# Call-By-Push-Value

Hype for Types

October 28, 2024

# What We'll Talk About

- The effect of adding effects to a language
- The call-by-push-value (CBPV) paradigm
- What it means for a type to be "positive" or "negative"
- How CBPV makes a type-level distinction between values and effectful computations
- How CBPV can be used as an intermediate representation (IR) in a compiler

# Effects

# It's Super Effective!

So far, we haven't introduced any "effectful" computations into the languages we've explored, so let's do so now:

$$\frac{\Gamma \vdash e : \tau \quad (s \in \Sigma^*)}{\Gamma \vdash \textbf{print } s; e : \tau}$$

# It's Super Effective!

So far, we haven't introduced any "effectful" computations into the languages we've explored, so let's do so now:

$$\frac{\Gamma \vdash e : \tau \quad (s \in \Sigma^*)}{\Gamma \vdash \textbf{print } s; e : \tau}$$

In STLC, we would have no way to tell whether an expression is effectful or not just by looking at its type:

$$\textbf{e1} = \textbf{Left true}$$

$$\textbf{e2} = \textbf{Left } (\textbf{print } \texttt{"meow"}; \textbf{true})$$

$$\textbf{e3} = \textbf{print } \texttt{"meow"}; \textbf{Left true}$$

# It's Super Effective!

So far, we haven't introduced any "effectful" computations into the languages we've explored, so let's do so now:

$$\frac{\Gamma \vdash e : \tau \quad (s \in \Sigma^*)}{\Gamma \vdash \textbf{print } s; e : \tau}$$

In STLC, we would have no way to tell whether an expression is effectful or not just by looking at its type:

$$\textbf{e1} = \textbf{Left true}$$

$$\textbf{e2} = \textbf{Left } (\textbf{print } \texttt{"meow"}; \textbf{true})$$

$$\textbf{e3} = \textbf{print } \texttt{"meow"}; \textbf{Left true}$$

All of these expressions have type $\textbf{bool} + \tau$, but some are effectful and some are not.

# It's Super Effective!

So far, we haven't introduced any "effectful" computations into the languages we've explored, so let's do so now:

$$\frac{\Gamma \vdash e : \tau \quad (s \in \Sigma^*)}{\Gamma \vdash \textbf{print } s; e : \tau}$$

In STLC, we would have no way to tell whether an expression is effectful or not just by looking at its type:

$$\textbf{e1} = \textbf{Left true}$$

$$\textbf{e2} = \textbf{Left } (\textbf{print } \texttt{"meow"}; \textbf{true})$$

$$\textbf{e3} = \textbf{print } \texttt{"meow"}; \textbf{Left true}$$

All of these expressions have type $\textbf{bool} + \tau$, but some are effectful and some are not. This may not be a big deal with benign effects, but what if an expression's effect could change the course of evaluation?

# It's Super Effective!

So far, we haven't introduced any "effectful" computations into the languages we've explored, so let's do so now:

$$\frac{\Gamma \vdash e : \tau \quad (s \in \Sigma^*)}{\Gamma \vdash \textbf{print } s; e : \tau}$$

In STLC, we would have no way to tell whether an expression is effectful or not just by looking at its type:

$$\textbf{e1} = \textbf{Left true}$$

$$\textbf{e2} = \textbf{Left } (\textbf{print } \texttt{"meow"}; \textbf{true})$$

$$\textbf{e3} = \textbf{print } \texttt{"meow"}; \textbf{Left true}$$

All of these expressions have type $\textbf{bool} + \tau$, but some are effectful and some are not. This may not be a big deal with benign effects, but what if an expression's effect could change the course of evaluation?

What if we could bring the distinction between values and effectful computations to the type level?

# Call-By-Push-Value

# Defining CBPV

In call-by-push-value (CBPV), we divide terms into *values* and *computations* based on the polarity of their type.

# Defining CBPV

In call-by-push-value (CBPV), we divide terms into *values* and *computations* based on the polarity of their type. We have two distinct categories of types:

$$\begin{array}{lccl} \text{Positive} & A & ::= & A_1 \otimes A_2 \mid A_1 + A_2 \mid \mathbf{U}(X) \\ \text{Negative} & X & ::= & X_1 \times X_2 \mid A \to X \mid \mathbf{F}(A) \end{array}$$

# Defining CBPV

In call-by-push-value (CBPV), we divide terms into *values* and *computations* based on the polarity of their type. We have two distinct categories of types:

$$\begin{array}{llcl} \text{Positive} & A & ::= & A_1 \otimes A_2 \mid A_1 + A_2 \mid \mathbf{U}(X) \\ \text{Negative} & X & ::= & X_1 \times X_2 \mid A \to X \mid \mathbf{F}(A) \end{array}$$

and two distinct categories of terms:

$$\begin{array}{llcl} \text{Values} & V & ::= & x \mid V_1 \otimes V_2 \mid \textbf{Left } V \mid \textbf{Right } V \mid \textbf{susp}(C) \\ \text{Computations} & C & ::= & \langle C_1, C_2 \rangle \mid \textbf{fst}(C) \mid \textbf{snd}(C) \mid \lambda x : A.\ C \mid \\ & & & \textbf{ap}(C; V) \mid \textbf{split } V \textbf{ of } x_1, x_2 \Rightarrow C \mid \\ & & & \textbf{case } V \textbf{ of } \{x_1 \Rightarrow C_1 \mid x_2 \Rightarrow C_2\} \mid \textbf{force}(V) \mid \\ & & & \textbf{ret}(V) \mid \textbf{bind } x = C_1 \textbf{ in } C_2 \mid \textbf{print } s; C \end{array}$$

# Defining CBPV

In call-by-push-value (CBPV), we divide terms into *values* and *computations* based on the polarity of their type. We have two distinct categories of types:

$$\begin{array}{lcl} \text{Positive} \quad A & ::= & A_1 \otimes A_2 \mid A_1 + A_2 \mid \mathbf{U}(X) \\ \text{Negative} \quad X & ::= & X_1 \times X_2 \mid A \to X \mid \mathbf{F}(A) \end{array}$$

and two distinct categories of terms:

$$\begin{array}{lcl} \text{Values} \quad V & ::= & x \mid V_1 \otimes V_2 \mid \textbf{Left } V \mid \textbf{Right } V \mid \textbf{susp}(C) \\ \text{Computations} \quad C & ::= & \langle C_1, C_2 \rangle \mid \textbf{fst}(C) \mid \textbf{snd}(C) \mid \lambda x : A.\ C \mid \\ & & \textbf{ap}(C; V) \mid \textbf{split } V \textbf{ of } x_1, x_2 \Rightarrow C \mid \\ & & \textbf{case } V \textbf{ of } \{x_1 \Rightarrow C_1 \mid x_2 \Rightarrow C_2\} \mid \textbf{force}(V) \mid \\ & & \textbf{ret}(V) \mid \textbf{bind } x = C_1 \textbf{ in } C_2 \mid \textbf{print } s; C \end{array}$$

As a result, we have two different type-checking judgment forms:

$$\Gamma \vdash V : A \qquad\qquad \Gamma \vdash C : X$$

# Defining CBPV

The governing slogan of the CBPV paradigm is:

# Defining CBPV

The governing slogan of the CBPV paradigm is:

**Slogan**

Values *are*, computations *do*

# Defining CBPV

The governing slogan of the CBPV paradigm is:

**Slogan**

Values *are*, computations *do*

But how did we get to this definition? What does it mean to be "positive" or "negative"? What are all of these new constructs?

# Defining CBPV

The governing slogan of the CBPV paradigm is:

**Slogan**

Values *are*, computations *do*

But how did we get to this definition? What does it mean to be "positive" or "negative"? What are all of these new constructs? Let's start with polarity...

# Polarity

# Why So Positive?

We can categorize a type as either *positive* or *negative*

## Definition

A **positive** type is one whose elements are defined by their introduction (i.e. how they are created)

# Why So Positive?

We can categorize a type as either *positive* or *negative*

**Definition**

A **positive** type is one whose elements are defined by their introduction (i.e. how they are created)

For example,

- The type $\tau_1 + \tau_2$ is positive because it is defined by the values we inject into it (**Left** $e_1$ and **Right** $e_2$)

# Why So Positive?

We can categorize a type as either *positive* or *negative*

## Definition

A **positive** type is one whose elements are defined by their introduction (i.e. how they are created)

For example,

- The type $\tau_1 + \tau_2$ is positive because it is defined by the values we inject into it (**Left** $e_1$ and **Right** $e_2$)

## Idea

For values of positive types, we derive meaning from the "structure" of the introductory forms, and the eliminations treat the value as a "black box"

# Why So Negative?

We can categorize a type as either *positive* or *negative*

## Definition

A **negative** type is one whose elements are defined by their elimination (i.e. how they are used)

# Why So Negative?

We can categorize a type as either *positive* or *negative*

**Definition**

A **negative** type is one whose elements are defined by their elimination (i.e. how they are used)

For example,

- The type $\tau_1 \times \tau_2$ is negative because it is defined by the fact that we can project out of it ($\mathbf{fst}(e)$ and $\mathbf{snd}(e)$)

# Why So Negative?

We can categorize a type as either *positive* or *negative*

**Definition**

A **negative** type is one whose elements are defined by their elimination (i.e. how they are used)

For example,

- The type $\tau_1 \times \tau_2$ is negative because it is defined by the fact that we can project out of it ($\textbf{fst}(e)$ and $\textbf{snd}(e)$)
- The type $\tau_1 \to \tau_2$ is negative because it is defined by the fact that we can apply it to other expressions

# Why So Negative?

We can categorize a type as either *positive* or *negative*

## Definition

A **negative** type is one whose elements are defined by their elimination (i.e. how they are used)

For example,

- The type $\tau_1 \times \tau_2$ is negative because it is defined by the fact that we can project out of it ($\textbf{fst}(e)$ and $\textbf{snd}(e)$)
- The type $\tau_1 \to \tau_2$ is negative because it is defined by the fact that we can apply it to other expressions

## Idea

For values of negative types, we can treat the value itself as a "black box" and derive meaning about the value through its elimination

# Polarizing Products

The distinction between positive and negative types gives rise to two different definitions of the product type:

# Polarizing Products

The distinction between positive and negative types gives rise to two different definitions of the product type:

- Negative product:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \textbf{fst}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \textbf{snd}(e) : \tau_2}$$

# Polarizing Products

The distinction between positive and negative types gives rise to two different definitions of the product type:

- Negative product:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \textbf{fst}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \textbf{snd}(e) : \tau_2}$$

- Positive product:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \otimes e_2 : \tau_1 \otimes \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \otimes \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e' : \tau}{\Gamma \vdash \textbf{split}\ e\ \textbf{of}\ x_1, x_2 \Rightarrow e' : \tau}$$

# What's The Connection?

The categorization of types based on their polarity lends itself to a distinction between values and computations:

# What's The Connection?

The categorization of types based on their polarity lends itself to a distinction between values and computations:

- We take positive types to be the types of values, since their characterization is *independent* of what computations you do with them

# What's The Connection?

The categorization of types based on their polarity lends itself to a distinction between values and computations:

- We take positive types to be the types of values, since their characterization is *independent* of what computations you do with them
  - The introduction forms create values, and the eliminations are computations on values

# What's The Connection?

The categorization of types based on their polarity lends itself to a distinction between values and computations:

- We take positive types to be the types of values, since their characterization is *independent* of what computations you do with them
  - The introduction forms create values, and the eliminations are computations on values
- We take negative types to be the types of computations, since their characterization is *dependent* on what computations you do with them

# What's The Connection?

The categorization of types based on their polarity lends itself to a distinction between values and computations:

- We take positive types to be the types of values, since their characterization is *independent* of what computations you do with them
  - The introduction forms create values, and the eliminations are computations on values
- We take negative types to be the types of computations, since their characterization is *dependent* on what computations you do with them
  - The introduction forms create computations, and the eliminations produce further computations

# Getting Closer...

We now have an explanation for how we split up our STLC types into our two categories:

$$\begin{array}{llcl} \text{Positive} & A & ::= & A_1 \otimes A_2 \mid A_1 + A_2 \mid \mathbf{U}(X) \\ \text{Negative} & X & ::= & X_1 \times X_2 \mid A \to X \mid \mathbf{F}(A) \end{array}$$

$$\begin{array}{llcl} \text{Values} & V & ::= & x \mid V_1 \otimes V_2 \mid \textbf{Left } V \mid \textbf{Right } V \mid \textbf{susp}(C) \\ \text{Computations} & C & ::= & \langle C_1, C_2 \rangle \mid \textbf{fst}(C) \mid \textbf{snd}(C) \mid \lambda x : A.\ C \mid \\ & & & \textbf{ap}(C; V) \mid \textbf{split } V \textbf{ of } x_1, x_2 \Rightarrow C \mid \\ & & & \textbf{case } V \textbf{ of } \{x_1 \Rightarrow C_1 \mid x_2 \Rightarrow C_2\} \mid \textbf{force}(V) \mid \\ & & & \textbf{ret}(V) \mid \textbf{bind } \mathrm{x} = C_1 \textbf{ in } C_2 \mid \textbf{print } s; C \end{array}$$

# Getting Closer...

We now have an explanation for how we split up our STLC types into our two categories:

$$\begin{array}{llcl} \text{Positive} & A & ::= & A_1 \otimes A_2 \mid A_1 + A_2 \mid \mathbf{U}(X) \\ \text{Negative} & X & ::= & X_1 \times X_2 \mid A \to X \mid \mathbf{F}(A) \end{array}$$

$$\begin{array}{llcl} \text{Values} & V & ::= & x \mid V_1 \otimes V_2 \mid \textbf{Left } V \mid \textbf{Right } V \mid \textbf{susp}(C) \\ \text{Computations} & C & ::= & \langle C_1, C_2 \rangle \mid \textbf{fst}(C) \mid \textbf{snd}(C) \mid \lambda x : A.\ C \mid \\ & & & \textbf{ap}(C; V) \mid \textbf{split } V \textbf{ of } x_1, x_2 \Rightarrow C \mid \\ & & & \textbf{case } V \textbf{ of } \{x_1 \Rightarrow C_1 \mid x_2 \Rightarrow C_2\} \mid \textbf{force}(V) \mid \\ & & & \textbf{ret}(V) \mid \textbf{bind } \mathrm{x} = C_1 \textbf{ in } C_2 \mid \textbf{print } s; C \end{array}$$

## Question

What are these **F** and **U** types?

# What the **F** are **U** Talking About?

The type $\mathbf{U}(X)$ represents *suspended computations*:

# What the F are U Talking About?

The type $\mathbf{U}(X)$ represents *suspended computations*:

$$\frac{\Gamma \vdash C : X}{\Gamma \vdash \mathbf{susp}(C) : \mathbf{U}(X)} \qquad \frac{\Gamma \vdash V : \mathbf{U}(X)}{\Gamma \vdash \mathbf{force}(V) : X}$$

# What the **F** are **U** Talking About?

The type $\mathbf{U}(X)$ represents *suspended computations*:

$$\frac{\Gamma \vdash C : X}{\Gamma \vdash \mathbf{susp}(C) : \mathbf{U}(X)} \qquad \frac{\Gamma \vdash V : \mathbf{U}(X)}{\Gamma \vdash \mathbf{force}(V) : X}$$

The type $\mathbf{F}(A)$ represents computations which *return values of type* $A$:

# What the **F** are **U** Talking About?

The type $\mathbf{U}(X)$ represents *suspended computations*:

$$\frac{\Gamma \vdash C : X}{\Gamma \vdash \mathbf{susp}(C) : \mathbf{U}(X)} \qquad \frac{\Gamma \vdash V : \mathbf{U}(X)}{\Gamma \vdash \mathbf{force}(V) : X}$$

The type $\mathbf{F}(A)$ represents computations which *return values of type* $A$:

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{ret}(V) : \mathbf{F}(A)} \qquad \frac{\Gamma \vdash C_1 : \mathbf{F}(A) \quad \Gamma, x : A \vdash C_2 : X}{\Gamma \vdash \mathbf{bind}\ \mathrm{x} = C_1\ \mathbf{in}\ C_2 : X}$$

# What the **F** are **U** Talking About?

The type $\mathbf{U}(X)$ represents *suspended computations*:

$$\frac{\Gamma \vdash C : X}{\Gamma \vdash \mathbf{susp}(C) : \mathbf{U}(X)} \qquad \frac{\Gamma \vdash V : \mathbf{U}(X)}{\Gamma \vdash \mathbf{force}(V) : X}$$

The type $\mathbf{F}(A)$ represents computations which *return values of type* $A$:

$$\frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{ret}(V) : \mathbf{F}(A)} \qquad \frac{\Gamma \vdash C_1 : \mathbf{F}(A) \quad \Gamma, x : A \vdash C_2 : X}{\Gamma \vdash \mathbf{bind}\ \mathrm{x} = C_1\ \mathbf{in}\ C_2 : X}$$

**Idea**

The **U** and **F** type constructors give us a way to express computations as values (and vice versa)

# CBPV and Effects

**Idea**

In a CBPV program, the types can be used to distinguish pure expressions to potentially effectful ones

Let's write the example functions from the beginning in CBPV and look at their new types (assuming we have the value type **bool**):

$$\mathbf{e1} = \mathbf{Left\ true} : \mathbf{bool} + A$$

$$\mathbf{e2} = \mathbf{Left}\ (\mathbf{susp}(\mathbf{print}\ \texttt{"meow"}; \mathbf{ret}(\mathbf{true}))) : \mathbf{U}(\mathbf{F}(\mathbf{bool})) + A$$

$$\mathbf{e3} = \mathbf{print}\ \texttt{"meow"}; \mathbf{ret}(\mathbf{Left\ true}) : \mathbf{F}(\mathbf{bool} + A)$$

# CBPV and Compilers

# CBPV as an IR

When translating a piece of source code to our target code, we go through many *intermediate representations* (IRs) in order to get our source program closer to the target while maintaining its correctness.

[1] This is the translation for call-by-value (CBV) dynamics

# CBPV as an IR

When translating a piece of source code to our target code, we go through many *intermediate representations* (IRs) in order to get our source program closer to the target while maintaining its correctness.
One such IR can be the CBPV paradigm itself, where we take the source code and translate it into its CBPV equivalent[1]:

[1]This is the translation for call-by-value (CBV) dynamics

# CBPV as an IR

When translating a piece of source code to our target code, we go through many *intermediate representations* (IRs) in order to get our source program closer to the target while maintaining its correctness.
One such IR can be the CBPV paradigm itself, where we take the source code and translate it into its CBPV equivalent[1]:

**Translation**

If $\Gamma \vdash e : A$, then $\|\Gamma\| \vdash \|e\| : \mathbf{F}(\|A\|)$ in CBPV

[1] This is the translation for call-by-value (CBV) dynamics

# CBPV as an IR

When translating a piece of source code to our target code, we go through many *intermediate representations* (IRs) in order to get our source program closer to the target while maintaining its correctness.
One such IR can be the CBPV paradigm itself, where we take the source code and translate it into its CBPV equivalent[1]:

**Translation**

If $\Gamma \vdash e : A$, then $\|\Gamma\| \vdash \|e\| : \mathbf{F}(\|A\|)$ in CBPV

In other words, we represent each STLC program (i.e. expression) of type $A$ as a computation which returns type $\|A\|$.

[1] This is the translation for call-by-value (CBV) dynamics

# CBPV as an IR

When translating a piece of source code to our target code, we go through many *intermediate representations* (IRs) in order to get our source program closer to the target while maintaining its correctness.
One such IR can be the CBPV paradigm itself, where we take the source code and translate it into its CBPV equivalent[1]:

**Translation**

If $\Gamma \vdash e : A$, then $\|\Gamma\| \vdash \|e\| : \mathbf{F}(\|A\|)$ in CBPV

In other words, we represent each STLC program (i.e. expression) of type $A$ as a computation which returns type $\|A\|$.

**Question**

Why would it be useful to represent our program as a CBPV computation?

[1]This is the translation for call-by-value (CBV) dynamics

# Closure Conversion

The *closure* of a function is the environment at the time when the function was declared:

```
val x = 1
val y = 2
val f = fn z => x + y + z
```

# Closure Conversion

The *closure* of a function is the environment at the time when the function was declared:

```
val x = 1
val y = 2
val f = fn z => x + y + z
```

*Closure conversion* is a transformation where we equip function declarations with their closure, so that they don't have to depend on the environment anymore.

# CCBPV

In the CBPV IR, function types undergo the following translation:

# CCBPV

In the CBPV IR, function types undergo the following translation:

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

# CCBPV

In the CBPV IR, function types undergo the following translation:

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

Since functions are computations, if our program is a function, we *suspend* it with a **U** type.

# CCBPV

In the CBPV IR, function types undergo the following translation:

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

Since functions are computations, if our program is a function, we *suspend* it with a **U** type.

## Idea

Closure conversion happens exactly at **U** types

# CCPV

Our strategy will be to use existential types ($\exists t.A$) to represent the environment for a suspended computation, and to introduce a new type ($\mathbb{U}(X)$) for the type of packed closures (that no longer depend on free variables):

# CCPV

Our strategy will be to use existential types ($\exists t.A$) to represent the environment for a suspended computation, and to introduce a new type ($\mathbb{U}(X)$) for the type of packed closures (that no longer depend on free variables):

$$
\begin{array}{llll}
\text{Positive} & A & ::= & \ldots \\
 & & & t \\
 & & & \exists t.A \\
 & & & \mathbb{U}(X) \\
\text{Value} & V & ::= & \ldots \\
 & & & \textbf{pack}(A; V) \\
 & & & \textbf{close}(C) \\
\text{Computation} & C & ::= & \ldots \\
 & & & \textbf{unpack}(V; x.C) \\
 & & & \textbf{open}(V)
\end{array}
$$

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

# CCPV

We then have the following translation from **U** to our closure conversion IR:

$$\|\mathbf{U}(X)\| \rightsquigarrow \exists t.(t \otimes \mathbb{U}(t \to |X|))$$

# CCPV

We then have the following translation from **U** to our closure conversion IR:

$$\|\mathbf{U}(X)\| \rightsquigarrow \exists t.(t \otimes \mathbb{U}(t \to |X|))$$

For more details, see https://github.com/aricursion/CompileBPV

# Bonus

Paul Blain Levy, the originator of CBPV, suggested the slogan and some mneumonics in his doctoral thesis[2]:

> We suggest the following slogans and mnemonics.
>
> - A value is, a computation does.
> - $U$ types are thUnk types, $F$ types are producer types.
> - For cpos, $U$ means nUthing, $F$ means “liFt”.

[2]https://www.cs.bham.ac.uk/ pbl/papers/thesisqmwphd.pdf
[3]https://maxsnew.com/docs/mfps2023-slides.pdf

# Bonus

Paul Blain Levy, the originator of CBPV, suggested the slogan and some mneumonics in his doctoral thesis[2]:

We suggest the following slogans and mnemonics.

- A value is, a computation does.
- $U$ types are thUnk types, $F$ types are producer types.
- For cpos, $U$ means nUthing, $F$ means "liFt".

Max S. New also gave it a shot in his talk[3] on CBPV as an IR:

A value *is*

- A *UB* is a "thUnked" *B*

A computation *does*

- An *FA* "Feturns" *A* values

[2] https://www.cs.bham.ac.uk/ pbl/papers/thesisqmwphd.pdf
[3] https://maxsnew.com/docs/mfps2023-slides.pdf

# Conclusion

Friendship ended with
functions are values
Now
functions are computations
is my
best friend