15-791 Project
Compiling by Push-Value

Zach Battleman (zbattlem)
Sonya Simkin (ssimkin)

1 Introduction

The Call-By-Push-Value (CBPV) framework, as first presented by Levy [3], lifts the distinction between values and computations to the type level. In this system, types are separated from one another according to their polarity: *positive* type (types that are characterized by their introduction) are considered to be value types, whereas *negative* types (types that are characterized by their elimination) as considered to be computation types.

In a talk given by Max New [5], it was suggested to use CBPV as a compiler IR, yielding both theoretical and practical benefits. For example, closure conversion is identified by precisely suspensions on a term level and $\mathsf{U}(-)$ on a type level, matching the intuition that computations execute in an environment, whereas values simply are. Additionally, while not in the scope of this report, there are other benefits as well, such as type-correct polymorphic CPS conversion.

In this project, we implement a compiler for a subset of ML using CBPV itself as an IR as well an additional CBPV-flavored IR for closure conversion. We justify the correctness for the translations on both a type and term level, and present a case for these IRs not only being correct, but helpful in lowering the complexity of implementing the compiler.

2 Compiler Phases

2.1 Source Language

The source language for our compiler is a limited ML-like language that contains functions, sums, positive products, and a print effect. The grammar for the language is given in Figure 1 (note that the s in the `print` construct is a string constant). A program is defined to be a singular well-typed term constructed from the grammar.

2.2 IR Translation

2.2.1 ML to CBPV

The first step of our translation is embedding our ML fragment into CBPV, taking inspiration from the translation between Lax and CBPV as described in [1]. The grammar for our CBPV IR is described in Figure 2. Note that products and sums are n-ary as opposed to the binary products and sums in our source language – this is simply for convenience in implementing closure conversion, as described in Section 2.3.

Instead of first defining an explicit translation to a Lax IR and then translating to CBPV, we make the Lax translation implicit and go straight from our ML to CBPV.

For an ML term of type A, we translate it to a computation term of type $\mathsf{F}(\|A\|)$, where the

Typ	A	$::=$	unit	unit	`unit`
			prod$(A_1; A_2)$	$A_1 \times A_2$	`A1 * A2`
			sum$(A_1; A_2)$	$A_1 + A_2$	`A1 + A2`
			arr$(A_1; A_2)$	$A_1 \to A_2$	`A1 -> A2`
Term	M	$::=$	x	x	`x`
			triv	$\langle\rangle$	`<>`
			check$(M; M')$	check M in M'	`check M in M'`
			pair$(M_1; M_2)$	$\langle M_1, M_2\rangle$	`<M1, M2>`
			split$(M; x_1.x_2.M')$	split M as x_1, x_2 in M'	`split M as x1, x2 in M'`
			inj[sum$(A_1; A_2)$](1)(M)	inj$[A_1 + A_2](1)(M)$	`inj[A1 + A2](1)(M)`
			inj[sum$(A_1; A_2)$](2)(M)	inj$[A_1 + A_2](2)(M)$	`inj[A1 + A2](2)(M)`
			case$(M; x_1.M_1; x_2.M_2)$	case M $\{x_1.M_1 \mid x_2.M_2\}$	`case M {x1 => M1 | x2 => M2}`
			$\lambda(x : A.\ M)$	$\lambda(x : A.\ M)$	`fn (x : A) => M`
			ap$(M; M')$	$M(M')$	`M M'`
			print$(s; M)$	print s; M	`print s; M`

Figure 1: Source language grammar

ValTyp	A	$::=$	tensorprod$(A_i)_{(i\in[n])}$	$A_1 \otimes ... \otimes A_n$
			sum$(A_i)_{(i\in[n])}$	$A_1 + ... + A_n$
			U(X)	U(X)
CompTyp	X	$::=$	arr$(A; X)$	$A \to X$
			F(A)	F(A)
ValTerm	V	$::=$	x	x
			prod$(V_i)_{(i\in[n])}$	$V_1 \otimes ... \otimes V_n$
			inj[sum$(A_i)_{(i\in[n])}$]$(i)(V)$	inj$[A_1 + ... + A_n](i)(V)$ $\quad (i \in [n])$
			susp(C)	susp(C)
CompTerm	C	$::=$	check$(V; C)$	check V in C
			split$(V; x_i.C)_{(i\in[n])}$	split V as $x_1, ..., x_n$ in C
			case$(V; x_i.C_i)_{(i\in[n])}$	case V $\{x_1.C_1 \mid ... \mid x_n.C_n\}$
			$\lambda(x : A.\ C)$	$\lambda(x : A.\ C)$
			ap$(C; V)$	$C(V)$
			print(s)	print s
			ret(V)	ret(V)
			bind$(C_1; x.C_2)$	bind$(C_1; x.C_2)$
			force(V)	force(V)

with the following definitions:

$$\text{unit} \triangleq \text{tensorprod}()$$
$$\star \triangleq \text{prod}()$$

Figure 2: CBPV grammar

translation $\|A\|$ is defined in the following manner:

$$\|\text{unit}\| \rightsquigarrow \text{unit}$$
$$\|A_1 \to A_2\| \rightsquigarrow \text{U}(\|A_1\| \to \text{F}(\|A_2\|))$$
$$\|A_1 \times A_2\| \rightsquigarrow \|A_1\| \otimes \|A_2\|$$
$$\|A_1 + A_2\| \rightsquigarrow \|A_1\| + \|A_2\|$$

Additionally, we define the translation $\|M\|$ on ML terms to computation terms in CBPV:

$$\begin{aligned}
\|x\| &\rightsquigarrow \mathsf{ret}(x) \\
\|\langle\rangle\| &\rightsquigarrow \mathsf{ret}(\star) \\
\|\langle M_1, M_2\rangle\| &\rightsquigarrow \mathsf{bind}(\|M_1\|; x_1.\mathsf{bind}(\|M_2\|; x_2.\mathsf{ret}(x_1 \otimes x_2))) \\
\|\mathsf{inj}[A_1 + A_2](i)(M)\| &\rightsquigarrow \mathsf{bind}(\|M\|; x.\mathsf{ret}(\mathsf{inj}[\|A_1 + A_2\|](i)(x))) \qquad (i \in \{1,2\}) \\
\|\lambda(x : A.\ M)\| &\rightsquigarrow \mathsf{ret}(\mathsf{susp}(\lambda(x : \|A\|.\ \|M\|))) \\
\|\mathsf{check}(M_1; M_2)\| &\rightsquigarrow \mathsf{bind}(\|M_1\|; x.\mathsf{check}(x; \|M_2\|)) \\
\|\mathsf{split}(M_1; x_1, x_2.\ M_2)\| &\rightsquigarrow \mathsf{bind}(\|M_1\|; v_1.\mathsf{split}(v_1; x_1, x_2.\|M_2\|)) \\
\|\mathsf{case}(M; x_1.M_1 \mid x_2.M_2)\| &\rightsquigarrow \mathsf{bind}(\|M\|; v.\mathsf{case}(v; x_1.\|M_1\| \mid x_2.\|M_2\|)) \\
\|\mathsf{ap}(M_1; M_2)\| &\rightsquigarrow \mathsf{bind}(\|M_1\|; f.\mathsf{bind}(\|M_2\|; v.\ \mathsf{ap}(\mathsf{force}(f); v))) \\
\|\mathsf{print}(s; M)\| &\rightsquigarrow \mathsf{bind}(\mathsf{print}\ s; u.\|M\|)
\end{aligned}$$

Observe how this embedding into CBPV codifies the dynamics of ML. Consider the ML term

$$\langle\langle\rangle, \mathsf{print}\ \texttt{"hello"}; \langle\rangle\rangle$$

According to our translation, this corresponds to the CBPV term:

$$\mathsf{bind}(\mathsf{ret}(\star); x_1.\mathsf{bind}(\mathsf{bind}(\mathsf{print}\ \texttt{"hello"}; u.\mathsf{ret}(\star)); x_2.\mathsf{ret}(x_1 \otimes x_2)))$$

This precisely captures the dynamics for evaluating a tuple in ML – we first evaluate the left component, then evaluate the right component, and finally place the resulting values into a pair.

2.2.2 Verifying the Translation

First, we verify that our term translation correctly translates terms of type A to computations of type $\mathsf{F}(\|A\|)$. This entails proving the following theorem (where $\|\Gamma\|$ is the interpretation of each $x : A$ in Γ as $x : \|A\|$):

Theorem 1. *If* $\Gamma \vdash_{\mathsf{ML}} M : A$*, then* $\|\Gamma\| \vdash_{\mathsf{CPBV}} \|M\| : F(\|A\|)$

Proof.

We proceed by rule induction on the derivation of $\Gamma \vdash_{\mathsf{ML}} M : A$. We will show the proof for $\Gamma \vdash_{\mathsf{ML}} \mathsf{ap}(M_1; M_2) : A$ as a representative case:

$$\textbf{Case}: \quad \frac{\Gamma \vdash_{\mathsf{ML}} M_1 : A_1 \to A_2 \quad \Gamma \vdash_{\mathsf{ML}} M_2 : A_1}{\Gamma \vdash_{\mathsf{ML}} \mathsf{ap}(M_1; M_2) : A_2}$$

WTS: $\|\Gamma\| \vdash_{\mathsf{CBPV}} \|\mathsf{ap}(M_1; M_2)\| : \mathsf{F}(\|A_2\|)$

By the definition of our translation, we have that

$$\|\mathsf{ap}(M_1; M_2)\| \triangleq \mathsf{bind}(\|M_1\|; f.\mathsf{bind}(\|M_2\|; v.\ \mathsf{ap}(\mathsf{force}(f); v)))$$

By the **Induction Hypothesis** on the two premises, we have that $\|\Gamma\| \vdash_{\mathsf{CBPV}} \|M_1\| : \mathsf{F}(\|A_1 \to A_2\|)$ and $\|\Gamma\| \vdash_{\mathsf{CBPV}} \|M_2\| : \mathsf{F}(\|A_1\|)$, respectively. By the definition of the translation, we have

$$\|A_1 \to A_2\| \triangleq \mathsf{U}(\|A_1\| \to \mathsf{F}(\|A_2\|))$$

We then have the following derivation for the typing of the translation (leaving the annotation off of $\vdash_{\mathsf{CBPV}}$ for brevity, and define $\Gamma' \triangleq \|\Gamma\|, f : \mathsf{U}(\|A_1\| \to \mathsf{F}(\|A_2\|))$):

$$\dfrac{\|\Gamma\| \vdash \|M_1\| : \mathsf{F}(\mathsf{U}(\|A_1\| \to \mathsf{F}(\|A_2\|))) \quad \mathsf{IH}_1 \qquad \dfrac{\dfrac{\|\Gamma\| \vdash \|M_2\| : \mathsf{F}(\|A_1\|)}{\Gamma' \vdash \|M_2\| : \mathsf{F}(\|A_1\|)}\ \mathsf{weak} \quad \mathsf{IH}_2 \qquad \dfrac{\dfrac{\dfrac{}{\Gamma', v : \|A_1\| \vdash f : \mathsf{U}(\|A_1\| \to \mathsf{F}(\|A_2\|))}\ \mathsf{var}}{\Gamma', v : \|A_1\| \vdash \mathsf{force}(f) : \|A_1\| \to \mathsf{F}(\|A_2\|)} \quad \dfrac{}{\Gamma', v : \|A_1\| \vdash v : \|A_1\|}\ \mathsf{var}}{\Gamma', v : \|A_1\| \vdash \mathsf{ap}(\mathsf{force}(f); v) : \mathsf{F}(\|A_2\|)}}{\Gamma' \vdash \mathsf{bind}(\|M_2\|; v.\ \mathsf{ap}(\mathsf{force}(f); v)) : \mathsf{F}(\|A_2\|)}}{\|\Gamma\| \vdash \mathsf{bind}(\|M_1\|; f.\mathsf{bind}(\|M_2\|; v.\ \mathsf{ap}(\mathsf{force}(f); v))) : \mathsf{F}(\|A_2\|)}$$

□

Next, we verify that our translation preserves any equations that hold for the ML-language. That is, we would like to show the following theorem:

Theorem 2. *If* $\Gamma \vdash_{\mathsf{ML}} M \equiv N : A$, *then* $\|\Gamma\| \vdash_{\mathsf{CBPV}} \|M\| \equiv \|N\| : \mathsf{F}(\|A\|)$.

Before we proceed with the proof, we observe that the translation of a term $\|[M_1/x]M_2\|$ is $\mathsf{bind}(\|M_1\|; x.\|M_2\|)$. This translation makes sense for two reasons:

1. If we simply pushed the translation inwards as $[\|M_1\|/x]\|M_2\|$, then the resulting term would not be well-typed, since, for $M_1 : A$, $\|M_1\| : F(\|A\|)$, but all variables x in CBPV are of a value type
2. Translating the substitution as $\mathsf{bind}(\|M_1\|; x.\|M_2\|)$ ensures that any effects embedded in M_1 are executed before the substitution into $\|M_2\|$ occurs

Proof.

We proceed by rule induction on the derivation of $\Gamma \vdash_{\mathsf{ML}} M \equiv N : A$. We will show the proof for the β rule for the function type as a representative case:

$$\textbf{Case}: \dfrac{\Gamma, x : A_1 \vdash_{\mathsf{ML}} M_2 : A_2 \quad \Gamma \vdash_{\mathsf{ML}} M_1 : A_1}{\Gamma \vdash_{\mathsf{ML}} \mathsf{ap}(\lambda(x : A_1.M_2); M_1) \equiv [M_1/x]M_2 : A_2} \to -\beta$$

WTS: $\|\Gamma\| \vdash_{\mathsf{CBPV}} \|\mathsf{ap}(\lambda(x : A_1.M_2); M_1)\| \equiv \|[M_1/x]M_2\| : \mathsf{F}(\|A_2\|)$

By definition of our translation, we have

$$\|\mathsf{ap}(\lambda(x : A_1.M_2); M_1)\| \triangleq \mathsf{bind}(\mathsf{ret}(\mathsf{susp}(\lambda(x : \|A_1\|.\ \|M_2\|))); f.\mathsf{bind}(\|M_1\|; v.\ \mathsf{ap}(\mathsf{force}(f); v)))$$

We also have that

$$\|[M_1/x]M_2\| \triangleq \mathsf{bind}(\|M_1\|; x.\|M_2\|)$$

Consider the following sequence of equivalences (with equivalence rules as described in [1]):

$$\begin{aligned}
&\mathsf{bind}(\mathsf{ret}(\mathsf{susp}(\lambda(x : \|A_1\|.\ \|M_2\|))); f.\mathsf{bind}(\|M_1\|; v.\ \mathsf{ap}(\mathsf{force}(f); v))) \\
&\equiv \mathsf{bind}(\|M_1\|; v.\ \mathsf{ap}(\mathsf{force}(\mathsf{susp}(\lambda(x : \|A_1\|.\ \|M_2\|))); v)) && (\textbf{Free}\text{-}\beta) \\
&\equiv \mathsf{bind}(\|M_1\|; v.\ \mathsf{ap}(\lambda(x : \|A_1\|.\ \|M_2\|); v)) && (\textbf{Thunk}\text{-}\beta) \\
&\equiv \mathsf{bind}(\|M_1\|; v.\ [v/x]\|M_2\|) && (\to -\beta)
\end{aligned}$$

We observe that the term $\mathsf{bind}(\|M_1\|; v.\ [v/x]\|M_2\|)$ is equivalent to $\mathsf{bind}(\|M_1\|; x.\|M_2\|)$, as in both expressions, the value resulting from the execution of $\|M_1\|$ will be substituted for x in $\|M_2\|$. Thus, by transitivity, we have that $\|\mathsf{ap}(\lambda(x : A_1.M_2); M_1)\| \equiv \|[M_1/x]M_2\|$. □

2.3 Closure Conversion

2.3.1 Motivation

In the next step of our compilation, we instrument closure conversion. While closure conversion immediately after elaboration into the first IR is somewhat unorthodox, we choose to do it this way because we felt it best highlights the benefits of using CBPV as an IR. In particular, given the translation of $\|A_1 \to A_2\| \rightsquigarrow \mathsf{U}(\|A_1\| \to \mathsf{F}(\|A_2\|))$ in the previous step, the closures occur exactly at $\mathsf{U}(-)$ allowing for a simple type directed translation.

2.3.2 The language

This new language, which we are calling Closure Convert By Push Value (CCBPV), is much the same as the original CBPV IR, but is slightly augmented to facilitate closure conversion.

We follow in the footsteps of [4, 5] and use polymorphic existential types to represent closures. Naturally, this raises the question of how to represent existentials. We take the view of existentials being positive and extend our value types with $\exists t.B$. Much like how term variables range only over values, we maintain that type variables only range over value types. Along with this, we extend our value terms with $\mathsf{pack}(A; V)$, and dually, our computation terms with $\mathsf{unpack}(V; x.C)$. The statics and dynamics for such terms are exactly the same as in [2], albeit with this new CBPV flavor and a renaming of the elimination rule for existential types. Finally, we introduce a new value type construct $\mathbb{U}(X)$ as well is its intro and elim forms, close and open. This represents a suspended, *closed* computation of type X. Importantly, this replaces the U in the original CBPV, as well as the force and susp constructions. This new construction has the following intro and elim forms:

$$\frac{\cdot \vdash C : X}{\Gamma \vdash \mathsf{close}(C) : \mathbb{U}(X)} \qquad \frac{\Gamma \vdash M : \mathbb{U}(X)}{\Gamma \vdash \mathsf{open}(M) : X}$$

For convenience, the formal grammar is displayed in Figure 3

ValTyp	A	::=	...
			t
			$\exists t.B$
			$\mathbb{U}(X)$
CompTyp	X	::=	...
ValTerm	V	::=	...
			$\mathsf{pack}(A; V)$
			$\mathsf{close}(C)$
CompTerm	C	::=	...
			$\mathsf{unpack}(V; x.C)$
			$\mathsf{open}(V)$

Figure 3: CCBPV grammar

$$\frac{\Gamma \vdash C : X \rightsquigarrow C'}{\Gamma \vdash \mathsf{susp}(C) : UX \rightsquigarrow \mathsf{pack}(\otimes\Gamma; \langle\langle x_1, \ldots, x_n\rangle, \mathsf{close}(\lambda g : \otimes\Gamma.\mathsf{split}(g; x_1 \ldots x_n.C'))\rangle)}$$

$$\frac{\Gamma \vdash M : \mathsf{U}X \rightsquigarrow M'}{\Gamma \vdash \mathsf{force}(M) : X \rightsquigarrow \mathsf{unpack}(M'; x.\mathsf{split}(x; y_{\mathsf{env}}.c.(\mathsf{open}(c))(y_{\mathsf{env}})))}$$

with definitions

$$\begin{aligned}\Gamma &:= \{x_1 : A_1, \ldots, x_n : A_n\}\\ \otimes\Gamma &:= A_1 \otimes \cdots \otimes A_n\end{aligned}$$

Figure 4: Closure Conversion Translations

2.3.3 Translation

We have set up this language to facilitate type-directed closure conversion and do so as follows. The type translation occurs in the following manner using $\|-\|$ for value types and $|-|$ for computation types.

$$\begin{aligned}\|A_1 \otimes \cdots \otimes A_n\| &\rightsquigarrow \|A_1\| \otimes \cdots \otimes \|A_n\|\\ \|A_1 + \cdots + A_n\| &\rightsquigarrow \|A_1\| + \cdots + \|A_n\|\\ \|\mathsf{U}(X)\| &\rightsquigarrow \exists t.(t \otimes \mathbb{U}(t \to |X|))\end{aligned}$$

$$\begin{aligned}|A \to X| &\rightsquigarrow \|A\| \to |X|\\ |\mathsf{F}(A)| &\rightsquigarrow \mathsf{F}(\|A\|)\end{aligned}$$

This translation is mostly straightforward, with the notable feature of capturing closures in polymorphic existential types. Perhaps most interestingly, in [4] they translate

$$|t_1 \to t_2| \rightsquigarrow \exists t.(t \times \mathsf{code}(t; t_1; t_2))$$

where $\mathsf{code}(\tau; \tau_1; \tau_2)$ is a closure which takes an environment of type τ, an argument of type τ_1 and returns a τ_2. Of course, this is translating from ML to what they call $\lambda^{\exists}$ which is a CBV language. It turns out that our translation pipeline, even with the redirection into CBPV, yields the same result, with

$$A \to B \rightsquigarrow \mathsf{U}(A \to \mathsf{F}B) \rightsquigarrow \exists t.(t \otimes \mathbb{U}(t \to \mathsf{F}(\|B\|))$$

This bears a striking similarity to the 1996 original, albeit with some syntactic changes.

In a similar manner, we introduce a term level translation. Unlike the translation from ML to CBPV, we must maintain a context for the purpose of capturing the environment. The translation is almost entirely identity, with the notable cases occurring at suspensions and forces. The rules are shown in Figure 4. The intuition for these rules is fairly straight forward: suspensions pull out all of the variables in C' into a tupled existential package, and when forced, the computation is reopened, and the variables are substituted back in.

2.3.4 Type Correctness

We motivate the type correctness with the following theorem:

Theorem 3. *1.* $\Gamma \vdash_{\mathsf{CPBV}} V : A$ *then* $\cdot;\ \|\Gamma\| \vdash_{\mathsf{CCBPV}} \|V\| : \|A\|$

2. $\Gamma \vdash_{\mathsf{CPBV}} C : X$ *then* $\cdot;\ \|\Gamma\| \vdash_{\mathsf{CCBPV}} |C| : |X|$

Proof. Since many cases of the translation are identity, we only show a few representative cases. We proceed by simultaneous rule induction on the derivation of $\Gamma \vdash_{\mathsf{CBPV}} V : A$ and $\Gamma \vdash_{\mathsf{CBPV}}: C : X$.

- We have the following rule:

$$\frac{\Gamma \vdash_{\mathsf{CBPV}} C : X}{\Gamma \vdash_{\mathsf{CBPV}} \mathsf{susp}(C) : \mathsf{U}X}$$

We would like to show that $\|\Gamma\| \vdash_{\mathsf{CCBPV}} \|\mathsf{susp}(C)\| : \|\mathsf{U}X\|$. To show this, it suffices to show that

$$\|\Gamma\| \vdash_{\mathsf{CCBPV}} \mathsf{pack}(\otimes\|\Gamma\|; \langle\langle x_1, \ldots, x_n\rangle, \mathsf{close}(\lambda g : \otimes\|\Gamma\|.\mathsf{split}(g; x_1 \ldots x_n.|C|))\rangle) : \exists t.(t \otimes \mathbb{U}(t \to |X|))$$

While a long term, the proof is fortunately straightforward. We omit the type variable context as well as the requirements of type wellformedness as those are immediate and we are tight for space:

$$\dfrac{\dfrac{x_i : \|A_i\| \in \|\Gamma\|}{\|\Gamma\| \vdash_{\mathsf{CCBPV}} x_i : \|A_i\|}}{\|\Gamma\| \vdash_{\mathsf{CCBPV}} \langle x_1, \ldots, x_n\rangle : \otimes\|\Gamma\|} \quad \dfrac{\dfrac{g : \otimes\|\Gamma\| \vdash_{\mathsf{CCBPV}} g : \otimes\|\Gamma\| \quad \dfrac{x_1 : \|A_1\|, \ldots, x_n : \|A_n\| \vdash_{\mathsf{CCBPV}} |C| : |X|}{g : \otimes\|\Gamma\|, x_1 : \|A_1\|, \ldots, x_n : \|A_n\| \vdash_{\mathsf{CCBPV}} |C| : |X|}\ \mathsf{IH}}{g : \otimes\|\Gamma\| \vdash_{\mathsf{CCBPV}} \mathsf{split}(g; x_1 \ldots x_n.|C|) : |X|}}{\cdot \vdash_{\mathsf{CCBPV}} \lambda g : \otimes\|\Gamma\|.\mathsf{split}(g; x_1 \ldots x_n.|C|)) : \otimes\|\Gamma\| \to |X|}}{\|\Gamma\| \vdash_{\mathsf{CCBPV}} \mathsf{close}(\lambda g : \otimes\|\Gamma\|.\mathsf{split}(g; x_1 \ldots x_n.|C|)) : \mathbb{U}(\otimes\|\Gamma\| \to |X|)}$$

$$\frac{\|\Gamma\| \vdash_{\mathsf{CCBPV}} \langle x_1, \ldots, x_n\rangle : \otimes\|\Gamma\| \quad \|\Gamma\| \vdash_{\mathsf{CCBPV}} \mathsf{close}(\lambda g : \otimes\|\Gamma\|.\mathsf{split}(g; x_1 \ldots x_n.|C|)) : \mathbb{U}(\otimes\|\Gamma\| \to |X|)}{\|\Gamma\| \vdash_{\mathsf{CCBPV}} \langle\langle x_1, \ldots, x_n\rangle, \mathsf{close}(\lambda g : \otimes\|\Gamma\|.\mathsf{split}(g; x_1 \ldots x_n.\gamma(|C|)))\rangle : \otimes\Gamma \otimes \mathbb{U}(\otimes\|\Gamma\| \to |X|)}$$

$$\frac{\|\Gamma\| \vdash_{\mathsf{CCBPV}} \mathsf{pack}(\otimes\|\Gamma\|; \langle\langle x_1, \ldots, x_n\rangle, \mathsf{close}(\lambda g : \otimes\|\Gamma\|.\mathsf{split}(g; x_1 \ldots x_n.|C|))\rangle) : \exists t.(t \otimes \mathbb{U}(t \to |X|))}{\|\Gamma\| \vdash_{\mathsf{CCBPV}} \langle\langle x_1, \ldots, x_n\rangle, \mathsf{close}(\lambda g : \otimes\|\Gamma\|.\mathsf{split}(g; x_1 \ldots x_n.|C|))\rangle : \otimes\Gamma \otimes \mathbb{U}(\otimes\|\Gamma\| \to |X|)}$$

- We have the following rule:

$$\frac{\Gamma \vdash_{\mathsf{CBPV}} M : \mathsf{U}X}{\Gamma \vdash_{\mathsf{CBPV}} \mathsf{force}(M) : X}$$

We would like to show that

$$\cdot; \Gamma \vdash_{\mathsf{CCBPV}} \mathsf{unpack}(|M|; x.\mathsf{split}(x; y_{\mathsf{env}}.c.(\mathsf{open}(c))(y_{\mathsf{env}}))) : |X|$$

Again, the proof is lengthy, but ultimately straightforward:

$$\dfrac{\dfrac{\dfrac{t : \mathsf{type}; \|\Gamma\|, c : \mathbb{U}(t \to |X|) \vdash_{\mathsf{CCBPV}} c : \mathbb{U}(t \to |X|)}{t : \mathsf{type}; \|\Gamma\|, c : \mathbb{U}(t \to |X|) \vdash_{\mathsf{CCBPV}} \mathsf{open}(c) : t \to |X|} \quad t : \mathsf{type}; y_{\mathsf{env}} : t \vdash_{\mathsf{CCBPV}} y_{\mathsf{env}} : t}{t : \mathsf{type}; \|\Gamma\|, y_{\mathsf{env}} : t, c : \mathbb{U}(t \to |X|) \vdash_{\mathsf{CCBPV}} (\mathsf{open}(c))(y_{\mathsf{env}}) : |X|}}{t : \mathsf{type}; \|\Gamma\|, x : t \otimes \mathbb{U}(t \to |X|) \vdash_{\mathsf{CCBPV}} \mathsf{split}(x; y_{\mathsf{env}}.c.(\mathsf{open}(c))(y_{\mathsf{env}})) : |X|} \quad \dfrac{\cdot; \|\Gamma\| \vdash_{\mathsf{CCBPV}} |V| : \exists t.(t \otimes \mathbb{U}(t \to |X|))}{\cdot; \|\Gamma\| \vdash_{\mathsf{CCBPV}} |V| : \exists t.(t \otimes \mathbb{U}(t \to |X|))}\ \mathsf{IH}$$

$$\cdot; \|\Gamma\| \vdash_{\mathsf{CCBPV}} \mathsf{unpack}(|V|; x.\mathsf{split}(x; y_{\mathsf{env}}.c.(\mathsf{open}(c))(y_{\mathsf{env}}))) : |X|$$

We use implicit weakening in some places and prune some branches of the tree for which the proof is immediate in the interest of the result fitting on the page.

□

2.3.5 Equational Correctness

Additionally, with this translation we conjecture an equational correctness theorem of the following form.

Theorem 4. *(Conjectured)*

1. *If* $\Gamma \vdash_{\mathsf{CBPV}} V \equiv U : A$ *then* $\cdot; \Gamma \vdash_{\mathsf{CCBPV}} \|V\| \equiv \|U\| : \|A\|$
2. *If* $\Gamma \vdash_{\mathsf{CBPV}} C \equiv D : X$ *then* $\cdot; \Gamma \vdash_{\mathsf{CCBPV}} |C| \equiv |D| : |X|$

For this theorem, we would need additional equations for the new forms introduced in CCBPV. In particular, we would add the following inspired by rules from [2, 6], with an emphasis on [6] whose β and η laws for existentials we are borrowing:

$$\frac{\cdot; \cdot \vdash_{\mathsf{CCBPV}} C : X}{\Delta; \Gamma \vdash_{\mathsf{CCBPV}} \mathsf{open}(\mathsf{close}(C)) \equiv C : X} \qquad \frac{\Delta; \Gamma \vdash_{\mathsf{CCBPV}} V : \mathbb{U}(X)}{\Delta; \Gamma \vdash_{\mathsf{CCBPV}} \mathsf{close}(\mathsf{open}(V)) \equiv V : \mathbb{U}(X)}$$

$$\frac{\Delta; \Gamma \vdash_{\mathsf{CCBPV}} V : [B/t]A \quad \Delta; \Gamma; x : [B/t]A \vdash_{\mathsf{CCBPV}} f(x) : X}{\Delta; \Gamma \vdash_{\mathsf{CCBPV}} \mathsf{unpack}(\mathsf{pack}(A; V); x.f(x)) \equiv f(V) : X}$$

$$\frac{\Delta; \Gamma, V : \exists t.A \vdash_{\mathsf{CCBPV}} C(V), D(V) : X \quad \Delta; \Gamma, y : [B/t]A \vdash_{\mathsf{CCBPV}} C(\mathsf{pack}(A; y)) \equiv D(\mathsf{pack}(A; y)) : X}{\Delta; \Gamma, V : \exists t.A \vdash_{\mathsf{CCBPV}} C(V) \equiv D(V) : X}$$

We leave this as a conjecture due to time constraints, but ultimately suspect it to be straightforward induction.

3 Implementation

The phases of the compiler as described in this report have been implemented in just under 1400 lines of Ocaml 5.1.1 which can be found on Github. The implementation is entirely faithful to this report – there is nothing in the compilation passes which is not explicitly described here. In other words, the type based compilation we describe is not only theoretically sound, but works in practice with no modifications.

In addition to the compiler phases described in this report, we also have implemented a parser for the source language, interpreters for both the CBPV and CCBPV languages, and type checkers for all languages which allows the compiler to be used and tested. Although we have not done rigorous testing, for all of the test programs we have tried, all of the intermediate representations type check and the interpreters agree on the output and evaluate as expected. Admittedly, there is not much else to say about the implementation, however, we believe this speaks to the credibility of the IRs and type-based translations in reducing the challenges of compilation.

While a type based compilation procedure such as the one described in this report makes implementation straight forward, it is possible there are bugs in the implementation we did not catch, and we make no claims about the compilers current implementation correctness. At the time of writing, the only known bug is that type variables can sometimes by considered equal when they are in fact not due to our use of named variables rather than DeBruijn indices. However, we are fairly confident that this bug cannot occur in code generated by our translation passes and thus is a non-issue for the correctness of the overall compiler.

4 Future Work

This project has many natural extensions, the most obvious of which are further compiler passes. Most pressing and immediately doable is hoisting the closed lambdas to the top level. While we did not implement it in this report, doing so would be fairly straightforward, simply lifting terms of the form $\mathsf{close}(-)$ to the top level. Additionally, one would want to prove the conjectured correctness theorem for closure conversion. As mentioned before, we suspect this would not be difficult. Finally, many of the equations we used, while we believe they are sound, are not themselves justified, and would enjoy a treatment via logical relations a la [4].

References

[1] Robert Harper. *Call-by-Push-Value and the Enriched Effects Calculus*. 2024. URL: https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/cbpv.pdf.

[2] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[3] Paul Blain Levy. *Call-By-Push-Value*. 2001. URL: https://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf.

[4] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. *Typed Closure Conversion*. 1996. URL: https://www.cs.cmu.edu/~rwh/papers/closures/tr.pdf.

[5] Max S. New. *Compiling with Call-by-push-value*. 2023. URL: https://maxsnew.com/docs/mfps2023-slides.pdf.

[6] Jonathan Sterling. *Reflections on existential types*. 2022. URL: https://www.jonmsterling.com/papers/sterling-2022-existentials.pdf.

[7] Zachary J. Sullivan. *Reflections of Closures*. 2023. URL: https://scholarsbank.uoregon.edu/xmlui/bitstream/handle/1794/29286/Sullivan_oregon_0171A_13753.pdf?sequence=1.