Classical Call by Push Value

Emma Tye
Supervised by Steffen van Bakel

June 23, 2020

Classical Call by Push Value

Emma Tye
Supervised by Steffen van Bakel

June 23, 2020

λ-calculus

$$s, t ::= x \mid \lambda x.s \mid st$$

λ-calculus
$s, t ::= x \mid \lambda x.s \mid st$

1. We'll start by recalling the λ-calculus. It is formed of variables, λ abstractions (functions) and application
2. The main reduction rule is function application, where you replace a variable with a term
3. Reduction can occur anywhere within a term, so if a subterm includes function application we can reduce that inside the term as a whole. This means there are multiple ways to reduce a term.

λ-calculus

$$s, t ::= x \mid \lambda x.s \mid st$$
$$(\lambda x.s)t \rightsquigarrow s\left[{}^{t}/_{x}\right]$$

λ-calculus
$s, t ::= x \mid \lambda x.s \mid st$
$(\lambda x.s)t \rightsquigarrow s\left[{}^{t}/_{x}\right]$

1. We'll start by recalling the λ-calculus. It is formed of variables, λ abstractions (functions) and application
2. The main reduction rule is function application, where you replace a variable with a term
3. Reduction can occur anywhere within a term, so if a subterm includes function application we can reduce that inside the term as a whole. This means there are multiple ways to reduce a term.

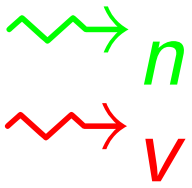
λ-calculus

$$s, t ::= x \mid \lambda x.s \mid st$$
$$(\lambda x.s)t \rightsquigarrow s\left[{}^{t}/_{x}\right]$$
$$((\lambda x.s_1)s_2)((\lambda y.t_1)t_2)$$

λ-calculus
$s, t ::= x \mid \lambda x.s \mid st$
$(\lambda x.s)t \rightsquigarrow s\left[{}^{t}/_{x}\right]$
$((\lambda x.s_1)s_2)((\lambda y.t_1)t_2)$

1. We'll start by recalling the λ-calculus. It is formed of variables, λ abstractions (functions) and application
2. The main reduction rule is function application, where you replace a variable with a term
3. Reduction can occur anywhere within a term, so if a subterm includes function application we can reduce that inside the term as a whole. This means there are multiple ways to reduce a term.

CBN & CBV



CBN & CBV

1. So we have reduction strategies to tell us which reductions to run. cbn and cbv are two useful and important reduction strategies

CBN & CBV

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

CBN & CBV
$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$

1. Let's use this λ-calculus example to see how they differ. It has a function which discards its first argument at the top level, and an infinite computation begin applied to it
2. In call-by-name, we run the outermost, leftmost computation
3. So we forget the infinite computation and don't even attempt to run it

CBN & CBV

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

CBN & CBV
$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$

1. Let's use this λ-calculus example to see how they differ. It has a function which discards its first argument at the top level, and an infinite computation begin applied to it
2. In call-by-name, we run the outermost, leftmost computation
3. So we forget the infinite computation and don't even attempt to run it

CBN & CBV

$$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$$
$$\rightsquigarrow \lambda y.y$$

CBN & CBV
$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$
$\rightsquigarrow \lambda y.y$

1. Let's use this λ-calculus example to see how they differ. It has a function which discards its first argument at the top level, and an infinite computation begin applied to it
2. In call-by-name, we run the outermost, leftmost computation
3. So we forget the infinite computation and don't even attempt to run it

CBN & CBV

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

CBN & CBV
$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$

1. In call-by-value, we run the innermost, rightmost computation
2. So the infinite computation runs, reducing to itself, so the whole term reduces to itself
3. So we run it again, and again, and so on

CBN & CBV

$$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$$
$$\rightsquigarrow(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$$

CBN & CBV
$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$
$\rightsquigarrow(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$

1. In call-by-value, we run the innermost, rightmost computation
2. So the infinite computation runs, reducing to itself, so the whole term reduces to itself
3. So we run it again, and again, and so on

CBN & CBV

$$
\begin{aligned}
&(\lambda xy.y)((\lambda x.xx)(\lambda x.xx)) \\
\rightsquigarrow &(\lambda xy.y)((\lambda x.xx)(\lambda x.xx)) \\
\rightsquigarrow &(\lambda xy.y)((\lambda x.xx)(\lambda x.xx)) \\
\rightsquigarrow &\ldots
\end{aligned}
$$

CBN & CBV
$(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$
$\rightsquigarrow(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$
$\rightsquigarrow(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$
$\rightsquigarrow\ldots$

1. In call-by-value, we run the innermost, rightmost computation
2. So the infinite computation runs, reducing to itself, so the whole term reduces to itself
3. So we run it again, and again, and so on

CBN & CBV

1. The previous example highlights the benefits of call-by-name, but there is a benefit to the call-by-value strategy as well
2. But consider this scenario. Imagine M has lots of different uses of the variable x, and being applied to it is a very long copmutation
3. In call-by-name, you'd immediately apply the argument without reducing it, and then have to compute it separately in every instance of x in M, which would invlove a lot of unnecessary computations
4. In call-by-value, you'd reduce the arugment first and then apply it, meaning that the computation of the argument is only done once, compared to the many times in call-by-name's strategy

CBN & CBV

$$(\lambda x.M)((\lambda x_1, x_2, x_3, \ldots x_n.N)N_1N_2N_3\ldots N_n)$$

CBN & CBV
$(\lambda x.M)((\lambda x_1, x_2, x_3, \ldots x_n.N)N_1N_2N_3\ldots N_n)$

1. The previous example highlights the benefits of call-by-name, but there is a benefit to the call-by-value strategy as well
2. But consider this scenario. Imagine M has lots of different uses of the variable x, and being applied to it is a very long copmutation
3. In call-by-name, you'd immediately apply the argument without reducing it, and then have to compute it separately in every instance of x in M, which would invlove a lot of unnecessary computations
4. In call-by-value, you'd reduce the arugment first and then apply it, meaning that the computation of the argument is only done once, compared to the many times in call-by-name's strategy

CBN & CBV

$$(\lambda x.M)((\lambda x_1, x_2, x_3, \ldots x_n.N)N_1N_2N_3\ldots N_n)$$

CBN & CBV
$(\lambda x.M)((\lambda x_1, x_2, x_3, \ldots x_n.N)N_1N_2N_3\ldots N_n)$

1. The previous example highlights the benefits of call-by-name, but there is a benefit to the call-by-value strategy as well
2. But consider this scenario. Imagine M has lots of different uses of the variable x, and being applied to it is a very long copmutation
3. In call-by-name, you'd immediately apply the argument without reducing it, and then have to compute it separately in every instance of x in M, which would invlove a lot of unnecessary computations
4. In call-by-value, you'd reduce the arugment first and then apply it, meaning that the computation of the argument is only done once, compared to the many times in call-by-name's strategy

CBN & CBV

$$(\lambda x.M)((\lambda x_1, x_2, x_3, \ldots x_n.N)N_1N_2N_3\ldots N_n)$$

CBN & CBV
$(\lambda x.M)((\lambda x_1, x_2, x_3, \ldots x_n.N)N_1N_2N_3\ldots N_n)$

1. The previous example highlights the benefits of call-by-name, but there is a benefit to the call-by-value strategy as well
2. But consider this scenario. Imagine M has lots of different uses of the variable x, and being applied to it is a very long copmutation
3. In call-by-name, you'd immediately apply the argument without reducing it, and then have to compute it separately in every instance of x in M, which would invlove a lot of unnecessary computations
4. In call-by-value, you'd reduce the arugment first and then apply it, meaning that the computation of the argument is only done once, compared to the many times in call-by-name's strategy

Abstract machine

$$s \quad E \Longrightarrow \langle s \mid E \rangle$$

Abstract machine
$s \quad E \Longrightarrow \langle s \mid E \rangle$

1. You can simulate both of the strategies in a stack machine
2. We will change the traditional machine notation to this style called a command. Here s is run with the stack E

Abstract machine

$$\langle (\lambda x.s) \mid t \cdot E \rangle$$

Abstract machine
$\langle (\lambda x.s) \mid t \cdot E \rangle$

1. The call-by-name and call-by-value machines are different. This difference lies in what happens when you have a λ term with a term on the stack
2. In call-by-name, you'd pop the term off the stack and substitute it, then continue running the resulting term in what's left of the stack
3. But in call-by-value, you'd swap the terms around, and continue running whatever was on the stack and saving the λ for later - we use a different kind of stack operator to mean "saving for application", rather than "saving to be applied"

Abstract machine

$$\begin{array}{ll} & \langle (\lambda x.s) \mid t \cdot E \rangle \\ \rightsquigarrow_n & \langle s \left[{}^{t}/_{x} \right] \mid E \rangle \end{array}$$

Abstract machine
$\langle (\lambda x.s) \mid t \cdot E \rangle$
$\rightsquigarrow_n$ $\langle s \left[{}^{t}/_{x} \right] \mid E \rangle$

1. The call-by-name and call-by-value machines are different. This difference lies in what happens when you have a λ term with a term on the stack
2. In call-by-name, you'd pop the term off the stack and substitute it, then continue running the resulting term in what's left of the stack
3. But in call-by-value, you'd swap the terms around, and continue running whatever was on the stack and saving the λ for later - we use a different kind of stack operator to mean "saving for application", rather than "saving to be applied"

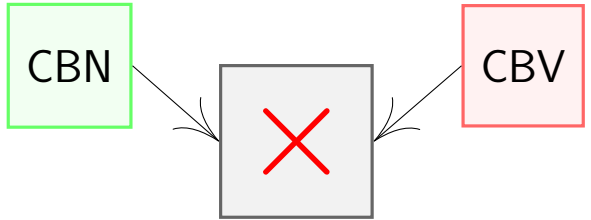
Abstract machine

$$\begin{aligned} & \langle (\lambda x.s) \mid t \cdot E \rangle \\ \rightsquigarrow_v \quad & \langle t \mid (\lambda x.s) \circ E \rangle \end{aligned}$$

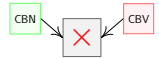
Abstract machine
$\langle (\lambda x.s) \mid t \cdot E \rangle$
$\rightsquigarrow_v$ $\langle t \mid (\lambda x.s) \circ E \rangle$

1. The call-by-name and call-by-value machines are different. This difference lies in what happens when you have a λ term with a term on the stack
2. In call-by-name, you'd pop the term off the stack and substitute it, then continue running the resulting term in what's left of the stack
3. But in call-by-value, you'd swap the terms around, and continue running whatever was on the stack and saving the λ for later - we use a different kind of stack operator to mean "saving for application", rather than "saving to be applied"

Abstract machines - incompatible

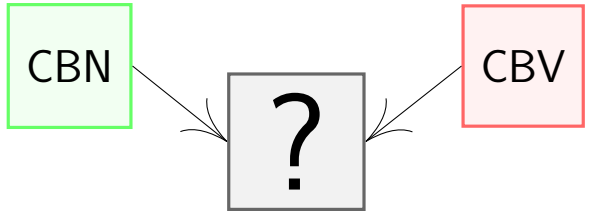


Abstract machines - incompatible



1. These machines are incompatible with each other beacuse of this difference
2. It would be interesting to have a unifying, non-deterministic machine that can run both reductions

Abstract machines - incompatible



Abstract machines - incompatible

1. These machines are incompatible with each other beacuse of this difference
2. It would be interesting to have a unifying, non-deterministic machine that can run both reductions

$\bar{\lambda}\mu\tilde{\mu}$

$$\bar{\lambda}\mu\tilde{\mu}$$

$\bar{\lambda}\mu\tilde{\mu}$
$\bar{\lambda}\mu\tilde{\mu}$

1. The $\bar{\lambda}\mu\tilde{\mu}$-calculus is exactly that - an abstract machine which runs both cbn and cbv reductions
2. For every step in reduction you take in the λ-calculus, $\bar{\lambda}\mu\tilde{\mu}$ gives you one choice - whether to reduce it via the call-by-name strategy or the call-by-value strategy. If the reduction is the same in both call-by-name and call-by-value, then it too only has one

$\bar{\lambda}\mu\tilde{\mu}$

$$c ::= \langle s \mid E \rangle$$
$$s ::= x \mid \bar{\lambda}x.s \mid \mu\alpha.c$$
$$E ::= \alpha \mid s \cdot E \mid \tilde{\mu}x.c$$

$\bar{\lambda}\mu\tilde{\mu}$
$c ::= \langle s \mid E \rangle$
$s ::= x \mid \bar{\lambda}x.s \mid \mu\alpha.c$
$E ::= \alpha \mid s \cdot E \mid \tilde{\mu}x.c$

1. Here is a high-level overview of the $\bar{\lambda}\mu\tilde{\mu}$ syntax - on top of the λ calculus, the new constructs are μ abstractions, $\tilde{\mu}$ abstractions and greek variables α
2. The idea behind introducing μ and $\tilde{\mu}$ is complicated and we don't have time in this presentation to cover them, if you want to know more you can read the report

$\bar{\lambda}\mu\tilde{\mu}$

$$\langle \bar{\lambda}x.s \mid t \cdot E \rangle \rightsquigarrow \langle t \mid \tilde{\mu}x.\langle s \mid E \rangle \rangle$$
$$\langle \mu\alpha.c \mid E \rangle \rightsquigarrow c\left[E/\alpha\right]$$
$$\langle s \mid \tilde{\mu}x.c \rangle \rightsquigarrow c\left[s/x\right]$$

$\bar{\lambda}\mu\tilde{\mu}$
$\langle \bar{\lambda}x.s \mid t \cdot E \rangle \rightsquigarrow \langle t \mid \tilde{\mu}x.\langle s \mid E \rangle \rangle$
$\langle \mu\alpha.c \mid E \rangle \rightsquigarrow c\left[E/\alpha\right]$
$\langle s \mid \tilde{\mu}x.c \rangle \rightsquigarrow c\left[s/x\right]$

1. The $\bar{\lambda}$ is a modified version of the λ abstraction, and can be thought of in a similar way
2. μ and $\tilde{\mu}$ are function abstractions that exist on either side of a command, and allow for a choice of reduction

CBN and CBV in $\bar{\lambda}\mu\tilde{\mu}$

$$(\lambda x.s)t$$

$$\langle \mu\alpha.c_t \mid \tilde{\mu}x.c_s \rangle$$

CBN and CBV in $\bar{\lambda}\mu\tilde{\mu}$
$(\lambda x.s)t$
$\langle \mu\alpha.c_t \mid \tilde{\mu}x.c_s \rangle$

1. In particular, the choice allows $\bar{\lambda}\mu\tilde{\mu}$ to implement both call-by-name and call-by-value reduction. The term $(\lambda x.s)t$ can be interpreted to something like the following form. c_s is the machine continuing to run the subterm s, and c_t is the machine continuing to run the subterm t
2. Activating the $\tilde{\mu}$ gives us CBN reduction, as it simulates substituting t into s directly
3. Activating the μ gives us CBV reduction, as it simulates putting s on the stack and reducing t

CBN and CBV in $\bar{\lambda}\mu\tilde{\mu}$

$$(\lambda x.s)t$$

$$\begin{aligned} & \langle \mu\alpha.c_t \mid \tilde{\mu}x.c_s \rangle \\ \rightsquigarrow \quad & c_s\left[{}^{\mu\alpha.c_t}/_{x}\right] \end{aligned}$$

CBN and CBV in $\bar{\lambda}\mu\tilde{\mu}$
$(\lambda x.s)t$
$\langle \mu\alpha.c_t \mid \tilde{\mu}x.c_s \rangle$
$\rightsquigarrow$ $c_s\left[{}^{\mu\alpha.c_t}/_{x}\right]$

1. In particular, the choice allows $\bar{\lambda}\mu\tilde{\mu}$ to implement both call-by-name and call-by-value reduction. The term $(\lambda x.s)t$ can be interpreted to something like the following form. c_s is the machine continuing to run the subterm s, and c_t is the machine continuing to run the subterm t
2. Activating the $\tilde{\mu}$ gives us CBN reduction, as it simulates substituting t into s directly
3. Activating the μ gives us CBV reduction, as it simulates putting s on the stack and reducing t

CBN and CBV in $\bar{\lambda}\mu\tilde{\mu}$

$$(\lambda x.s)t$$

$$\begin{aligned} & \langle \mu\alpha.c_t \mid \tilde{\mu}x.c_s \rangle \\ \rightsquigarrow \quad & c_t \left[{}^{\tilde{\mu}x.c_s}/_{\alpha} \right] \end{aligned}$$

CBN and CBV in $\bar{\lambda}\mu\tilde{\mu}$
$(\lambda x.s)t$
$\langle \mu\alpha.c_t \mid \tilde{\mu}x.c_s \rangle$
$\rightsquigarrow$ $c_t \left[{}^{\tilde{\mu}x.c_s}/_{\alpha} \right]$

1. In particular, the choice allows $\bar{\lambda}\mu\tilde{\mu}$ to implement both call-by-name and call-by-value reduction. The term $(\lambda x.s)t$ can be interpreted to something like the following form. c_s is the machine continuing to run the subterm s, and c_t is the machine continuing to run the subterm t
2. Activating the $\tilde{\mu}$ gives us CBN reduction, as it simulates substituting t into s directly
3. Activating the μ gives us CBV reduction, as it simulates putting s on the stack and reducing t

A different approach...

1. Around the same time as the creation of $\bar{\lambda}\mu\tilde{\mu}$, another calculus was being developed around modelling cbn and cbv from a different approach
2. Call-by-push-value. This calculus deals with both cbn and cbv distinctly - you use different syntax to describe which strategy you want to use at different times
3. In particular, it differentiates between computations and values in a strict way - they are completely distinct sets of terms. In fact, CBPV's main slogan is "a computation does, a value is"

CBPV

CBPV

CBPV

CBPV

1. Around the same time as the creation of $\bar{\lambda}\mu\tilde{\mu}$, another calculus was being developed around modelling cbn and cbv from a different approach
2. Call-by-push-value. This calculus deals with both cbn and cbv distinctly - you use different syntax to describe which strategy you want to use at different times
3. In particular, it differentiates between computations and values in a strict way - they are completely distinct sets of terms. In fact, CBPV's main slogan is "a computation does, a value is"

CBPV

CBPV

"A computation does, a value is"

CBPV
CBPV
"A computation does, a value is"

1. Around the same time as the creation of $\bar{\lambda}\mu\tilde{\mu}$, another calculus was being developed around modelling cbn and cbv from a different approach
2. Call-by-push-value. This calculus deals with both cbn and cbv distinctly - you use different syntax to describe which strategy you want to use at different times
3. In particular, it differentiates between computations and values in a strict way - they are completely distinct sets of terms. In fact, CBPV's main slogan is "a computation does, a value is"

CBPV

MN

CBPV
MN

1. To run an application in a call-by-name style, you'd
2. Freeze the applied term, and unfreeze it where necessary after it's been substituted. Frozen computations won't be executed, so it can be seen as a value - it is, not does

CBPV

$$M\{N\}$$

CBPV
$M\{N\}$

1. To run an application in a call-by-name style, you'd
2. Freeze the applied term, and unfreeze it where necessary after it's been substituted. Frozen computations won't be executed, so it can be seen as a value - it is, not does

CBPV

MN

CBPV
MN

1. To run an application in a call-by-value style, you'd
2. First run M until it stops, i.e. is no longer a computation but is a value. We use the syntax return to say a computation has finished running, and place whatever value it is in the variable f
3. Then run N until it returns, and place whatever value it is in the variable x
4. And then apply the two. But since f is a value, we expect it to be a computation that has been frozen and we want to re-start it, which is what the ! does

CBPV

$$MN$$

$$\texttt{let } f \leftarrow M \texttt{ in}$$

CBPV
MN
$\texttt{let } f \leftarrow M \texttt{ in}$

1. To run an application in a call-by-value style, you'd
2. First run M until it stops, i.e. is no longer a computation but is a value. We use the syntax return to say a computation has finished running, and place whatever value it is in the variable f
3. Then run N until it returns, and place whatever value it is in the variable x
4. And then apply the two. But since f is a value, we expect it to be a computation that has been frozen and we want to re-start it, which is what the ! does

CBPV

$$MN$$

$$\texttt{let } f \leftarrow M \texttt{ in}$$

$$\texttt{let } x \leftarrow N \texttt{ in}$$

CBPV
MN
$\texttt{let } f \leftarrow M \texttt{ in}$
$\texttt{let } x \leftarrow N \texttt{ in}$

1. To run an application in a call-by-value style, you'd
2. First run M until it stops, i.e. is no longer a computation but is a value. We use the syntax return to say a computation has finished running, and place whatever value it is in the variable f
3. Then run N until it returns, and place whatever value it is in the variable x
4. And then apply the two. But since f is a value, we expect it to be a computation that has been frozen and we want to re-start it, which is what the ! does

CBPV

$$MN$$

$$\texttt{let } f \leftarrow M \texttt{ in}$$

$$\texttt{let } x \leftarrow N \texttt{ in}$$

$$(f!)x$$

CBPV
MN
$\texttt{let } f \leftarrow M \texttt{ in}$
$\texttt{let } x \leftarrow N \texttt{ in}$
$(f!)x$

1. To run an application in a call-by-value style, you'd
2. First run M until it stops, i.e. is no longer a computation but is a value. We use the syntax return to say a computation has finished running, and place whatever value it is in the variable f
3. Then run N until it returns, and place whatever value it is in the variable x
4. And then apply the two. But since f is a value, we expect it to be a computation that has been frozen and we want to re-start it, which is what the ! does

Forcing and thunking

1. Both call-by-name and call-by-value use this freezing and unfreezing of computations to enforce the reduction strategy
2. This is done with syntactical constructs called "force" for unfreezing, and "thunk" for freezing

Forcing and thunking

- Force - $\cdot!$
- Thunk - $\{\cdot\}$

Forcing and thunking
- Force - $\cdot!$
- Thunk - $\{\cdot\}$

1. Both call-by-name and call-by-value use this freezing and unfreezing of computations to enforce the reduction strategy
2. This is done with syntactical constructs called "force" for unfreezing, and "thunk" for freezing

CBPV

$$\begin{aligned}\text{(Computations):}\ M, N ::= & \lambda x.M \mid MV \mid V! \\ & \mid \texttt{let}\ x \leftarrow M\ \texttt{in}\ N \\ & \mid \texttt{return}\ V \\ \text{(Values):}\ V, W ::= & x \mid \{M\}\end{aligned}$$

CBPV
(Computations): $M, N ::= \lambda x.M \mid MV \mid V!$
$\mid \texttt{let}\ x \leftarrow M\ \texttt{in}\ N$
$\mid \texttt{return}\ V$
(Values): $V, W ::= x \mid \{M\}$

1. Here is all the syntax put together
2. Here are the reduction rules imported from the λ calculus with some restrictions

CBPV

$$(\lambda x.M)V \rightsquigarrow s\,[V/x]$$

$$M \rightsquigarrow M' \implies MV \rightsquigarrow M'V$$

CBPV
$(\lambda x.M)V \rightsquigarrow s\,[V/x]$
$M \rightsquigarrow M' \implies MV \rightsquigarrow M'V$

1. Here is all the syntax put together
2. Here are the reduction rules imported from the λ calculus with some restrictions

CBPV

$$\{M\}! \rightsquigarrow M$$

$$\texttt{let}\ x \leftarrow \texttt{return}\ V\ \texttt{in}\ N \rightsquigarrow N[V/x]$$

$$M \rightsquigarrow M' \implies \texttt{let}\ x \leftarrow M\ \texttt{in}\ N \rightsquigarrow \texttt{let}\ x \leftarrow M'\ \texttt{in}\ N$$

CBPV
$\{M\}! \rightsquigarrow M$
$\texttt{let}\ x \leftarrow \texttt{return}\ V\ \texttt{in}\ N \rightsquigarrow N[V/x]$
$M \rightsquigarrow M' \implies \texttt{let}\ x \leftarrow M\ \texttt{in}\ N \rightsquigarrow \texttt{let}\ x \leftarrow M'\ \texttt{in}\ N$

1. Here are the special constructs' reduction rules, based on the intuition before. If a computation is stopped, and then started again we just conitnue as if nothing happened. If a computation returns a value, we should store that value in an assigned variables and continue runnning whatever is next. We can almost think of the let construct as a sequencing operation, so we should run M first and then N.

A machine for CBPV?

$$\text{CBPV} \to \bar{\lambda}\mu\tilde{\mu}$$

A machine for CBPV?
CBPV $\to \bar{\lambda}\mu\tilde{\mu}$

1. Since $\bar{\lambda}\mu\tilde{\mu}$ is an abstract machine for both the call-by-name and the call-by-value strategies, and CBPV is a subsumption of both call-by-name and call-by-value, and interesting question is can $\bar{\lambda}\mu\tilde{\mu}$ be an abstract machine for CBPV?
2. This project investigated this question, and the answer is...

A machine for CBPV? - NO!

$$\cancel{\text{CBPV} \rightarrow \bar{\lambda}\mu\tilde{\mu}}$$

$\cancel{\text{CBPV} \rightarrow \bar{\lambda}\mu\tilde{\mu}}$

1. No! $\bar{\lambda}\mu\tilde{\mu}$ cannot simulate CBPV
2. In particular, it's forcing and thunking that cause the problem

The problem with forcing/thunking

$$\{M\}! \rightsquigarrow M$$

The problem with forcing/thunking
$\{M\}! \rightsquigarrow M$

1. Recall the only reduction rule for forcing and thunking
2. We can simulate this in an abstract machine as so. But this doesn't interact with the stack at all! Which defeats the whole purpose of using an abstract machine

The problem with forcing/thunking

$$\begin{array}{rl} & \langle \{M\}! \mid E \rangle \\ \rightsquigarrow & \langle M \mid E \rangle \end{array}$$

The problem with forcing/thunking
$\langle \{M\}! \mid E \rangle \rightsquigarrow \langle M \mid E \rangle$

1. Recall the only reduction rule for forcing and thunking
2. We can simulate this in an abstract machine as so. But this doesn't interact with the stack at all! Which defeats the whole purpose of using an abstract machine

The problem with forcing/thunking

$$\langle V! \mid E \rangle$$
$$\rightsquigarrow \langle V \mid ! \circ E \rangle$$

$$\langle \{M\} \mid ! \circ E \rangle$$
$$\rightsquigarrow \langle M \mid E \rangle$$

The problem with forcing/thunking

$\langle V! \mid E \rangle$
$\rightsquigarrow \langle V \mid ! \circ E \rangle$

$\langle \{M\} \mid ! \circ E \rangle$
$\rightsquigarrow \langle M \mid E \rangle$

1. We could think about forcing and thunking in a different way, where we remove a force to look at it's value, and thunking takes a force off the stack and then runs normally
2. But various interpretations of these stack rules into $\bar{\lambda}\mu\tilde{\mu}$ clash with the other constructs of CBPV in $\bar{\lambda}\mu\tilde{\mu}$, which have fixed interpretations, so we can't simulate these constructs in $\bar{\lambda}\mu\tilde{\mu}$
3. An example of an interpretation which doesn't work is detailed in the report

An improved machine

$$c ::= \langle s \mid E \rangle$$
$$s ::= x \mid \bar{\lambda} x.s \mid \mu\alpha.c \mid \{s\}$$
$$E ::= \alpha \mid s \cdot E \mid \tilde{\mu} x.c \mid ! \circ E$$

An improved machine
$c ::= \langle s \mid E \rangle$
$s ::= x \mid \bar{\lambda} x.s \mid \mu\alpha.c \mid \{s\}$
$E ::= \alpha \mid s \cdot E \mid \tilde{\mu} x.c \mid ! \circ E$

1. To overcome this, we added the constructs of forcing and thunking into $\bar{\lambda}\mu\tilde{\mu}$
2. This new calculus, $\bar{\lambda}\mu\tilde{\mu}^{CBPV}$, can simulate CBPV as an abstract machine

What now?

$\bar{\lambda}\mu\tilde{\mu} \rightarrow$ new $\bar{\lambda}\mu\tilde{\mu}$

What now?
$\bar{\lambda}\mu\tilde{\mu} \rightarrow$ new $\bar{\lambda}\mu\tilde{\mu}$

1. What now? We can try to re-create the work done with the original $\bar{\lambda}\mu\tilde{\mu}$ with our new machine - for example, developing a type system and finding a Curry-Howard-style correspondence for our new machine
2. Or we can see if there's still a link between the semantics of CBPV and the semantics of $\bar{\lambda}\mu\tilde{\mu}$, even if $\bar{\lambda}\mu\tilde{\mu}$ doesn't simulate forcing and thunking. Introducing categorical semantics for $\bar{\lambda}\mu\tilde{\mu}$ would be a first step, as there is a lot of work done in that area for CBPV

What now?

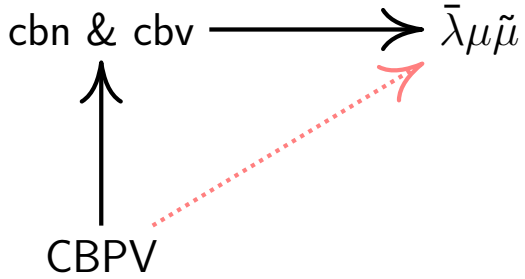
$\bar{\lambda}\mu\tilde{\mu} \rightarrow$ new $\bar{\lambda}\mu\tilde{\mu}$

CBPV $\leftrightarrow \bar{\lambda}\mu\tilde{\mu}$

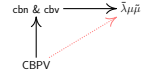
What now?
$\bar{\lambda}\mu\tilde{\mu} \rightarrow$ new $\bar{\lambda}\mu\tilde{\mu}$
CBPV $\leftrightarrow \bar{\lambda}\mu\tilde{\mu}$

1. What now? We can try to re-create the work done with the original $\bar{\lambda}\mu\tilde{\mu}$ with our new machine - for example, developing a type system and finding a Curry-Howard-style correspondence for our new machine
2. Or we can see if there's still a link between the semantics of CBPV and the semantics of $\bar{\lambda}\mu\tilde{\mu}$, even if $\bar{\lambda}\mu\tilde{\mu}$ doesn't simulate forcing and thunking. Introducing categorical semantics for $\bar{\lambda}\mu\tilde{\mu}$ would be a first step, as there is a lot of work done in that area for CBPV

Conclusion



Conclusion



1. Overall, we've combined two different calculi that are both based on cbn and cbv, and found out that although CBPV subsumes both cbn and cbv, $\bar{\lambda}\mu\tilde{\mu}$ cannot simulate it like it can with cbn and cbv