COMP523 Final Project: Call-By-Push-Value

Emily Martin (261019878) and L Denney (261042191)

April 13, 2025

Abstract

Call-By-Push-Value is a programming language paradigm that subsumes both Call-By-Value and Call-By-Name. Introduced by Paul Blain Levy in his 2001 thesis, Call-By-Push-Value is an intermediate language into which Call-By-Value and Call-By-Name programs can be translated, providing a unified framework for reasoning about both paradigms simultaneously. In this project, we explore Call-By-Push-Value static and operational semantics, and implement a typechecker and evaluator for Call-By-Push-Value programs. We also provide a transpiler from Call-By-Name programs into Call-By-Push-Value programs, as well as proving correctness of the translation on paper. Finally, we provide some interesting examples of programs written in Call-By-Push-Value, and a discussion of extensions to the language we would like to add in the future.

1 Introduction and Motivation

Modern programming languages are usually designed around one of two core evaluation strategies (or optimized modifications of these evaluation strategies): Call-By-Value (CBV) or Call-By-Name (CBN). In CBV, only fully evaluated (terminal) terms are allowed to be bound to identifiers; for instance, when a function is applied to an argument, the argument is fully evaluated before it is bound to the variable on which the function operates. CBN, in contrast, allows fully unevaluated terms to be bound to identifiers (for instance, function arguments are passed into functions unevaluated).[1]

Having two separate paradigms leads to an inherent duplication of work; each time a new form of semantics is introduced or a new result is proved, the work must be done once for each paradigm.[1] Having a single intermediate language which subsumes both CBV and CBN would allow semantics to be provided or results to be proven only once for the intermediate language, which would automatically provide semantics or prove a result for the subsumed languages.[1]

While each modern programming language can generally be categorized as either CBV or CBN, languages often include some features that defer to the other evaluation strategy. For instance, in Haskell, which is based on an Call-By-Need evaluation (an optimization of Call-By-Name where the result of a computation is cached at the first access and subsequent accesses simply retrieve rather than re-compute the value), allows strict function application using $!, which simulates CBV by forcing function arguments to be evaluated before being passed to the function.[2] Similarly, Haskell introduces bang patterns, `StrictData`, and `Strict` to support CBV behavior in pattern matching, `let` and `where` bindings, and constructor fields.[2] On the flip side, F# (a CBV language) provides built-in support for CBN-style evaluation through the `lazy` keyword, and Scala provides support for by-name parameters.[3,4]

Instead of requiring languages to pick only one evaluation strategy, if we could define an intermediate language and translations from both CBN and CBV to that intermediate language, the programmer would be able to write a program with some parts in CBV and others in CBN to then be translated into the intermediate language, which has strict rules to govern the

interactions between them. This strategy better integrates these two paradigms and allows programs to have blocks written in the appropriate or preferred paradigm for the task, giving greater freedom and control for the programmer.

Call-By-Push-Value, introduced by Paul Blain Levy in 2001,[1] provides such a unifying framework that subsumes both CBV and CBN by decomposing the terms of programs into two disjoint sets: values, which are fully already evaluated and can be passed around in a program, and computations, which evaluate to values or produce effects. Besides simply providing a subsuming language for which results can be proven once and automatically apply to both CBN and CBV languages, CBPV makes evaluation order and effect tracking more explicit and admits intermediate representations that can be exploited for compiler optimizations.

In this paper, we explore a small typed CBPV language. We implement a typechecker for CBPV programs using two mutually recursive functions (one for values and one for computations) and an evaluator for CBPV programs based on big-step operational semantics. We also provide a mechanized translation for CBN programs into CBPV programs and provide an on-paper proof that the translation from CBN to CBPV is correct. Finally, we give examples of some familiar programs written in CBPV, discuss some practical advantages of CBPV in terms of syntax-directed compiler optimizations, and outline extensions to the language we would like to explore and implement in the future.

As mentioned previously, CBPV explicitly separates **value types** and **computation types**. The types of the language are extended from Levy's 2001 dissertation[1] as follows:

Value types	$A ::= U\,\underline{B} \mid A_1 \times^v A_2 \mid A_1 + A_2 \mid \text{Unit} \mid \text{Nat} \mid \text{Bool}$
Computation types	$\underline{B} ::= F\,A \mid A \to \underline{B} \mid \underline{B_1} \times^c \underline{B_2}$

Notice that we have two product types. The product of value types is a *pattern matched product* destructed using

$$\texttt{pm t as (x,y).}t'$$

whereas the product of computation types it is a *projection product* destructed using

$$\texttt{fst, snd}$$

With pattern matched product types, we exploit the fact that both components of the product are value types and can thus be eagerly bound to identifiers and placed in the context; however, with projection products, we only extract and force to evaluate the component of the product that we need.

Here we provide the terms of our CBPV language where M is a computation and V is a value. The terms are likewise extended from those in Levy's thesis.[1]

$$\begin{aligned}
\text{Terms } V, M ::= {} & x \mid \mathit{unit} \mid V \text{`` } M \mid \lambda x : A.M \mid \texttt{fix}\, x : \underline{B}.M \\
& \mid (V, V')^v \mid \texttt{pm } V \texttt{ as } (x, y) \texttt{ in } M \\
& \mid (M, M')^c \mid \texttt{fst } M \mid \texttt{snd } M \\
& \mid \texttt{inl}^A\, V \mid \texttt{inr}^A\, V \mid (\texttt{case } V \texttt{ of } \mid \texttt{inl}^A \to M_1 \mid \texttt{inr}^{A'} \to M_2) \\
& \mid \texttt{let } x = V \texttt{ in } M \mid \texttt{bind } M \texttt{ to } x \texttt{ in } N \\
& \mid \texttt{thunk } M \mid \texttt{force } V \mid \texttt{produce } V \\
& \mid \texttt{true} \mid \texttt{false} \mid \texttt{zero} \mid \texttt{succ}\, V \mid \texttt{pred}\, V \mid \texttt{iszero}\, V
\end{aligned}$$

The lambda abstractions and applications in CBPV admit a handy stack interpretation. The command V' is understood as `push` V, where V must be a value. λx is understood as `pop` x, where the identifier x binds the value that is on the top of the stack. The applications in CBPV (V‘ M) can thus be interpreted as first pushing V to the stack and subsequently popping the top value from the stack and binding it to the first identifier abstracted over in M.

As mentioned above, only value types can be pushed to the stack or bound to an identifier in the context. However, to pass a computation around, we can "freeze" a computation into a value using `thunk`, which allows us to push it to the stack. Once we pop it, we can use `force` to unfreeze them.

The real power of the stack interpretation happens when we consider effects. In CBPV, effects are restricted to being in computations, and only values (and `thunk`ed computations) can be pushed to the stack. Anytime we push a computation to the stack, we know it will have to be `thunk`ed which freezes it, preventing it from being evaluated. This means we will never have effects occurring while they are in the stack, and are only able to occur after we pop and unfreeze them with `force`. Hence, it is easier to reason about when effects will occur at runtime in CBPV.

Additionally, the explicit thunking and forcing of computations in CBPV admits some syntax-directed compiler optimizations. For example, in CBPV, function arity and evaluation staging are exposed syntactically; in the example below, we can tell at compile time whether a function takes a single argument and returns another function or expects two arguments and returns a value.[5]

```
fun x -> fun y -> x
=> (Fun "x" (Produce (Thunk (Fun "y" (Produce (Var "x"))))

fun x y -> x
=> (Fun "x" (Fun "y" (Produce (Var "x"))))
```

In CBN/CBV languages, the latter is simply a syntactic sugar for the former and all functions are ultimately semantically curried, meaning that they are indistinguishable from each other from the compiler's perspective after de-sugaring.

2 Implementation

Our implementation consists of four main sections:

1. Typechecker for CBPV programs
2. Evaluator for CBPV programs
3. Mechanized translation from CBN to CBPV programs
4. Example programs in CBPV

2.1 Type Checking

Our language doesn't include dependent types and (when required, such as on lambda binders or sum types) our syntax of terms ask for information about types. This means our typechecker doesn't require a unification algorithm to resolve the dependencies that would occur if we didn't place these limitations on the programmer. So we are able to simply define our typechecker using two mutually recursive functions, one to check value types and one to check computation types. The mutual recursion is crucial, as typechecking a term of computation type may require

us to typecheck a term of value type nested inside and vice versa. An example of this is if we were to check the type of the program `force thunk produce true`.

Here we define the static semantics of our CBPV language following Levy's thesis,[1] modified slightly to include only binary product and sum types rather than allowing them to contain an arbitrary finite set of elements. In addition to the rules Levy defines in section 3,[1] we include booleans, natural numbers, `iszero`, `if then else`, and `fix`.

$$\frac{}{\Gamma, x : A, \Gamma' \vdash^v x : A} \qquad \frac{\Gamma \vdash^v V : A \qquad \Gamma, x : A \vdash^c M : \underline{B}}{\Gamma \vdash^c \texttt{let } x = V \texttt{ in } M : \underline{B}} \qquad \frac{\Gamma \vdash^v V : A}{\Gamma \vdash^c \texttt{produce } V : FA}$$

$$\frac{\Gamma \vdash^c M : FA \qquad \Gamma, x : A \vdash^c N : \underline{B}}{\Gamma \vdash^c \texttt{bind } M \texttt{ to } x \texttt{ in } N : \underline{B}} \qquad \frac{\Gamma \vdash^c M : \underline{B}}{\Gamma \vdash^v \texttt{thunk } M : U\underline{B}} \qquad \frac{\Gamma \vdash^v V : U\underline{B}}{\Gamma \vdash^c \texttt{force } V : \underline{B}}$$

$$\frac{\Gamma \vdash^v V : A \qquad \Gamma \vdash^v V' : A'}{\Gamma \vdash^v (V, V')^v : A \times^v A'} \qquad \frac{\Gamma \vdash^v V : A \times^v A' \qquad \Gamma, x : A, y : A' \vdash^c M : \underline{B}}{\Gamma \vdash^c \texttt{pm } V \texttt{ as } (x, y) \texttt{ in } M : \underline{B}}$$

$$\frac{\Gamma \vdash^c M : \underline{B} \qquad \Gamma \vdash^c N : \underline{B'}}{\Gamma \vdash^c (M, N)^c : \underline{B} \times^c \underline{B'}} \qquad \frac{\Gamma \vdash^c M : \underline{B} \times^c \underline{B'}}{\Gamma \vdash^c \texttt{fst } M : \underline{B}} \qquad \frac{\Gamma \vdash^v V : A}{\Gamma \vdash^v \texttt{inl}^{A'} V : A + A'}$$

$$\frac{\Gamma, x : A \vdash^c M : \underline{B}}{\Gamma \vdash^c \lambda x.M : A \to \underline{B}} \qquad \frac{\Gamma \vdash^v V_1 : A + A' \qquad \Gamma, x : A \vdash^c N_1 : \underline{B} \qquad \Gamma, y : A' \vdash^c N_2 : \underline{B}}{\Gamma \vdash^c \texttt{case } V_1 \texttt{ of } \mid \texttt{inl } x \Rightarrow N_1 \mid \texttt{inl } x \Rightarrow N_2 : \underline{B}}$$

$$\frac{\Gamma \vdash^v V : A \qquad \Gamma \vdash^c M : A \to \underline{B}}{\Gamma \vdash^c V \text{`} M : \underline{B}} \qquad \frac{\Gamma \vdash^v V_1 : \text{Bool} \qquad \Gamma \vdash^c N_1 : \underline{B} \qquad \Gamma \vdash^c N_2 : \underline{B}}{\Gamma \vdash^c \texttt{if } V_1 \texttt{ then } N_1 \texttt{ else } N_2 : \underline{B}}$$

$$\frac{\Gamma \vdash^v V : \text{Nat}}{\Gamma \vdash^v \texttt{iszero } V : \text{Bool}} \qquad \frac{\Gamma \vdash^v N : \text{Nat}}{\Gamma \vdash^v \texttt{succ } N : \text{Nat}} \qquad \frac{\Gamma, f : U\underline{B} \vdash^c M : \underline{B}}{\Gamma \vdash^c \texttt{fix } f.M : \underline{B}} \qquad \frac{}{\Gamma \vdash^v \texttt{true} : \text{Bool}}$$

Notably, when adding f to the context in the `fix` rule, we thunk its type because we enforce that f has a computation type (often an Arrow) and we allow only value types to be added to the context.

2.2 Evaluation

2.2.1 Conversion to nameless representation (De Bruijn indices), shift, and subst

To facilitate treatment of variables and avoid tracking a context of variable names, we opt to convert our named terms to a nameless representation using De Bruijn indices before evaluation.

We extend the function Debruijnify (HW1) to cover all of the additional syntax in our CBPV language. Importantly, we have four different classes of terms that introduce binders in our language: `bind M to x in` N, `let x =` V `in` M, `pm` (v, v') `as` (x, y) `in` M, and $\lambda x.M$. All of these binders behave like lambdas except for the pattern matching for pairs, which behaves like two nested `let...in` statements, e.g. `let x = v in (let y = v' in M)`, where x would be assigned index 1 and y would be assigned index 0.

Additionally, we extend the functions shift and subst to all terms in our CBPV language.

2.2.2 Operational semantics

We define our evaluator based on the following CBPV big-step operational semantics adapted from Levy's thesis.[1] This list is incomplete but the interesting cases are included, and those included cover the form of those omitted.

$$\frac{}{\texttt{produce}\ V \Downarrow \texttt{produce}\ V} \qquad \frac{M \Downarrow \texttt{produce}\ V \qquad [V/x]N \Downarrow T}{\texttt{bind}\ M\ \texttt{to}\ x\ \texttt{in}\ N \Downarrow T} \qquad \frac{[V/x]M \Downarrow T}{\texttt{let}\ x = V\ \texttt{in}\ M \Downarrow T}$$

$$\frac{M \Downarrow T}{\texttt{force thunk}\ M \Downarrow T} \qquad \frac{[V/x, V'/y]M \Downarrow T}{\texttt{pm}\ (V, V')\ \texttt{as}\ (x, y)\ \texttt{in}\ M \Downarrow T} \qquad \frac{}{\lambda x.M \Downarrow \lambda x.M}$$

$$\frac{M \Downarrow \lambda x.M \qquad [V/x]N \Downarrow T}{V\ M \Downarrow T} \qquad \frac{M \Downarrow (N, N') \qquad N \Downarrow T}{\texttt{fst}\ M \Downarrow T} \qquad \frac{[\texttt{fix}\ f.M/f]M \Downarrow T}{\texttt{fix}\ f.M \Downarrow T}$$

$$\frac{M \Downarrow \texttt{inl}\ V \qquad [V/x]N \Downarrow W}{\texttt{case}\ M\ \texttt{of}\ |\texttt{inl}\ x \Rightarrow N|\texttt{inr}\ y \Rightarrow N' \Downarrow W} \qquad \frac{M \Downarrow \texttt{succ}\ T}{\texttt{iszero}\ M \Downarrow \texttt{false}} \qquad \frac{M \Downarrow \texttt{zero}}{\texttt{iszero}\ M \Downarrow \texttt{true}}$$

3 Translation from CBN to CBPV

Translation from CBN programs to CBPV programs is crucial as it demonstrates how CBPV is capable of subsuming CBN semantics, enabling a unified operational framework/intermediate language if such translations are defined for both CBN and CBV. The primary consideration in translating CBN programs into CBPV programs is preserving lazy evaluation, particularly ensuring that computations are not prematurely forced. We achieve that with this translation by using `thunk` to encapsulate delayed computations.

3.1 Implementation of the Translation

The implementation can be found at the bottom of CBPV-typecheck.ml.

3.2 Proving the Translation

3.2.1 Defining relations

First, we define translation relation on types and contexts as in Levy's thesis.[1]
The translation on types is from types A in CBN to computation types A^n in CBPV.

$\boxed{\texttt{trans\_tp}\ \mathrm{A}\ \mathrm{A}^n}$

$\texttt{trans\_tp}\ \mathrm{Bool}\ (F\,\mathrm{Bool})$

$\texttt{trans\_tp}\ \mathrm{Unit}\ (F\,\mathrm{Unit})$

$\texttt{trans\_tp}\ (A_1 \to A_2)\ (U\,A_1^n \to A_2^n)$ when
$\texttt{trans\_tp}\ A_1\ A_1^n$
$\texttt{trans\_tp}\ A_2\ A_2^n$

$\texttt{trans\_tp}\ (A_1 + A_2)\ (F\,(U\,A_1^n + U\,A_2^n))$ when
$\texttt{trans\_tp}\ A_1\ A_1^n$
$\texttt{trans\_tp}\ A_2\ A_2^n$

$\texttt{trans\_tp}\ (A_1 \times A_2)\ (A_1^n \times^c A_2^n)$ when
$\texttt{trans\_tp}\ A_1\ A_1^n$
$\texttt{trans\_tp}\ A_2\ A_2^n$

$\boxed{\texttt{trans\_ctx}\ \Delta\ \Gamma}$

$\texttt{trans\_ctx}\ \cdot\ \cdot$

$\texttt{trans\_ctx}\ (\Delta, x : A)\ (\Gamma, x : U\ A^n)$ when $\texttt{trans\_tp}\ A\ A^n$, $\texttt{trans\_ctx}\ \Delta\ \Gamma$

Next we define our translation on terms from terms in CBN M to terms in CBPV M^n. Only the translations necessary for completing the cases covered in the proof below are included. We have too many terms in our language to include here, and many of them are immediate when keeping in mind the CBN operational semantics. Again, these are adapted from Levy's thesis.[1]

$\boxed{\texttt{trans}\ \text{M}\ \text{M}^n}$

$\texttt{trans}\ \texttt{true}\ (\texttt{produce}\ \texttt{true})$

$\texttt{trans}\ (M_1, M_2)\ (M_1^n, M_2^n)^c$ when $\texttt{trans}\ M_1\ M_1^n$, $\texttt{trans}\ M_2\ M_2^n$

$\texttt{trans}\ (M_1\text{`}\ M_2)\ ((\texttt{thunk}\ M_1^n)\text{`}\ M_2^n)$ when $\texttt{trans}\ M_1\ M_1^n$, $\texttt{trans}\ M_2\ M_2^n$

$\texttt{trans}\ (\texttt{case}\ M\ \texttt{of}\ \texttt{|inl}\ x \Rightarrow N_1 \texttt{|inr}\ y \Rightarrow N_2)$
$(\texttt{bind}\ M^n\ \texttt{to}\ z\ \texttt{in case}\ z\ \texttt{of}\ \texttt{|inl}\ x \Rightarrow N_1^n \texttt{|inr}\ y \Rightarrow N_2^n)$ when $\texttt{trans}\ M\ M^n$, $\texttt{trans}\ N_1\ N_1^n$, $\texttt{trans}\ N_2\ N_2^n$

3.2.2 Proving the translation

We want to prove it is possible to translate any program written in CBN to CBPV. This proof uses the above relations and proceeds by structural induction on the typing derivation of the CBN term. We will show a few of the cases, selected because their structures cover that of the omitted cases.

Theorem 3.1. *If* $\Delta \overset{\mathcal{D}}{\vdash} M : C$, $\texttt{trans\_ctx}\ \Delta\ \Gamma$, $\texttt{trans}\ M\ M^n$, and $\texttt{trans\_tp}\ C\ C^n$ then $\Gamma \vdash^c M^n : C^n$

Proof. by structural induction on the derivation $\mathcal{D}$, we proceed by cases

Case (true)

$$\mathcal{D} = \frac{}{\Delta \vdash \texttt{true} : \text{Bool}}$$

$\texttt{trans}\ \texttt{true}\ M^n$	by ass
$M^n := \texttt{produce}\ \texttt{true}$	by def of trans
$\texttt{trans\_ctx}\ \Delta\ \Gamma$	by ass
$\texttt{trans\_tp}\ \text{Bool}\ (F\ \text{Bool})$	by ass

Construct:

$$\frac{\dfrac{}{\Gamma \vdash^v \texttt{true} : \text{Bool}}}{\Gamma \vdash^c \texttt{produce true} : F\ \text{Bool}}$$

Case (pair)

$$\mathcal{D} = \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash M_1 : A_1 & \Delta \vdash M_2 : A_2\end{array}}{\Delta \vdash (M_1, M_2) : A_1 \times A_2}$$

$a :$	$\texttt{trans\_ctx}\ \Delta\ \Gamma$	by ass
	$\texttt{trans\_tp}\ (A_1 \times A_2)\ A^n$	by ass
	$A^n := A_1^n \times^c A_2^n$	by def of `trans_tp`
	with	
$c_1 :$	$\texttt{trans\_tp}\ A_1\ A_1^n$	
$c_2 :$	$\texttt{trans\_tp}\ A_2\ A_2^n$	
	$\texttt{trans}\ (M_1, M_2)\ M^n$	by ass
	$M^n := (M_1^n, M_2^n)^c$	by def of `trans`
	with	
$b_1 :$	$\texttt{trans}\ M_1\ M_1^n$	
$b_2 :$	$\texttt{trans}\ M_2\ M_2^n$	

Construct:

$$\frac{\begin{array}{cc}\text{IH}(\mathcal{D}_1)\ \text{w}/a, b_1, c_1 & \text{IH}(\mathcal{D}_2)\ \text{w}/a, b_2, c_2 \\ \Gamma \vdash^c M_1^n : A_1^n & \Gamma \vdash^c M_2^n : A_2^n\end{array}}{\Gamma \vdash^c (M_1^n, M_2^n)^c : A_1^n \times^c A_2^n}$$

Case (app)

$$\mathcal{D} = \frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \Delta \vdash N_1 : A & \Delta \vdash N_2 : A \to B\end{array}}{\Delta \vdash N_1\text{`}\ N_2 : B}$$

a :	$\texttt{trans\_ctx}\ \Delta\ \Gamma$	by ass
	$\texttt{trans}\ \ (N_1\text{`}\ N_2)\ M^n$	by ass
	$M^n := (\texttt{thunk}\ \ N_1^n)\text{`}\ N_2^n$	by def of `trans`
	with	
b_1 :	$\texttt{trans}\ N_1\ N_1^n$	
b_2 :	$\texttt{trans}\ N_2\ N_2^n$	
c_2 :	$\texttt{trans\_tp}\ \ (A \to B)\ D^n$	by ass
	$D^n := (U\ A^n \to B^n)$	by def of `trans_tp`
	with	
c_1 :	$\texttt{trans\_tp}\ A\ A^n$	
	$\texttt{trans\_tp}\ B\ B^n$	

Construct:

$$\dfrac{\dfrac{\begin{array}{c}\text{IH}(\mathcal{D}_1)\ \text{w}/a, b_1, c_1 \\ \Gamma \vdash^c N_1^n : A^n\end{array}}{\Gamma \vdash^v \texttt{thunk}\ \ N_1^n : U\ A^n} \qquad \begin{array}{c}\text{IH}(\mathcal{D}_2)\ \text{w}/a, b_2, c_2 \\ \Gamma \vdash^c N_2^n : U\ A^n \to B^n\end{array}}{\Gamma \vdash^c (\texttt{thunk}\ \ N_1^n)\text{`}\ N_2 : B^n}$$

Case (case)

$$\mathcal{D} = \dfrac{\begin{array}{ccc}\mathcal{D}_1 & \mathcal{D}_2 & \mathcal{D}_3 \\ \Delta \vdash M : A_1 + A_2 & \Delta, x : A_1 \vdash N_1 : B & \Delta, y : A_2 \vdash N_2 : B\end{array}}{\Delta \vdash \texttt{case}\ \ M\ \ \texttt{of}\ \ \ \texttt{|inl}\ \ x \Rightarrow N_1 \texttt{|inr}\ \ y \Rightarrow N_2 : B}$$

a :	$\texttt{trans\_ctx}\ \Delta\ \Gamma$	by ass
	$\texttt{trans}\ \ (\texttt{case}\ \ M\ \ \texttt{of|inl}\ \ x \Rightarrow N_1 \texttt{|inr}\ \ y \Rightarrow N_2)\ N'$	by ass
	$N' := (\texttt{bind}\ \ M^n\ \ \texttt{to}\ \ z\ \ \texttt{in case}\ \ z\ \ \texttt{of|inl}\ \ x \Rightarrow N_1^n \texttt{|inr}\ \ y \Rightarrow N_2^n)$	by def of `trans`
	with	
h :	$\texttt{trans}\ N_1\ N_1^n$	
i :	$\texttt{trans}\ N_2\ N_2^n$	
b :	$\texttt{trans}\ M\ M^n$	
j :	$\texttt{trans\_tp}\ \ B\ B^n$	by ass
c :	$\texttt{trans\_tp}\ \ (A_1 + A_2)\ (F(UA_1^n + UA_2^n))$	by def of `trans_tp`
	with	
d :	$\texttt{trans\_tp}\ A_1\ A_1^n$	
e :	$\texttt{trans\_tp}\ A_2\ A_2^n$	
f :	$\texttt{trans\_ctx}\ \ (\Delta, x : A_1)\ (\Gamma, x : UA_1^n)$	by def of `trans_ctx` w/a, d
g :	$\texttt{trans\_ctx}\ \ (\Delta, y : A_2)\ (\Gamma, y : UA_2^n)$	by def of `trans_ctx` w/a, e

Construct:

$$\dfrac{\begin{array}{lll} & & \mathrm{IH}(\mathcal{D}_2)\ \mathrm{w}/f,h,j \qquad\qquad \mathrm{IH}(\mathcal{D}_3)\ \mathrm{w}/g,i,j \\ \mathrm{IH}(\mathcal{D}_1)\ \mathrm{w}/a,b,c & \dfrac{}{\Gamma, z:\alpha \vdash^v z:\alpha} & \Gamma, z:\alpha, x:UA_1^n, \vdash^c N_1^n : B^n \qquad \Gamma, z:\alpha, y:UA_2^n \vdash^c N_2^n : B^n \\ \Gamma \vdash^c M^n : F(\alpha) & & \Gamma, z:\alpha \vdash^c \texttt{case}\ z\ \texttt{of|inl}\ x \Rightarrow N_1^n\ \texttt{|inr}\ y \Rightarrow N_2^n : B^n \end{array}}{\Gamma \vdash^c \texttt{bind}\ M^n\ \texttt{to}\ z\ \texttt{in case}\ z\ \texttt{of|inl}\ x \Rightarrow N_1^n\ \texttt{|inr}\ y \Rightarrow N_2^n : B^n}$$

Where $\alpha := UA_1^n + UA_2^n$ and we weaken the result of both $\mathrm{IH}(\mathcal{D}_2)$ and $\mathrm{IH}(\mathcal{D}_3)$ □

4 Example Programs

A simple yet interesting program is diverge.[1]

```
Fix ("diverge", F Unit , Force (Var "diverge"))
```

Diverge typechecks and when run, loops forever. Although we maybe have some intuition that our language is not normalizing from the introduction of an unrestrained `fix`, this is firm proof that it isn't.

Another more complicated but certainly more productive example is the factorial function. It refers to other functions defined above, namely addcomp and timescomp which perform addition and multiplication of two computations, respectively. Factorial is able to refer to the aforementioned functions because of the nesting of let-ins. A downside of this implementation of a program with many functions is that running it only outputs a single result, that of the function defined last in the given file.
The program below will compute the value of 4! when evaluated.

```
Letin ("add",
  Fix("add", Arrow (U (CCross(F Nat,F Nat)), F Nat),
  Lam("xy", U (CCross( F Nat, F Nat)),
  Bind (Fst (Force (Var "xy")), "x" ,
  Bind (Snd (Force (Var "xy")), "y",
  IfThEl (IsZero (Var "x"),
  Produce (Var "y"),
  App (
    Thunk (CompPair(Produce (Pred (Var "x")),
    Produce (Succ (Var "y")))), Force (Var "add" )))))),
Letin ("times",
  Fix ("times", Arrow (U (CCross(F Nat,F Nat)), F Nat),
  Lam("ab",U (CCross( F Nat,F Nat)),
  Bind (Fst (Force (Var "ab")), "a" ,
  Bind (Snd (Force (Var "ab")), "b",
  IfThEl (IsZero (Var "a"),
    Produce Zero,
    App (
    Thunk (CompPair (Produce (Var "b"),
        App (
            Thunk (CompPair( Produce (Pred (Var "a")), Produce (Var "b"))),
            Force (Var "times")))),
    Force (Var "add")))))),
Letin ("factorial",
  Fix ("factorial", Arrow (Nat, F Nat),
  Lam ("n", Nat,
```

```
    IfThEl( IsZero (Var "n"),
        Produce (Succ Zero),
        App ( Thunk (CompPair( Produce (Var "n"),
            App( Pred (Var "n") ,Force (Var "factorial")))),
                Force (Var "times")))),

  App (Succ (Succ (Succ( Succ Zero))), Force (Var "factorial"))
  )))
```

An important decision in the implementation of factorial was that add and times take in a computation pair rather than a value pair. This choice was motivated by the fact that we want to be able to pass unevaluated functions to them in factorial. If add and times were defined to take in value pairs, and we want to pass in a computation as an argument, we would be required to thunk it to satisfy the type system. However, this would necessitate that add or times explicitly force the argument, but we might also want their arguments to simply be Nats. This requires distinct implementations, one that can accept thunked arrow types and one that can accept Nats.

In contrast, choosing to use computation pairs as the inputs to add and times lets us more easily support both evaluated and unevaluated arguments, with the tradeoff of wrapping Nats in a `produce`. This avoids the confusion of wondering if add is able to accept a Nat or not.

Versions of add and times that accept a value pair of Nats (referred to as valadd and valtimes) are included in the supporting file `CBPV-typecheck.ml`.

5 Related Works

As mentioned previously, the power of CBPV becomes apparent when effects are introduced. For this reason, extending CBPV with algebraic effects is a popular direction to take.[1,6,7] Naturally, there has also been work done to model environments in CBPV, and to model its operational semantics in the form of a Felleisen-Friedman CK-machine in which a new judgment of stacks falls out.[8,9] Following this, there has been work in logic on Adjunction models for CBPV following the CK-machine, and also on an interpretation of CBPV in the Scott model of Linear Logic.[8,10] One last implementation worth mentioning is the formalization of effect-free CBPV in the proof assistant Coq.[11]

6 Conclusion and Future Work

In this project, we have explored a small CBPV language for which we have implemented a typechecker and evaluator as well as a translation from CBN programs into CBPV programs.

In future work, we plan to implement the translation from CBV to CBPV. This is towards the subsumption of CBN and CBV by CBPV, affirming CBPV's ability to be an intermediate language into which both CBV and CBN programs can be transpiled. Additionally, we aim to extend the language to support dependent types, which would enable a more expressive type system to guarantee richer program invariants. Finally, incorporating an effect system, particularly one based on algebraic effects and handlers, would allow CBPV to model real-world side effects and would in turn allow more expressive and useful programs to be written in CBPV or translated into CBPV. Finally, we hope to further explore the practical compiler optimizations that translating CBV/CBN programs into CBPV admits.

References

[1] Paul Blain Levy. *Call-By-Push-Value*. PhD thesis, Queen Mary and Westfield College, University of London, 2001.

[2] GHC Team. *The Glasgow Haskell Compiler User's Guide: Bang patterns and Strict Haskell*. The Glasgow Haskell Compiler, 2023. `https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/strict.html`.

[3] Microsoft. *Microsoft Learn: Lazy Expressions - F#*, 2022. `https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/lazy-expressions`.

[4] EPFL. *Scala Documentation: By-name Parameters*, 2025. `https://docs.scala-lang.org/tour/by-name-parameters.html`.

[5] Ethan Smith. *"I'm betting on Call-by-Push-Value"*. `https://thunderseethe.dev/posts/bet-on-cbpv/`.

[6] Ohad Kammar. *Algebraic theory of type-and-effect systems*. PhD thesis, 06 2014.

[7] Gordon Plotkin and Matija Pretnar. A logic for algebraic effects. In *Logic in Computer Science, 2008. LICS '08. 23rd Annual IEEE Symposium on*, pages 118–129, United States, 2008. Institute of Electrical and Electronics Engineers. Twenty-Third Annual IEEE Symposium on Logic in Computer Science ; Conference date: 24-06-2008.

[8] Paul Blain Levy. Adjunction models for call-by-push-value with stacks. *Electronic Notes in Theoretical Computer Science*, 69:248–271, 2003. CTCS'02, Category Theory and Computer Science.

[9] Paul Blain Levy, John Power, and Hayo Thielecke. Modelling environments in call-by-value programming languages. *Information and Computation*, 185(2):182–210, 2003.

[10] Thomas Ehrhard. Call-by-push-value from a linear logic point of view. In Peter Thiemann, editor, *Programming Languages and Systems*, pages 202–228, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[11] Yannick Forster, Steven Schäfer, Simon Spies, and Kathrin Stark. 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, page 118–131, New York, NY, USA, 2019. Association for Computing Machinery.