forster et al. positions their work as an extension of Levy and Rizkallah et al's work [3]. Forster et al. formalised weak and strong operational semantics for effect-free call-by-push-value, and verified adequacy of reduction for the denotational semantics [3]. Furthermore they prove confluence of strong reduction for CBPV using Takahashi's method, and soundness of their equational theory [3].

The work of Crary [2], Rizkallah et al. [7], and Forster et al. [3] were all formally verified using proof assistants. Crary used Twelf, and Rizkallah et al. and Forster et al. used Coq.

1.2 Call-by-value and call-by-name

This section will present and discuss call-by-value and call-by-name. The semantics for the left-to-right variant of call-by-value and call-by-name can be seen in Table 1 and Table 2. To simplify the semantics for call-by-value, an addition has been added to the formation rules as seen in Definition 3, which is used to denote values in the call-by-value semantics.

Definition 2 (Terms). *The formation rules for untyped lambda calculus is as follows*

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

Definition 3 (Values). *Values have the following form*

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

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

$$\text{(CBV-2)} \qquad \frac{N \to N'}{V\ N \to V\ N'}$$

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

Table 1: Call-by-value semantics

The two evaluation strategies differ in their expressiveness and their use case. Call-by-value requires that the argument N, in an application MN, is a value V. Whereas call-by-name simply substitutes the unevaluated argument into the function body.

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

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

Table 2: Call-by-name semantics

While both strategies are expressive, they are not equivalent. Consider the expression $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$. In a call-by-value semantics, we have

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

since via rule (CBV-1) we have $((\lambda x.xx)(\lambda x.xx)) \to ((\lambda x.xx)(\lambda x.xx))$, which is an infinite reduction sequence

$$M \to M \to \cdots$$

whereas we, in the call-by-name semantics, have a terminating reduction sequence

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

In the above example, call-by-name results in a terminating sequence while call-by-value does not. This means that the two evaluation strategies are not equivalent. Plotkin demonstrates this [6], which makes the work of Levy with call-by-push-value so interesting.

1.3 Contextual equivalence

Plotkin [6] and Crary [2] both consider the equivalence of terms in the language, particularly contextual equivalence and the property of adequacy of reduction.

Two lambda terms are said to be contextually equivalent, if they exhibit identical behaviour with respect to *termination*, regardless of the context they are placed in. While adequacy of reduction, says that contextual equivalence respects evaluation. Where *termination* is defined as seen in Definition 4.

Definition 4 (Termination)**.** *Let M be a lambda term. We write $M \downarrow$ if there exists a terminating reduction sequence*

$$M \to \cdots \to N \not\to$$

Before defining contextual equivalence and adequacy of reduction, it is necessary to define what a context is, which can be seen in Definition 5.

Definition 5 (Context)**.** *A context is defined as follows*

$$C ::= [\,]N \mid M[\,] \mid \lambda x.[\,]$$

Where $[\,]$ denotes the "hole" in the context, and $C[M]$ denotes the hole filled by M.

Now we can define contextual equivalence, as seen in Definition 6.

Definition 6 (Contextual equivalence)**.** *Let M and N be lambda terms. We say that $M \equiv N$, if for any context C it holds that $C[M] \downarrow$ if and only if $C[N] \downarrow$.*

Lastly the definition of adequacy of reduction can be seen in Definition 7, and says that contextual equivalence is preserved by evaluation.

Definition 7 (Adequacy of reduction)**.** *A relation R is adequate when $M\ R\ M'$ implies that M halts if and only if M' halts.*

In proving two programs contextually equivalent, you need to prove that they agree on termination in any context. Example 1 demonstrates two terms, which are not contextually equivalent.

Example 1 (Contextual equivalence)**.** *As defined in Definition 6, for two terms to be contextually equivalent they need to agree on termination in any context. Therefore, the way to prove that two terms are not contextually equivalent, is to provide a context where the terms diverge, i.e. a context where the two terms do not agree on termination. Consider terms M_1 and M_2, and the context C.*

$$M_1 = \lambda x.xx$$

$$M_2 = \lambda x.x$$

Context: $C = (\lambda x.xx)[\,]$, and so we have the following reduction sequences

$$C[M_1] \to (\lambda x.xx)(\lambda x.xx) = C[M_1]$$

$$C[M_2] \to^2 \lambda x.x \not\to$$

where $C[M_1] \not\downarrow$, but $C[M_2] \downarrow$. It then follows that $M_1 \not\equiv M_2$, as we have found a context where they diverge.

When we want to show that $M \not\equiv N$, then it is enough to find a single context C such that $C[M] \downarrow$, but $C[N] \not\downarrow$ or opposite.

However, if you want to show that $M_1 \equiv M_2$, you need to prove that $C[M] \downarrow$ if and only if $C[N] \downarrow$, which requires a proof by structural induction on C.

2 Standardisation for call-by-value

This chapter will present Karl Crary's proof of call-by-value standardisation for untyped lambda calculus. Therefore, unless otherwise specified, the provided definitions and more were sourced from Crary [2], and the formation rules for the untyped lambda calculus can be found in Section 1.2. While the theorem itself is not a new result, as it was proven by Plotkin back in the 1970's [6], Crary manages to make it very concise and direct. He does this by adapting Takahashi's method, which employs the notion of parallel reduction [8].

This section will provide the reduction semantics that Crary utilises for his proof, as well as formally introduce standardisation and the definition of a standard reduction. Having defined and motivated why standardisation is worth pursuing, we will explore the proof sketch and the specific lemmas that Crary employs in proving standardisation for call-by-value.

2.1 Reduction semantics

Crary makes use of three reduction semantics, the definition of which can be seen in Definition 8, and the rules in Tables 3-5.

Definition 8 (Reduction semantics)**.** *The evaluation relation has reductions of the form*

$$M \mapsto N$$

The rules defining $\mapsto$ are shown in Table 3. The parallel reduction relation has reductions of the form

$$M \Longrightarrow N$$

The rules defining $\Longrightarrow$ are shown in Table 4. The internal reduction relation has reductions of the form

$$M \overset{int}{\Longrightarrow} N$$

The rules defining $\overset{int}{\Longrightarrow}$ are shown in Table 5.

Evaluation, or $\mapsto$, is the left-to-right call-by-value semantics, which was presented earlier in Section 1.2, and have been renamed to more closely follow Crary's proof. The rules (EVAL-1) and (EVAL-2) enforce the left-to-right evaluation order by requiring that in an application MN, M will be evaluated to a value V before evaluating N. The last evaluation rule, (EVAL-3), also known as

$$\text{(Eval-1)} \qquad \frac{M \mapsto N}{M\ N \mapsto M'\ N}$$

$$\text{(Eval-2)} \qquad \frac{N \mapsto N'}{V\ N \mapsto V\ N'}$$

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

Table 3: Evaluation [2]

beta-reduction, substitutes the argument into the lambda body, but when the argument has been evaluated to a value V.

$$\text{(Par-1)} \qquad \frac{}{M \Longrightarrow M}$$

$$\text{(Par-2)} \qquad \frac{M \Longrightarrow N}{\lambda x.M \Longrightarrow \lambda x.N}$$

$$\text{(Par-3)} \qquad \frac{M \Longrightarrow M' \quad N \Longrightarrow N'}{M\ N \Longrightarrow M'N'}$$

$$\text{(Par-4)} \qquad \frac{M \Longrightarrow M' \quad V \Longrightarrow V'}{(\lambda x.M)V \Longrightarrow [V'/x]M'}$$

Table 4: Parallel reduction [2]

Parallel reduction, or $\Longrightarrow$, enables reduction under abstractions as well as reduction in parallel via rules (Par-2) and (Par-3) respectively. Rule (Par-4) is the beta-reduction rule for parallel reduction, and, as with $\mapsto$, it requires the argument to be a value V. Lastly rule (Par-1) permits sequential reduction, when one side cannot be reduced further, thereby permitting (Par-3) and (Par-4) to be applied to all cases.

Internal parallel reduction, or $\overset{\text{int}}{\Longrightarrow}$, is parallel reduction where the beta-reduction rule is removed. The relation enables reduction under abstractions via rule (IntPar-2), as well as reduction in parallel on an application via rules (IntPar-3) and (IntPar-4). However, it cannot perform substitutions due to the lack of the beta-reduction rule.

Example 2 ($\Longrightarrow$ and $\overset{\text{int}}{\Longrightarrow}$)**.** *The key difference between* $\Longrightarrow$ *and* $\overset{int}{\Longrightarrow}$*, is that* $\overset{int}{\Longrightarrow}$ *cannot perform beta-reduction on an application* MN*, but if* M *is itself an application,* $M = RS$*, then it allows for beta-reduction in the argument* N *via rule* (IntPar-3)*. This is clear in the following internal parallel reduction sequence.*

$$\text{(IntPar-1)} \qquad \frac{}{M \overset{\text{int}}{\Longrightarrow} M}$$

$$\text{(IntPar-2)} \qquad \frac{M \Longrightarrow N}{\lambda x.M \overset{\text{int}}{\Longrightarrow} \lambda x.N}$$

$$\text{(IntPar-3)} \qquad \frac{M \text{ nonvalue} \quad M \overset{\text{int}}{\Longrightarrow} M' \quad N \Longrightarrow N'}{M\ N \overset{\text{int}}{\Longrightarrow} M'\ N'}$$

$$\text{(IntPar-4)} \qquad \frac{M \overset{\text{int}}{\Longrightarrow} M' \quad N \overset{\text{int}}{\Longrightarrow} N'}{M\ N \overset{\text{int}}{\Longrightarrow} M'\ N'}$$

Table 5: Internal parallel reduction [2]

$$\begin{aligned}((\lambda z.z)(21+22))((\lambda x.(\lambda y.y-x)1)2) &\overset{int}{\Longrightarrow} ((\lambda z.z)43)((\lambda y.y-2)1)\\ &\overset{int}{\Longrightarrow} ((\lambda z.z)43)(1-2)\\ &\overset{int}{\Longrightarrow} ((\lambda z.z)43)(-1)\\ &\overset{int}{\not\Longrightarrow}\end{aligned}$$

While $\overset{int}{\Longrightarrow}$ is able to perform beta-reduction in N for an expression MN where $M = RS$, it is not able to reduce in M. In the other hand, the following parallel reduction sequence demonstrates that $\Longrightarrow$ can.

$$\begin{aligned}((\lambda z.z)(21+22))((\lambda x.(\lambda y.y-x)1)2) &\Longrightarrow ((\lambda z.z)43)((\lambda y.y-2)1)\\ &\Longrightarrow ((\lambda z.z)43)(1-2)\\ &\Longrightarrow (43)(-1)\\ &\Longrightarrow 42\end{aligned}$$

2.2 Standard reductions

The definition of a standard reduction can be seen in Definition 9.

Definition 9 (Standard reduction)**.**

1. *M is a standard reduction.*
2. *If $M_0 \to M_1$ and $M_1, \ldots, M_m$ is a standard reduction, then $M_0, M_1, \ldots, M_m$ is a standard reduction.*
3. *If $M_1, \ldots, M_m$ is a standard reduction, then $\lambda x.M_1, \ldots, \lambda x.M_m$ is a standard reduction.*

4. If $M_1, \dots, M_m$ and $N_1, \dots, N_n$ are standard reductions, then $M_1N_1, \dots, M_mN_1, \dots, M_mN_n$ is a standard reduction.

Rule 1. states that any term is in itself, a standard reduction, Rule 2. states that any term resulting from a standard reduction is also a standard reduction, that is, standard reductions are transitive. Rule 3. states that standard reductions can take place under abstraction, and the final rule, Rule 4. states that standard reductions are to take place in a left-to-right manner.

The standardisation theorem, seen in Theorem 1, says that for any lambda term M it holds that if $M \Longrightarrow^* N$, then we have $M \rightarrow_s^* N$, where $\rightarrow_s^*$ denotes a standard reduction. That is, $\rightarrow_s^*$ is equivalent to $\Longrightarrow^*$.

Theorem 1 (Standardisation)**.** *If $M \Longrightarrow^* N$ then there exists a standard reduction of the form*

$$M \rightarrow_s^* N$$

Standardisation is important because it alleviates the labour required in proving properties of reductions in the language, such as the equivalence between programs, which is a extremely useful. A practical use of proving equivalence between programs is in compiler optimisations, since the ability to reason about the equivalence between two pieces of code, enables the compiler to replace inefficient instructions with more efficient instructions. While ensuring that the behaviour of the program is preserved. Contextual equivalence, which was presented in Section 1.3, is one way of defining an equivalence relation over lambda terms.

Theorem 1, tells us that it is enough to consider only standard reductions, and the proof of Theorem 1 details a strategy for transforming a non-standard reduction sequence into a standard one, which can be seen in Examples 3 and 4.

It is worth noting, that standard reduction does not only provide a tool to assist in proving program equivalence, but indeed also provides a way to execute programs. In other words, in order to execute any programs in the language, all you need to implement is an interpreter that can execute standard reductions.

Example 3 (A non-standard reduction sequence)**.** *The following is not an example of a standard reduction.*

$$\begin{aligned}(\lambda z.(\lambda q.21 + 22 + z))((\lambda x.(\lambda y.y - x)1)2) &\Longrightarrow (\lambda z.(\lambda q.43 + z))((\lambda x.1 - x)2)\\ &\Longrightarrow (\lambda z.(\lambda q.43 + z))(1 - 2)\\ &\Longrightarrow (\lambda z.(\lambda q.43 + z))(-1)\\ &\Longrightarrow \lambda q.43 + (-1)\\ &\Longrightarrow \lambda q.42\end{aligned}$$

because of two particular reductions. The internal reductions $(\lambda z.21 + 22 + z) \Longrightarrow (\lambda z.43 + z)$ and $((\lambda x.(\lambda y.y - x)1)2) \Longrightarrow ((\lambda x.1 - x)2)$ are both allowed under standard reduction, but they are both performed before evaluation has been *exhausted. They therefore go against the order set for standard reductions, where evaluation is performed before internal reductions. Furthermore, the reductions are performed in parallel, which is not allowed for standard reductions, as they are to take place in a left-to-right manner.*

Example 4 (Transforming a non-standard reduction sequence into a standard reduction)**.** *Transforming the reduction sequence of Example 3, into a standard reduction would mean postponing the internal reductions to after evaluation has been exhausted. Below is the standard reduction form of the example sequence from Example 3.*

$$\begin{aligned}(\lambda z.(\lambda q.21 + 22 + z))((\lambda x.(\lambda y.y - x)1)2) &\rightarrow_s^* (\lambda z.(\lambda q.21 + 22 + z))(\lambda y.y - 1)2) \\ &\rightarrow_s^* (\lambda z.(\lambda q.21 + 22 + z))(1 - 2) \\ &\rightarrow_s^* (\lambda z.(\lambda q.21 + 22 + z))(-1) \\ &\rightarrow_s^* \lambda q.21 + 22 + (-1) \\ &\rightarrow_s^* \lambda q.42\end{aligned}$$

The internal reduction of $(\lambda z.21 + 22 + z) \rightarrow_s^* (\lambda z.43 + z)$ *does not take place until the end, as the internal reduction is postponed. Furthermore the beta-reduction of* λx *and* λy *follows from the outermost term, so* λx *is reduced before* λy*. Thus the reduction sequence is now a standard reduction.*

2.3 Proof of standardisation

The final result of Crary's paper, is the proof of the standardisation theorem, Theorem 1, but also that of *adequacy of reduction*, seen in Theorem 2. Adequacy of reduction, which was defined in Definition 7, states that contextual equivalence respects reduction, here $\Longrightarrow^*$. That is, if $M \Longrightarrow^* N$ then M and N are contextually equivalent, $C[M] \equiv C[N]$ for any C. In proving Theorem 1 and 2, he employs Lemma 1, also called the *Bifurcation* lemma.

Lemma 1 (Bifurcation)**.** *If* $M \Longrightarrow^* N$ *then* $M \mapsto^* P$ *and* $P \overset{int}{\Longrightarrow}^* N$*.*

Lemma 1, says that any parallel reduction sequence can be bifurcated into an evaluation sequence followed by an internal parallel reduction sequence, and it is worth noting that Lemma 1 actually provides the structure of a standard reduction. It is essential for proving Theorem 1, which in turn embodies a strategy for converting a reduction into a standard reduction.

Theorem 1 (Standardisation)**.** *If* $M \Longrightarrow^* N$ *then there exists a standard reduction of the form*

$$M \rightarrow_s^* N$$

Proof. By structural induction on N and Lemma 1, $M = M_0 \mapsto \cdots \mapsto M_m \overset{\text{int}}{\Longrightarrow}^* N$. We have the following cases on N.

1. Suppose $N = x$. Then $M_m = x$. So $M = M_0, \dots, M_m = N$ is a standard reduction.

2. Suppose $N = \lambda x.Q$. Then $M_m = \lambda x.P$ and $P \Longrightarrow^* Q$. By induction, there exists a standard reduction $P = P_0, \dots, P_p = Q$. So $M = M_0, \dots, M_m = \lambda x.P_0, \dots, \lambda x.P_p = N$ is a standard reduction.

3. Suppose $N = RS$. Then $M_m = PQ$ and $P \Longrightarrow^* R$ and $Q \Longrightarrow^* S$. By induction, there exists standard reductions $P = P_0, \dots, P_p = R$ and $Q = Q_0, \dots, Q_q = S$. So $M = M_0, \dots, M_m = P_0Q_0, \dots, P_pQ_0, \dots, P_pQ_q = N$ is a standard reduction.

□

Theorem 2 (Adequacy of Reduction)**.** *If $M \Longrightarrow^* N$ then M halts if and only if N halts.*

Proof. Suppose $N \mapsto^* P$, for some value P. Then $M \Longrightarrow^* P$. By Lemma 1, $M \mapsto^* P' \overset{\text{int}}{\Longrightarrow}^* P$. Since P is a value, so is P'. Thus M halts.

Conversely, suppose $M \mapsto^* P$ for some value P. Then $M \Longrightarrow^* P$. By Lemma 2, there exists Q such that $N \Longrightarrow^* Q$ and $P \Longrightarrow^* Q$. By Lemma 1, $N \mapsto^* Q' \overset{\text{int}}{\Longrightarrow}^* Q$. Since P is a value, so is Q, and then so is Q'. Thus N halts. □

The proof for Theorem 2 employs Lemma 1 as well, but requires an additional lemma, namely Lemma 2. Lemma 2 states two properties of $\Longrightarrow$, 1. that in beta-reduction the argument and body can be reduced in parallel, and 2. that $\Longrightarrow$ is *confluent*, where confluence is defined as seen in Definition 1.

Lemma 2 (Parallel reduction)**.**

1. *If $M \Longrightarrow M'$ and $N \Longrightarrow N'$ then $[N/x]M \Longrightarrow [N'/x]M'$.*

2. *If $M \Longrightarrow^* P$ and $M \Longrightarrow^* Q$ then there exists N such that $P \Longrightarrow^* N$ and $Q \Longrightarrow^* N$.*

3 Call-by-push-value

Call-by-push-value, or CBPV, is an idealised calculus first discovered by Paul Levy [5]. It is a variant of simply typed lambda calculus, which incorporates computational effects, and therefore combines functional and imperative programming. A very noteworthy aspect of CBPV is that it subsumes CBV and CBN. Here subsume means that CBV and CBN cannot only be translated to and from CBPV, but that the transformation preserves every known form of semantics, which suggests that from a semantic viewpoint CBV and CBV are but sub-systems of CBPV [5].

CBPV has many variants, and since this report does not require special consideration, the choice is arbitrary. So while the classic CBPV is typed, as presented by Levy [5], I will instead present an untyped variant of CBPV presented in Rizkallah et al. [7].

This section will present the syntax and semantics of CBPV, how properties for the lambda calculus extend to CBPV, and finally demonstrate translations of CBV and CBN to CBPV.

3.1 Syntax and semantics

A fundamental idea presented in CBPV is the distinction between *values* and *computations*, or as Levy [5] puts it "A value is, a computation does". This mantra can be seen in the syntax in Definition 10 and 11.

Definition 10 (Values)**.** *The formation rules for values in call-by-push-value is as follows*

$$V ::= x \mid n \mid \textit{thunk}\ M$$

Definition 11 (Computations)**.** *The formation rules or computations, in call-by-push-value are as follows*

$$M, N ::= \textit{force}\ V \mid \textit{letrec}\ x_1 = M_1, \ldots, x_n = M_n\ \textit{in}\ N \mid \textit{prd}\ V \mid$$

$$M\ \textit{to}\ x\ \textit{in}\ N \mid \lambda x.M \mid V.M \mid V_1 \oplus V_2 \mid \textit{if0}\ V\ M_1\ M_2$$

A value is:

- a variable x
- a natural number n
- or a thunk M, where a *thunk* is a suspended computation, such as a function.

A computation is:

- a force V, which runs a suspended computation, i.e. a thunk
- letrec, a mutually recursive definition, which is present in the variant used by Rizkallah et al. [7], and its formalisation is a central part of their contribution
- M to x in N, where M is first evaluated to a computation of form prd V (read *produce* V), and then bound to x in N, $N[V/x]$
- λ-abstraction $\lambda x.M$ and application $V.M$ (where V is the argument and M is the function)
- binary arithmetic operations $V_1 \oplus V_2$, where $\oplus$ is addition, subtraction and less-than

- and lastly conditional statements if0 V M_1 M_2, where M_1 runs if V is 0, and otherwise M_2

While the new constructs have their uses, the important ones, which make up the essence of CBPV, are *thunk*, *force*, *produce* (also referred to as *return*), and the *to* construct. The semantics of CBPV can be seen in Tables 6 and 7. The semantics rely on two relations, the small-step evaluation relation, denoted by $\rightarrow$, and the auxiliary relation, denoted by $\rightsquigarrow$ [7]. The auxiliary relation is introduced by Rizkallah et al. to unroll a letrec term to a prd-term or a lambda-term [7].

$$\text{(CBPV-1)} \qquad \frac{}{M \rightsquigarrow M}$$

$$\text{(CBPV-2)} \qquad \frac{N \rightsquigarrow N' \quad N' \rightarrow M}{N \rightarrow M}$$

$$\text{(CBPV-3)} \qquad \frac{}{\text{if0 } 0\ M_1\ M_2 \rightarrow M_1}$$

$$\text{(CBPV-4)} \qquad \frac{}{\text{if0 } n\ M_1\ M_2 \rightarrow M_2 (n \neq 0)}$$

$$\text{(CBPV-5)} \qquad \frac{M \rightsquigarrow \lambda x.N}{V.M \rightarrow \{V/x\}N}$$

$$\text{(CBPV-6)} \qquad \frac{M \rightarrow M'}{V.M \rightarrow V.M'}$$

$$\text{(CBPV-7)} \qquad \frac{\{\overline{\text{thunk } (\text{letrec } \overline{x_i = M_i}^{i} \text{ in } M_i)/x_i}^{i}\}N \rightsquigarrow N'}{\text{letrec } \overline{x_i = M_i}^{i} \text{ in } N \rightsquigarrow N'}$$

$$\text{(CBPV-8)} \qquad \frac{\{\overline{\text{thunk } (\text{letrec } \overline{x_i = M_i}^{i} \text{ in } M_i)/x_i}^{i}\}N \rightarrow N'}{\text{letrec } \overline{x_i = M_i}^{i} \text{ in } N \rightarrow N'}$$

Table 6: Semantics of CBPV 1/2 [7]

Table 6 provides the rules for the general constructs, such as conditional statements, applications, arithmetic operations, as well as various step-wise reductions. It also presents the rules of the letrec construct, but since it is a construct of the specific CBPV variant, and not integral to CBPV, it will not be discussed further. Table 7, covers the rules of *force*, *thunk*, *produce*, and *to*.

3.2 Properties of CBPV

Various properties of untyped lambda calculus were presented in Section 1 and 2. Properties such as confluence, adequacy of reduction, contextual equivalence and standardisation.

$$(\text{CBPV-9}) \qquad \frac{}{\text{force}(\text{thunk } M) \to M}$$

$$(\text{CBPV-10}) \qquad \frac{M \rightsquigarrow \text{prd } V}{M \text{ to } x \text{ in } N \to \{V/x\}N}$$

$$(\text{CBPV-11}) \qquad \frac{M \to M'}{M \text{ to } x \text{ in } N \to M' \text{ to } x \text{ in } N}$$

$$(\text{CBPV-12}) \qquad \frac{M \rightsquigarrow (n_1 \oplus n_2)}{M \text{ to } x \text{ in } N \to \{n_1 [\![\oplus]\!] n_2/x\}N}$$

Table 7: Semantics of CBPV 2/2 [7]

The properties of adequacy of reduction and contextual equivalence remains unchanged, and are central components of Rizkallah et al. [7] and Forster et al. [3], as they both present and prove soundness of an equational theory, where an equational theory is a set of axioms in the form of $M \equiv N$ and soundness can be seen in Definition 12. Standardisation has, to the extent of my knowledge, not been proven for CBPV.

Definition 12 (Soundness)**.** *Let R be a binary relation over terms. We can then say that R is sound, with respect to contextual equivalence, if for all terms M and N that are related by $M\ R\ N$ it holds that $M \equiv N$.*

3.3 Translation of CBV and CBN to CBPV

The rules for translating CBV and CBN into CBPV, and vice versa, can be seen in Table 8 and 9.

CBV term	CBPV computation
x	prd x
inl M	M to x. prd inl x
λx.M	prd thunk λx.M
MN (operator-first)	M to f. N to x. (force f)x
MN (operand-first)	N to x. M to f. (force f)x

Table 8: Decomposition of CBV into CBPV [5]

Example 5 and 6 demonstrate the translation of the example program, presented in Section 1.2, to CBPV.

Example 5 (Translating CBV to CBPV)**.** *Using the translations for CBV in Table 5 we get the following translation:*

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

Table 9: Decomposition of CBN into CBPV [5]

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \to (\textit{force } (\lambda x.xx)(\lambda x.xx))(\lambda x.y)$$

It is necessary to point out that there exists two translations for an application in CBV, as they can either be evaluated operator-first or operand-first, hence two translations [5]. The operator-first translation has been chosen arbitrarily.

Executing the translation using the semantics of CBPV we have:

$$(\textit{force } (\lambda x.xx)(\lambda x.xx))(\lambda x.y) \to (\textit{force } (\lambda x.xx)(\lambda x.xx))(\lambda x.y)$$

force runs the function $(\lambda x.xx)(\lambda x.xx)$, but as witnessed in Section 1.2, that is a non-terminating reduction sequence in CBV, i.e. $M \to M \to \cdots$. Thus the translation preserves the CBV semantics for this particular example.

Example 6 (Translating CBN to CBPV)**.** *Using the translations for CBN in Table 5 we get the following translation:*

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \to (\lambda x.y)(\textit{thunk } (\lambda x.xx)(\lambda x.xx))$$

Then executing the translation using the semantics of CBPV we have:

$$(\lambda x.y)(\textit{thunk } (\lambda x.xx)(\lambda x.xx)) \to y$$

thunk suspends the computation of $(\lambda x.xx)(\lambda x.xx)$, and thus the application takes place before the computation of the argument, as is expected with the semantics of CBN. The term is also a terminating sequence in CBN, which was also the case in Section 1.2. Thus the translation preserves the CBN semantics for this particular example.

Both the Example 5 and 6 demonstrate that, for the particular example, the semantics of CBV and CBN are preserved by the translation to CBPV.

4 Conclusion

This paper presents an overview of the untyped lambda calculus and a set of evaluation strategies with the purpose of studying the strategies, their properties, and their standardisation. Section 1, introduces the untyped lambda calculus, related work with respect to call-by-value (CBV), call-by-name (CBN), and call-by-push-value (CBPV). Furthermore it presents a comparison between CBV and CBN, and finally presents key terminology for reasoning about lambda terms and their equivalence. Section 2, presents an in-depth look at Crary's proof of CBV standardisation [2], the key definitions, and lemmas needed to understand his work. Section 3, presents CBPV, its syntax and semantics, as well as CBPV's key feature of translation to and from, CBV and CBN.

This paper serves as an introduction into evaluation strategies for untyped lambda calculus, and the important work of standardisation and proving equivalence between programs. While much work has been put into formalisations and proofs of CBV and CBN. The application and properties of CBPV are still an active area of research, as seen in [7] and [3].

Rizkallah et al. and Forster et al. both suggest further investigating whether adapting and scaling Crary's proof method to CBPV could result in simpler proofs for their respective equational theories [7, 3]. A simpler proof is a valuable contribution, as it makes it easier to extend, maintain, and understand their formalisations, which is important in order for ones work to be more applicable and reach a wider audience. That is not to say, that standardisation of CBPV itself is not a desirable goal, as it has currently not been proved, and would therefore be a new result.

Using the standardisation proof of Crary for CBV [2], it would be interesting to see if one could create a program in Haskell, Coq or the like, that could construct standard reductions. That is, a tool that could automatise the transformation of a reduction into a standard reduction.

It could also be interesting to see what changes would be needed if one were to extend the language to include more complex constructs, such as Rizkallah et al. [7] did with their letrec construct. Though it is evident, that it took a considerable amount of work to reach their results.

References

[1] Henk Barendregt. "Introduction to Generalized Type Systems". In: *J. Funct. Program.* 1.2 (1991), pp. 125–154. DOI: 10.1017/S0956796800020025. URL: https://doi.org/10.1017/s0956796800020025.

[2] Karl Crary. "A Simple Proof of Call-by-Value Standardization". In: 2009. URL: https://api.semanticscholar.org/CorpusID:14327563.

[3] 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.

[4] Georges Gonthier, Jean-Jacques Lévy, and Paul-André Melliès. "An abstract standardisation theorem". In: *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992.* IEEE Computer Society, 1992, pp. 72–81. DOI: `10.1109/LICS.1992.185521`. URL: `https://doi.org/10.1109/LICS.1992.185521`.

[5] 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`.

[6] Gordon D. Plotkin. "Call-by-Name, Call-by-Value and the lambda-Calculus". In: *Theor. Comput. Sci.* 1.2 (1975), pp. 125–159. DOI: `10.1016/0304-3975(75)90017-1`. URL: `https://doi.org/10.1016/0304-3975(75)90017-1`.

[7] 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`.

[8] Masako Takahashi. "Parallel Reductions in lambda-Calculus". In: *Inf. Comput.* 118.1 (1995), pp. 120–127. DOI: `10.1006/INCO.1995.1057`. URL: `https://doi.org/10.1006/inco.1995.1057`.