Converting Combinatory Logic to and from Concatenative Calculus

Daniel Kiyoshi Hashimoto Vouzella de Andrade
Universidade Federal do Rio de Janeiro
Instituto de Computação
Rio de Janeiro, RJ, Brazil
dkhashimoto@ic.ufrj.br

Hugo Musso Gualandi
Universidade Federal do Rio de Janeiro
Instituto de Computação
Rio de Janeiro, RJ, Brazil
hugomg@ic.ufrj.br

Abstract

Combinatory logic and its combinators *BCKWI* are a foundation for tacit (point-free) programming. But what does each letter mean? Informally, *B* stands for composing, *C* for swapping, *K* for discarding, *W* for duplicating and *I* is the identity function. However, *B* does more than that: depending on where it appears in the expression, it can also call functions or defer values for later. To help tell these purposes apart, we relate combinatory logic to the concatenative calculus of Kerby, which features separate primitives for the different facets of *B*.

We provide translations from combinatory logic to concatenative calculus and vice versa. In both directions we have contributions. From concatenative to combinatory, we show that first-order stack programs map to regular combinators, and which patterns of combinators higher-order stack programs map to. From combinatory to concatenative, we extend an algorithm of Kerby to make it compatible with a call-by-value evaluation order, in addition to call-by-name.

To enforce a strong association between the two worlds, our translations are simulations. That is, one evaluation step in the original program is simulated by zero or more evaluations steps in the translated program.

Keywords: concatenative calculus, combinatory logic, stack languages, conversion, simulation, tacit programming, point-free programming

1 Introduction

Tacit programming, also known as point-free programming, is the practice of programming without named variables. It builds functions out of smaller combinators. For example, the function composition operator allow us to write $\lambda x \,.\, f(g(x))$ as $f \circ g$, without the variable x. Tacit programming has applications in the theory of computing, where variables may introduce mathematical complexity; and in day-to-day programming, where it may lead to concise programs that emphasize their control flow.

Combinatory logic. An important tacit model for functional programming is Combinatory Logic, created by Schönfinkel [11] and expanded and popularized by Curry [1]. This model is built on top of a set of basic combinators all named with a single letter:

$$\begin{array}{lcl@{\qquad}lcl} B\,q\,f\,x & \longrightarrow & q\,(f\,x) & S\;q\,f\,x & \longrightarrow & q\,x\,(f\,x) \\ C\,q\,x\,y & \longrightarrow & q\,y\,x & W\,q\,x & \longrightarrow & q\,x\,x \\ K\,q\,x & \longrightarrow & q & I\;\;x & \longrightarrow & x \end{array}$$

We can understand combinators by their effects. C permutes x and y, K discards x, W duplicates x, and I is the identity function. B is often described as function composition, but it actually does more than that. For starters, if f is a curried function, the composition may also be seen as partial application. Furthermore, B can help other combinators reach deeper arguments. For example, in $B\,K$ the B makes K discard the second argument after q, instead of the first:

$$B\,K\,q\,x\,y \;\longrightarrow\; K\,(q\,x)\,y \;\longrightarrow\; q\,x$$

We want to be able to look at a combinator and intuitively understand what it means. A first obstacle is that some combinators serve more than one purpose. That was already bad for *B*, and it gets even worse for *S*. A famous result of combinatory logic is that any combinator can be expressed in terms of only *S* and *K*. Thus, combinations of *S* and *K* can literally do anything.

A second obstacle is that the combinator basis *BCKWI* encourages continuation-passing style, because their first argument behaves like a continuation callback. This is serviceable, but can take some time to get used to.

Higher-order concatenative programming. The higher-order concatenative (stack) languages, such as Joy [12, 13] and Factor [8], are also well-suited for tacit programming, but unlike combinatory logic they have separate operators for the things that B does: composition, partial application and "digging". We will illustrate with examples from Kerby's concatenative calculus [3].

In the concatenative calculus, as in any concatenative language, programs are sequences of instructions in reverse-polish notation. To compose two programs, we concatenate their sequences. For instance, the sequence `swap dup` is the composition of `swap` and `dup`:

$$\texttt{3 4 swap dup} \mapsto \texttt{4 3 dup} \mapsto \texttt{4 3 3}$$

The calculus uses square brackets to write anonymous subroutines, which it calls "quotations". In the following example, `[+]` is a quotation that adds two numbers. The `cons` operator performs partial application: it combines that quotation with a first number, producing an one-argument quotation that expects the remaining number.

$$\texttt{7 [+] cons} \mapsto \texttt{[7 +]}$$

The `dip` operator runs a quotation one level deeper into the stack. Whereas `zap` discards the topmost argument, the program `[zap] dip` discards the second from the top. This `[] dip` pattern is analogous to the B in the $B\,K$ combinator.

$$\texttt{3 4 [zap] dip} \mapsto \texttt{3 zap 4} \mapsto \texttt{4}$$

As we have started to see, concatenative instructions can give a new point of view into combinatory logic combinators. At the basic level, it appears that C is related to `swap`, K to `zap`, W to `dup`. B seems to be related to composition, partial application or `dip`, depending on the situation. In this work, we formalize this correspondence and generalize it to more complex combinators. We describe two relations: one that maps an arbitrary concatenative program into combinators, and one that maps an arbitrary combinator into a concatenative program. We show that our relations are simulations: one reduction step in the original program maps to zero or more reduction steps in the output. This preserves the relation between the two sides at each step of their evaluation. Thus, we ensure that a concatenative program like `swap swap` should relate to a combinator that calls C twice, instead of one that optimizes the swaps into a no-op.

From concatenative programs to regular combinators. The first conversion that we demonstrate maps stack programs into combinatory expressions. We show that first-order stack programs correspond to the well-studied *regular combinators*, which follow a continuation-passing discipline. We then extend our translation to higher-order programs, supporting unrestricted use of quotations and the `call`, `dip` and `cons` instructions. The resulting combinators follow a higher-order variant of continuation-passing style: it may assume that a non-continuation argument is a continuation-passing combinator.

Interestingly, our conversion can also be played backwards, in which case it converts a continuation-passing combinator into a direct-style concatenative program.

From combinatory to concatenative. Many combinators are not regular. To fill that gap, we will describe three methods to convert an arbitrary combinator into a concatenative program. The first of them, due to Kerby, outputs a concatenative program that emulates a call-by-name reduction for the original combinator. We also present two novel methods that emulate a call-by-value order. Whereas call-by-name always reduces the outermost combinator expression, call-by-value reduces the innermost one. The main challenge we had to solve was how to identify when an evaluation step will produce an inner reducible expression, because that impacts whether the conversion should use the `cons` or the `call` instruction. The first of our call-by-value methods works for every combinator, but requires that we modify the concatenative calculus so that the reducibility of a quotation expression may be inspected at run-time. Our second call-by-value method uses compile-time type inference to compute which sub-expressions will be reducible. It is restricted to simply-typed combinators, but can be applied to the unmodified concatenative calculus.

Contributions. We present a formal model of how combinatory logic relates to the concatenative calculus.

- A mapping between regular combinators and first-order concatenative programs, and a mapping between a superset of regular combinators and higher-order programs. (Section 3)
- Methods for compiling arbitrary combinators into the concatenative calculus stack machine, using call-by-value evaluation order. (Section 4)

In Section 2 we review combinatory logic and the concatenative calculus. In Section 5, we summarize our results and discuss related work. A preliminary version of this work can be found in Hashimoto Vouzella de Andrade [2].

2 Definitions

In this section, we review combinatory logic and the concatenative calculus, including their syntax, semantics, and important definitions.

2.1 Combinatory Logic

A **combinator** is either a basic combinator or an application of two smaller combinators. Different variants of combinatory logic chose different sets of basic combinators. The minimalist bases *SK* and *SKI* are popular choices. In this paper we will focus on the classic basis *BCKWI*; we want to draw connections with stack-based programming and for that goal we would rather have distinct combinators that do one thing each, instead of few combinators that do many things.

Uppercase letters stand for combinators and lower case letters for variables (x, y, z). Similarly to the lambda-calculus, application is written with spaces and is left-associative.

A **proper combinator** is a combinator which does not introduce new terms into its output: everything comes from the input. All of the basic combinators are proper, but applications of them might not be. For instance, $K\,I$ leaves an I in its output: $K\,I\,x \longrightarrow I$.

A **regular combinator** is a combinator whose first argument appears exactly once in the output, also in the initial position. This is the case for all basic combinators, for example: $C\,q\,x\,y \longrightarrow q\,y\,x$. The first argument of a regular combinator acts as a callback or continuation. To emphasize that, we use the letter q. (Alas, c and k were already taken.) In the literature, regular combinators are sometimes

Instructions:	`swap` `zap` `dup` `apply` `call` `dip` `cons`
Values (x, y, f, g):	quotation `[` P `]`, or opaque function f
Programs (P, Q, R):	sequence of instructions and values

Figure 1. Syntax of the Concatenative Calculus

$$
\begin{array}{rcl}
y\ x\ \texttt{swap} & \mapsto & x\ y \\
x\ \texttt{zap} & \mapsto & \\
x\ \texttt{dup} & \mapsto & x\ x \\
x\ f\ \texttt{apply} & \mapsto & f(x) \\
\texttt{[}\ P\ \texttt{]}\ \texttt{call} & \mapsto & P \\
x\ \texttt{[}\ P\ \texttt{]}\ \texttt{dip} & \mapsto & P\ x \\
x\ \texttt{[}\ P\ \texttt{]}\ \texttt{cons} & \mapsto & \texttt{[}\ x\ P\ \texttt{]}
\end{array}
$$

$$
\begin{array}{l}
\text{if } P \mapsto Q \text{ then } P\,R \mapsto Q\,R \\
\text{if } P \mapsto Q \text{ then } R\,P \mapsto R\,Q
\end{array}
$$

Figure 2. Reduction rules for the concatenative calculus

required to be proper. However, we allow improper combinators because we want to be able to embed constants inside our combinators.

2.2 The Concatenative Calculus

In this section, we summarize the version of concatenative calculus that we will use in this paper. Figure 1 describes its syntax. A **concatenative program** is a sequence of instructions and values. Uppercase metavariables (`P`, `Q`, `R`) represent arbitrary programs. The empty program is ε. The metavariables x, y, z, w represent values on the stack and f, g represent function values.

Concatenative programs represent stack transformations. In Figure 2, we describe an operational semantics for the calculus. Instead of working with an explicit stack, the relation $\mapsto$ describes a reduction from one stack transformation to another.

The `swap`, `zap` and `dup` instructions are typical stack shuffling operations. Intuitively, the arguments to the left of an instruction represent the top of the stack: `swap` exchanges the top two values on the stack, `zap` discards the top value and `dup` duplicates the top value.

To support higher-order programming, the concatenative calculus features quotations and instructions to manipulate them. A **quotation** is a subprogram, written between square brackets. They play a similar role to anonymous functions. The `call` instruction runs a quotation, removing its square brackets: x `[dup]` `call` $\mapsto$ x `dup` $\mapsto$ $x\,x$. Note that in the concatenative calculus, composition is merely concatenation while calling is an explicit instruction.

The `dip` instruction works similarly to `call` except that the topmost value is not passed to the quotation and is preserved for the rest of the program. Observe how in the following example, the x is left untouched: $y\,x$ `[dup]` `dip` $\mapsto$ y `dup` x. Moreover, we can nest `dip` instructions to access deeper values: $z\,y\,x$ `[` `[dup]` `dip` `]` `dip` $\mapsto$ $z\,y$ `[dup]` `dip` x $\mapsto$ z `dup` $y\,x$.

The `cons` instruction partially applies a quotation. It receives a quotation and a value and puts the value inside the quotation: x `[swap]` `cons` $\mapsto$ `[` x `swap` `]`. Now the partially applied `swap` only asks for one more argument to be called: y `[` x `swap` `]` `call` $\mapsto$ $y\,x$ `swap`.

We extend Kerby's concatenative calculus with opaque functions. Unlike quotations, whose code is concatenative, **opaque functions** stand for an external or builtin operations. We model them as arbitrary mathematical functions and introduce an `apply` instruction to run them:

$$x\ f\ \texttt{apply} \mapsto f(x)$$

The need for separating `call` and `apply` will become more clear on Section 3, where we use `call` with continuation-passing style programs while we use `apply` for functions not in continuation-passing style.

Finally, we have two structural rules, which say that a reduction `P` $\mapsto$ `Q` can still occur if we concatenate a program `R` to the left or right of `P`. These rules allow us to make reductions in the middle of the instruction list, but not inside quotations.

We can divide the concatenative programs into two kinds. A **first-order program** is composed of only opaque functions, `swap`, `zap`, `dup`, `apply` or a quotation immediately followed by a `dip`. These programs model stack languages that are not able to push and manipulate function objects on the stack, such as classic FORTH [7, 10]. Only a restricted use of `dip` is permitted, to allow access to deeper stack values. Conversely, a **higher-order program** allows unrestricted use of quotations, `call`, `dip` and `cons`. They model higher-order stack languages such as Joy [12, 13] and Factor [8].

3 From Concatenative to Combinatory

In this section, we relate regular combinators and concatenative programs and we will show that this relation is a simulation: one evaluation step on the concatenative side corresponds to zero or more on the combinatory side. We will start with first-order concatenative programs, and then we will extend it to the higher-order ones.

3.1 First-order programs

Recall that regular combinators keep their first argument in place after the reduction. For instance, $C\,q\,x\,y\,z\,w \longrightarrow q\,y\,x\,z\,w$. If we interpret this first argument q as a continuation, we can think of C as a variadic function written in continuation-passing style, which returns values by passing them to the continuation. In the previous example, we receive $(x\,y)$ and pass to the continuation the proper results $(y\,x)$ followed by the unused arguments $(z\,w)$. This is analogous to the concatenative `swap` instruction, which exchanges the two values at the top of the stack, without touching deeper values: $w\,z\,y\,x$ `swap` $\mapsto$ $w\,z\,x\,y$.

(a) values

$$\textsc{v-function}\quad f \sim f \qquad\qquad \textsc{v-quot}\quad \frac{\alpha \leftrightarrow \mathtt{P}}{\alpha \sim [\ \mathtt{P}\ ]}$$

(b) first-order programs

$$\textsc{r-apply}\quad B \leftrightarrow \mathtt{apply} \qquad \textsc{r-swap}\quad C \leftrightarrow \mathtt{swap} \qquad \textsc{r-zap}\quad K \leftrightarrow \mathtt{zap} \qquad \textsc{r-dup}\quad W \leftrightarrow \mathtt{dup}$$

$$\textsc{r-empty}\quad I \leftrightarrow \varepsilon \qquad \textsc{r-concat}\quad \frac{\alpha \leftrightarrow \mathtt{P} \qquad \beta \leftrightarrow \mathtt{Q}}{B\,\alpha\,\beta \leftrightarrow \mathtt{P\ Q}} \qquad \textsc{r-quot-dip}\quad \frac{\alpha \leftrightarrow \mathtt{P}}{B\,\alpha \leftrightarrow [\ \mathtt{P}\ ]\ \mathtt{dip}}$$

$$\textsc{r-push}\quad \frac{\alpha \sim x \qquad \beta \leftrightarrow \mathtt{Q}}{C\,\beta\,\alpha \leftrightarrow x\ \mathtt{Q}}$$

(c) higher-order instructions

$$\textsc{r-call}\quad C\,I \leftrightarrow \mathtt{call} \qquad \textsc{r-dip}\quad C\,B \leftrightarrow \mathtt{dip} \qquad \textsc{r-cons}\quad C\,(B\,B\,B)\,C \leftrightarrow \mathtt{cons}$$

(d) simulation

$$\textsc{s-reg}\quad \frac{\alpha \leftrightarrow \mathtt{P}}{\alpha\,q \Leftrightarrow \mathtt{P}} \qquad \textsc{s-empty}\quad q \Leftrightarrow \varepsilon \qquad \textsc{s-push}\quad \frac{\alpha \sim x}{q\,\alpha \Leftrightarrow x} \qquad \textsc{s-concat}\quad \frac{\hat{\alpha} \Leftrightarrow \mathtt{P} \qquad \hat{\beta} \Leftrightarrow \mathtt{Q}}{\hat{\alpha}\{\hat{\beta}/q\} \Leftrightarrow \mathtt{P\ Q}}$$

Figure 3. Regular combinators and concatenative programs

We can build upon this pattern to find matching combinators for any concatenative program. In Figure 3a, we show the relation $\alpha \sim x$, which relates a combinator α to a concatenative value x. Opaque functions are the same on both sides and a quotation relates to the same regular combinator as its subprogram. Figure 3b defines a relation $\alpha \leftrightarrow \mathtt{P}$ that relates a regular combinator to a concatenative program. We map B, C, K, W to the corresponding primitive instructions and I to the empty program. It is known that the two-argument form of B composes two regular combinators and the one-argument form defers values [9]. Thus, we map two-argument B to concatenation and one-argument B to `dip`. The R-PUSH rule is here because we allow improper combinators, which contain embedded values. It describes the instruction that pushes a value to the stack. Notice how $C\,\beta\,\alpha\,q$ reduces to $\beta\,q\,\alpha$, inserting α at the top of the list of non-q arguments for β. Thus, if α is the combinator for a stack value x, and β is the combinator for program `Q`, then $C\,\beta\,\alpha$ stands for pushing x to the stack before running `Q`. Figure 3c describes the higher-order part of the calculus, which we will cover in Section 3.2.

Unfortunately, the relation $\leftrightarrow$ is not suitable for a simulation because evaluation steps in the concatenative side might not correspond to those on the combinatory side. Consider the reduction $x\,\mathtt{zap} \mapsto \varepsilon$ and the associated combinators $C\,K\,x$ and I. As one would hope, if we supply the continuation argument to kickstart the evaluation, the two combinators eventually arrive at a common value q. However, this is not a simulation because $C\,K\,x\,q$ cannot reduce to $I\,q$ without "walking backwards".

$$\begin{aligned} C\,K\,x\,q \longrightarrow K\,q\,x &\longrightarrow q \\ I\,q &\longrightarrow q \end{aligned}$$

To address this deficiency, we will no longer treat the continuation as an ordinary argument and will allow it to appear inside the combinator. In Figure 3d, we describe a relation $\hat{\alpha} \Leftrightarrow \mathtt{P}$ which relates a concatenative program `P` to a combinator $\hat{\alpha}$ that contains the special continuation variable q exactly once. The circumflex hat is a reminder that there is a q nested inside $\hat{\alpha}$. This relation extends relation $\leftrightarrow$ and adds new cases so that we never need to "walk backwards". The rule S-REG covers the case when the expression is a regular combinator applied to the continuation. This rule makes $\Leftrightarrow$ be an extension of $\leftrightarrow$. The remaining rules fix the problematic cases that had to walk backwards. S-EMPTY matches the empty program without using I. S-PUSH describes how to push values to the stack without C. The last rule S-CONCAT describes how to concatenate without B. The notation $\hat{\alpha}\{\hat{\beta}/q\}$ stands for substituting $\hat{\beta}$ for q inside $\hat{\alpha}$. For example, from $C\,q \Leftrightarrow \mathtt{swap}$ and $K\,q \Leftrightarrow \mathtt{zap}$, we can deduce $C\,(K\,q) \Leftrightarrow \mathtt{swap\ zap}$.

Let's illustrate these rules with a concrete example. Now that the continuation q can be nested inside the combinator, each reduction step in the combinatory side matches a reduction step in the concatenative side:

$$\begin{aligned} K\,(C\,(W\,q))\,x\,y\,z &\quad\Leftrightarrow\quad z\ y\ x\ \mathtt{zap\ swap\ dup} \\ C\,(W\,q)\,y\,z &\quad\Leftrightarrow\quad z\ y\ \mathtt{swap\ dup} \\ W\,q\,z\,y &\quad\Leftrightarrow\quad y\ z\ \mathtt{dup} \\ q\,z\,z\,y &\quad\Leftrightarrow\quad y\ z\ z \end{aligned}$$

Non-injectivity of $\alpha \leftrightarrow \mathtt{P}$**.** It is possible to have two different combinators that relate to the same concatenative program, because R-CONCAT allows multiple choices for `P` and `Q`. This reflects how function composition is associative ($B\,\alpha\,(B\,\beta\,\gamma) \equiv B\,(B\,\alpha\,\beta)\,\gamma$) and has the identity function as a neutral element ($B\,\alpha\,I \equiv \alpha \equiv B\,I\,\alpha$).

Non-injectivity of $\hat{\alpha} \Leftrightarrow \mathtt{P}$**.** The $\Leftrightarrow$ relation introduces even more ways to represent the same concatenative program. For example, we have both $B\,K\,W\,q \Leftrightarrow \mathtt{zap\ dup}$ (using R-CONCAT and S-REG) and $K\,(W\,q) \Leftrightarrow \mathtt{zap\ dup}$ (using only S-CONCAT). But notice that the former reduces to the latter. We can state that as a lemma:

Lemma 3.1 (q-normal-form)**.** *If $\hat{\alpha} \Leftrightarrow \mathtt{P}$, there exists $\hat{\beta}$ such that $\hat{\alpha} \longrightarrow^{*} \hat{\beta}$ and $\hat{\beta} \Leftrightarrow \mathtt{P}$ where the rules R-EMPTY, R-CONCAT, and, R-PUSH are only used inside quotations.*

Proof. By induction in the size of $\hat{\alpha}$. Suppose that $\hat{\alpha} \Leftrightarrow \mathtt{P}$ is not in q-normal-form. There must be at least one instance where rule S-REG was used together with R-EMPTY, R-CONCAT, or R-PUSH. Suppose that it was R-CONCAT. We would have $B\,\alpha\,\beta\,q \Leftrightarrow \mathtt{P\,Q}$ with $\alpha \leftrightarrow \mathtt{P}$ and $\beta \leftrightarrow \mathtt{Q}$ and thus $(\alpha\,q) \Leftrightarrow \mathtt{P}$ and $(\beta\,q) \Leftrightarrow \mathtt{Q}$. From the inductive hypothesis, $(\alpha\,q) \longrightarrow^* \hat{\alpha}$ and $(\beta\,q) \longrightarrow^* \hat{\beta}$, both in q-normal-form. Therefore, $B\,\alpha\,\beta\,q \longrightarrow \alpha\,(\beta\,q) \longrightarrow^* \hat{\alpha}\{\hat{\beta}/q\} \Leftrightarrow \mathtt{P\,Q}$, the latter which is in q-normal-form. A similar argument can be used for the R-EMPTY and R-PUSH cases. □

Lemma 3.2. *If $\hat{\delta} \Leftrightarrow \mathtt{P\,Q}$ and $\hat{\delta}$ is in q-normal-form then $\hat{\delta} = \hat{\alpha}\{\hat{\beta}/q\}$ with $\hat{\alpha} \Leftrightarrow \mathtt{P}$ and $\hat{\beta} \Leftrightarrow \mathtt{Q}$.*

Proof. By induction in the length of $\mathtt{P\,Q}$. In the base case at least one of $\mathtt{P}$ or $\mathtt{Q}$ are empty, and we can choose $\hat{\alpha} = q$ and $\hat{\beta} = \hat{\delta}$ or vice-versa. In the inductive case, neither is empty. Because $\hat{\delta}$ is in q-normal form, the only rule that matches the concatenation is S-CONCAT. However, it might not split right between $\mathtt{P}$ and $\mathtt{Q}$. Suppose, without loss of generality, that it splits $\mathtt{P} = \mathtt{X\,Y}$. It will have split $\hat{\delta}$ into $\hat{X}$ and $\widehat{YQ}$, with $\hat{X} \Leftrightarrow \mathtt{X}$, $\widehat{YQ} \Leftrightarrow \mathtt{Y\,Q}$, and $\hat{X}\{\widehat{YQ}/q\} \Leftrightarrow \mathtt{X}\,(\mathtt{Y\,Q})$. But we can fix that. According to the inductive hypothesis from $\widehat{YQ} \Leftrightarrow \mathtt{Y\,Q}$, we can find $\hat{Y} \Leftrightarrow \mathtt{Y}$ and $\hat{Q} \Leftrightarrow \mathtt{Q}$ with $\widehat{YQ} = \hat{Y}\{\hat{Q}/q\}$. Thus, we currently have $\hat{X}\{(\hat{Y}\{\hat{Q}/q\})/q\} \Leftrightarrow \mathtt{X}\,(\mathtt{Y\,Q})$. Since substitution and concatenation are associative, that also gives us $(\hat{X}\{\hat{Y}/q\})\{\hat{Q}/q\} \Leftrightarrow (\mathtt{X\,Y})\,\mathtt{Q}$. Considering that $\mathtt{X\,Y} = \mathtt{P}$, we can fulfill our goal with $\hat{\alpha} = \hat{X}\{\hat{Y}/q\}$ and $\hat{\beta} = \hat{Q}$. We must only confirm $\hat{\alpha} \Leftrightarrow \mathtt{P}$, which follows from $\hat{X} \Leftrightarrow \mathtt{X}$ and $\hat{Y} \Leftrightarrow \mathtt{Y}$. □

We are now ready to prove our main simulation theorem, which says that if a combinator relates to a concatenative program then one step in the concatenative side may be simulated by zero or more steps in the combinatory side:

Theorem 3.3 (first-order simulation)**.** *If $\hat{\delta} \Leftrightarrow \mathtt{P}$ and $\mathtt{P} \mapsto \mathtt{Q}$ then there exists $\hat{\eta}$ such that $\hat{\delta} \longrightarrow^* \hat{\eta}$ and $\hat{\eta} \Leftrightarrow \mathtt{Q}$.*

Proof. By induction on $\mathtt{P} \mapsto \mathtt{Q}$. The base case is a reduction of a primitive instruction. We will show the proof for `swap`; the others are similar. Given any ($\hat{\alpha} \Leftrightarrow y\,x\,\mathtt{swap}$), the q-normal-form lemma guarantees that $\hat{\alpha} \longrightarrow^* C\,q\,x\,y$. That in turn reduces to $q\,y\,x$ which corresponds to $x\,y$.

$$\begin{array}{rcl} \hat{\alpha} & \Leftrightarrow & y\,x\,\mathtt{swap} \\ C\,q\,x\,y & \Leftrightarrow & y\,x\,\mathtt{swap} \\ q\,y\,x & \Leftrightarrow & x\,y \end{array}$$

Now consider the structural rules $\mathtt{P\,R} \mapsto \mathtt{Q\,R}$ and $\mathtt{R\,P} \mapsto \mathtt{R\,Q}$. We will show the proof for the former. Given $\mathtt{P\,R} \mapsto \mathtt{Q\,R}$ and $\hat{\delta} \Leftrightarrow \mathtt{P\,R}$, lemmas 3.1 and 3.2 tell us that $\hat{\delta}$ reduces to $\hat{\alpha}\{\hat{\gamma}/q\}$ such that $\hat{\alpha} \Leftrightarrow \mathtt{P}$ and $\hat{\gamma} \Leftrightarrow \mathtt{R}$. By the induction hypothesis, there exists $\hat{\beta} \Leftrightarrow \mathtt{Q}$ such that $\hat{\alpha} \longrightarrow^* \hat{\beta}$. If we replay this reduction sequence step-by-step but substitute $\hat{\gamma}$ for q, we conclude that $\hat{\alpha}\{\hat{\gamma}/q\} \longrightarrow^* \hat{\beta}\{\hat{\gamma}/q\}$. We are now done.

$$\begin{array}{rcl} \hat{\delta} & \Leftrightarrow & \mathtt{P\,R} \\ \hat{\alpha}\{\hat{\gamma}/q\} & \Leftrightarrow & \mathtt{P\,R} \\ \hat{\beta}\{\hat{\gamma}/q\} & \Leftrightarrow & \mathtt{Q\,R} \end{array}$$

The last case left is `dip`. From now on, we skip the first step of the proof, where we apply the q-normal-form lemma. After that step, it is left to show that given $\alpha \leftrightarrow \mathtt{P}$, the following commutes:

$$\begin{array}{rcl} B\,\alpha\,q\,x & \Leftrightarrow & x\,[\,\mathtt{P}\,]\,\mathtt{dip} \\ \alpha\,(q\,x) & \Leftrightarrow & \mathtt{P}\,x \end{array}$$

From $\alpha \leftrightarrow \mathtt{P}$ and S-REG, we obtain $(\alpha\,q) \Leftrightarrow \mathtt{P}$. Then, with $(q\,x) \Leftrightarrow x$, and S-CONCAT we arrive at $\alpha\,(q\,x) \Leftrightarrow \mathtt{P}\,x$. □

3.2 Higher order programs

To compile higher-order concatenative programs into combinators, we must provide rules for the `call` and `cons` instructions and we must allow `dip` to appear by itself, without being immediately preceded by a quotation. We list the new rules in in Figure 3c.

The simplest higher-order instruction is `call`. We want a combinator that receives a continuation q and the combinator α for the quotation; and then transfers the control to α with q as its continuation. The answer is $C\,I$. In the diagram below, we assume $\alpha \leftrightarrow \mathtt{P}$, and show that $C\,I$ simulates the concatenative reduction step.

$$\begin{array}{rcl} C\,I\,q\,\alpha & \Leftrightarrow & [\,\mathtt{P}\,]\,\mathtt{call} \\ I\,\alpha\,q & & \\ \alpha\,q & \Leftrightarrow & \mathtt{P} \end{array}$$

Next, let's examine the `dip` instruction. We want a combinator that receives q, α, and a value x; and then transfer control to α, but with $q\,x$ as the continuation. This way, the x is not passed to the quotation α and is left at the top of the stack after α returns (that is, calls the continuation). The combinator $C\,B$ can perform this task. In the diagram, we assume $\alpha \leftrightarrow \mathtt{P}$ and $\varphi \sim x$ and show that $C\,B$ simulates the reduction step on the right.

$$\begin{array}{rcl} C\,B\,q\,\alpha\,\varphi & \Leftrightarrow & x\,[\,\mathtt{P}\,]\,\mathtt{dip} \\ B\,\alpha\,q\,\varphi & & \\ \alpha\,(q\,\varphi) & \Leftrightarrow & \mathtt{P}\,x \end{array}$$

An attentive reader might notice that the $B\,\alpha\,q\,\varphi$ in the second line is the first-order translation for `dip`, as described by R-QUOT-DIP. This confirms that the higher-order instruction generalizes the first-order one.

Lastly, we must find a combinator for `cons`, which constructs a new quotation $[\,x\,\mathtt{P}\,]$ and pushes it to the stack. In combinatory terms, this means using R-PUSH to construct the combinator $C\,\alpha\,\varphi$ and then S-PUSH to pass it to the continuation q. To find a combinator that does this, we used the bracket-abstraction algorithm [1, 9]. Working backwards from $q\,(C\,\alpha\,\varphi)$, we found $C\,(B\,B\,B)\,C$.

$$\begin{array}{rcl}
C\,(B\,B\,B)\;C\;q\;\alpha\;\varphi & \Leftrightarrow & x\;\texttt{[P] cons} \\
B\,B\,B\;q\;C\;\alpha\;\varphi & & \\
B\,(B\,q)\;C\;\alpha\;\varphi & & \\
B\;q\;(C\;\alpha)\;\varphi & & \\
q\;(C\;\alpha\;\varphi) & \Leftrightarrow & \texttt{[} x \texttt{ P]}
\end{array}$$

3.3 $\alpha \leftrightarrow$ P as a function

If we use only the rules from Figures 3a and 3b, the relation $\leftrightarrow$ covers every first-order concatenative program. If we also add the higher-order rules from Figure 3c, we can cover every concatenative program.

On the concatenative side of the $\leftrightarrow$ relation, the combinators are almost always regular combinators. The sole exception are $C\,I$, $C\,B$ and $C\,(B\,B\,B)\;C$ from the rules R-CALL, R-DIP and R-CONS. They technically are not regular, but behave as such if their second argument is regular.

Taking this into account, we can view $\leftrightarrow$ either as a partial function from regular combinators to concatenative programs, or as a total function from concatenative programs to a set of equivalent combinators. We think the latter interpretation is particularly interesting because the our main goal when we designed $\leftrightarrow$ was to match every concatenative program. In the following figure, we provide an alternative presentation of $\leftrightarrow$ using function notation. Since programs might match more than one of the cases, there are two ways to interpret this. The first option is to treat it as a non-deterministic transformation that can match any of the cases. The other is to introduce an order of preference between the cases (match the first case, pick the smallest non-empty P for $[\![\texttt{P Q}]\!]$). This turns it into a function that returns a single representative combinator.

$$\begin{array}{lcllcl}
[\![\texttt{apply}]\!] & = & B & [\![\texttt{P Q}]\!] & = & B\,[\![\texttt{P}]\!]\;[\![\texttt{Q}]\!] \\
[\![\texttt{swap}]\!] & = & C & [\![\texttt{[P] dip}]\!] & = & B\,[\![\texttt{P}]\!] \\
[\![\texttt{zap}]\!] & = & K & [\![\texttt{dip}]\!] & = & C\,B \\
[\![\texttt{dup}]\!] & = & W & [\![\texttt{call}]\!] & = & C\,I \\
[\![\varepsilon]\!] & = & I & [\![\texttt{cons}]\!] & = & C\,(B\,B\,B)\;C \\
[\![x\;\texttt{P}]\!] & = & C\,[\![\texttt{P}]\!]\;x & & &
\end{array}$$

4 From Combinatory to Concatenative

In this section, we describe how to convert any combinator to concatenative program, including irregular combinators. We start by presenting a conversion developed by Kerby which behaves in a call-by-name manner. We then adapt this technique for call-by-value.

In call-by-name, we always reduce the outermost redex. In call-by-value, we reduce innermost redexes first. That is, in call-by-value the arguments of a basic combinator must be fully reduced before reducing the basic combinator. Unlike some treatments of call-by-value, we do not mandate a left-to-right or a right-to-left order.

The difference between call-by-name and call-by-value is most apparent for B. In Figure 4, we compare the two

(a) call-by-name	**(b)** call-by-value
$B\,(B\,C)\;K\;x\;y\;z\;w$	$B\,(B\,C)\;K\;x\;y\;z\;w$
$B\,C\,(K\,x)\;y\;z\;w$	$B\,C\,(K\,x)\;y\;z\;w$
$C\,(K\,x\,y)\;z\;w$	$C\,(K\,x\,y)\;z\;w$
$K\,x\,y\,w\,z$	$C\,x\,z\,w$
$x\,w\,z$	$x\,w\,z$

Figure 4. Evaluation orders for $B\,(B\,C)\;K$

$$\begin{array}{lcllcl}
\text{«}\,B\,\text{»} & := & \texttt{[cons] dip call} & \text{«}\,W\text{»} & := & \texttt{[dup] dip call} \\
\text{«}\,C\,\text{»} & := & \texttt{[swap] dip call} & \text{«}\,I\,\text{»} & := & \texttt{call} \\
\text{«}\,K\,\text{»} & := & \texttt{[zap] dip call} & \text{«}\alpha\;\beta\text{»} & := & \texttt{[}\text{«}\beta\text{»}\texttt{]}\;\text{«}\alpha\text{»}
\end{array}$$

Figure 5. Kerby's call-by-name conversion

reduction orders for $B\,(B\,C)\;K$. The first B partially applies K to x, producing $(K\,x)$. The second B produces $(K\,x\,y)$, which has enough arguments to reduce. Now, the evaluation orders take different paths. In call-by-name the $(K\,x\,y)$ is an inner expression and won't be evaluated just yet. Effectively, it is also a partial application. Conversely, call-by-value wants to reduce $(K\,x\,y)$ right away. This poses a problem: the first B wants to partially apply, while the second B does not. In concatenative terms, this boils down to a choice between `cons` and `call`.

In the following subsections, we will describe three conversion algorithms. The first is a call-by-name translation, due to Kerby [3]. The second is a call-by-value translation that chooses between `cons` and `call` at run-time. And the third is a call-by-value translation that chooses at compile-time.

4.1 Call-by-name

When Kerby introduced the concatenative calculus, he also described an algorithm to convert a combinator to a concatenative program [3]. We summarize his method in Figure 5. He represents terms as quotations. The basic combinators B, C, K and W are translated to a dipped instruction (to jump over the q argument) followed by a `call` to evaluate the resulting term. I can be translated to the empty program. In an application, we leave the leftmost combinatory term unquoted, and its argument quoted. If the left argument is another application, we recursively flatten the entire left spine of the tree.

Let's go through the example in Figure 6. In the first line, the compiled versions of q, x and y are quoted until it is time for them to be evaluated. The combinator C gets compiled to `[swap] dip call` and, because it is at the head of the list, it is not quoted. The `[swap] dip` performs the swapping duties and the final `call` unquotes the new head of the list («q»). This method emulates a call-by-name evaluation order because it always reduces the outermost combinator expression. After all, the other ones are all quoted and cannot be reduced.

$$
\begin{aligned}
«C\,q\,x\,y» &= \texttt{[}«y»\texttt{]}\ \texttt{[}«x»\texttt{]}\ \texttt{[}«q»\texttt{]}\ \texttt{[swap] dip call}\\
&\ \texttt{[}«y»\texttt{]}\ \texttt{[}«x»\texttt{]}\ \texttt{swap}\ \texttt{[}«q»\texttt{]}\ \texttt{call}\\
&\ \texttt{[}«x»\texttt{]}\ \texttt{[}«y»\texttt{]}\ \texttt{[}«q»\texttt{]}\ \texttt{call}\\
«q\,y\,x» &= \texttt{[}«x»\texttt{]}\ \texttt{[}«y»\texttt{]}\ «q»
\end{aligned}
$$

Figure 6. A call-by-name example

$$
\begin{aligned}
x\ \texttt{[P]}_n\ \star &\mapsto x\ \texttt{P} && \text{if } n = 1 \text{ and } x \text{ is a value}\\
x\ \texttt{[P]}_n\ \star &\mapsto \texttt{[}\ x\ \texttt{P]}_{n-1} && \text{if } n \geq 2 \text{ and } x \text{ is a value}
\end{aligned}
$$

Figure 7. Dynamic application

$$
\begin{aligned}
\langle B \rangle &:= \texttt{[[}\star\texttt{] dip}\,\star\ \texttt{]}_3 & \langle K \rangle &:= \texttt{[[zap] dip]}_2\\
\langle C \rangle &:= \texttt{[[swap] dip}\,\star\star\ \texttt{]}_3 & \langle I \rangle &:= \texttt{[]}_1\\
\langle W \rangle &:= \texttt{[[dup] dip}\,\star\star\ \texttt{]}_2 & \langle \alpha\ \beta \rangle &:= \langle \beta \rangle\ \langle \alpha \rangle\ \star
\end{aligned}
$$

Figure 8. Dynamic call-by-value conversion

4.2 Dynamic call-by-value

A basic combinator can only be reduced once it has received enough arguments. If we represent a partially applied combinator as a quotation, the question becomes how to know if a quotation already has enough arguments to be called. If we are willing to modify the concatenative calculus, we can make this question easier to answer.

We describe our modifications in Figure 7. First we create a new kind of quotation: subscripted quotations. Their subscript indicates how many arguments they ask for. For instance, $\texttt{[swap]}_2$ indicates that swap expects two arguments and $\texttt{[}\ x\ \texttt{swap]}_1$ indicates that x swap needs only one more argument. The second modification is a new instruction: dynamic application, written $\star$. It consults the counter to choose at run-time whether to behave as call or cons. If $n = 1$, then we are applying the last value and $\star$ acts as a call. If $n \geq 2$, then the quotation expects more arguments to come; $\star$ acts as a cons and decrements the counter.

In Figure 8, we use $\star$ to build a call-by-value conversion algorithm. Subscripted quotations correspond to terms in the original combinatory expression. Unsubscripted quotations appear only in the inner code of basic combinators, always together with dip. The conversion rules always compile applications into $\star$ and do not directly output any cons nor call. Unlike the call-by-name conversion, this one wraps basic combinators in a quotation.

We use exactly one $\star$ per application, including the ones that might be “invisible”. This becomes clearer if we write parenthesis around every application. For example, $B\,x\,y\,z$ reduces to $(x\ (y\,z))$. Looking back at Figure 8, the inner parenthesis around $y\,z$ is the $\star$ inside the dip’s quotation and the outer parenthesis is the $\star$ after the dip. K and I don’t have any $\star$, because they don’t introduce any new applications. C and W produce two applications in sequence. See $C I x I \longrightarrow ((II)\,x) \longrightarrow (Ix) \longrightarrow x$.

$$
\begin{aligned}
&\langle BKIxy \rangle =\\
&\quad y\ x\ \texttt{[]}_1\ \texttt{[[zap] dip]}_2\ \texttt{[[}\star\texttt{] dip}\,\star\ \texttt{]}_3\ \star\ \star\ \star\ \star\\
&\quad y\ x\ \texttt{[]}_1\ \texttt{[[zap] dip]}_2\ \texttt{[}\star\texttt{] dip}\ \star\ \star\\
&\langle K(Ix)y \rangle =\\
&\quad y\ x\ \texttt{[]}_1\ \star\ \texttt{[[zap] dip]}_2\ \star\ \star\\
&\quad y\ x\ \varepsilon\ \texttt{[[zap] dip]}_2\ \star\ \star\\
&\langle Kxy \rangle =\\
&\quad y\ x\ \texttt{[[zap] dip]}_2\ \star\ \star\\
&\quad y\ x\ \texttt{[zap] dip}\\
&\langle x \rangle =\\
&\quad x
\end{aligned}
$$

Figure 9. Dynamic compilation of $B K I x\,y$

Let’s now illustrate the behavior of $\star$ with a concrete example. In Figure 9, we translate the combinatory expression $B K I x\,y$. The subsequent steps closely mirror the call-by-value reduction $B K I x\,y \longrightarrow K\,(I x)\,y \longrightarrow K x\,y \longrightarrow x$. The first and second steps consume three $\star$ to unquote the B and then run it. The third and fourth steps consume one $\star$ to unquote and run the I. Finally, the last two steps unquote and run the K.

The conversion function $\langle \alpha \rangle$ has one downside: partially applied combinators such as $K I$ compile into concatenative programs that are not values:

$$\langle KI \rangle = \texttt{[]}_1\ \texttt{[[zap] dip]}_2\ \star \mapsto \texttt{[[]}_1\ \texttt{[zap] dip]}_1$$

To fully emulate a call-by-value reduction order, we must add a simplification step that removes the $\star$ from values.

Definition 4.1. Let the relation $\alpha \Rightarrow \texttt{P}$ mean that $\langle \alpha \rangle = \texttt{Q}$ and $\texttt{Q} \mapsto^* \texttt{P}$ using only reduction rules that eliminate $\star$.

Lemma 4.2. *If α is an irreducible combinator expression then there exists an irreducible concatenative value v such that $\alpha \Rightarrow v$.*

Proof. If α is a combinatory value, the only possible concatenative reduction steps taken starting from $\langle \alpha \rangle$ are simplifications where $\star$ performs a cons. The result will be a single subscripted quotation, which is a value. □

We are now ready to prove that our translation emulates a call-by-value evaluation order. Our construction guarantees that we only reduce a combinator after its arguments have been reduced to values, because $\star$ requires that the argument be a value. Firstly, we point out that the concatenative program only reduces a combinator after its arguments have been reduced to values. The other thing we must show is that the translated program can follow the entire combinator reduction sequence, without getting stuck.

Theorem 4.3 (call-by-value simulation)**.** *If $\alpha \longrightarrow \beta$ then there exist concatenative programs* P *and* Q *such that $\alpha \Rightarrow \texttt{P}$ and $\beta \Rightarrow \texttt{Q}$ and $\texttt{P} \mapsto^* \texttt{Q}$.*

Proof. By induction in the small-step reduction relation $\alpha \longrightarrow \beta$. The base cases are the reduction steps for basic combinators. Let's begin with the $B\,x\,y\,z \longrightarrow (x\,(y\,z))$ case.

$$\begin{array}{lcl} B\;x\;y\;z & \Rightarrow & \dot{z}\;\dot{y}\;\dot{x}\;[\star]\;\texttt{dip}\;\star \\ (x\;(y\;z)) & \Rightarrow & \dot{z}\;\dot{y}\;\star\;\dot{x}\;\star \end{array}$$

We know that $\langle B\,x\,y\,z\rangle = \langle z\rangle\,\langle y\rangle\,\langle x\rangle\;[\;[\star]\;\texttt{dip}\;\star\;]_3\;\star\star\star$. As we are using call-by-value, x, y, and z must be values. Let $\dot{x}$, $\dot{y}$, and $\dot{z}$ be the corresponding concatenative values from Lemma 4.2. We reduce $\langle B\,x\,y\,z\rangle$ to $\dot{z}\,\dot{y}\,\dot{x}\;[\;[\star]\;\texttt{dip}\;\star\;]_3\;\star\star\star$ and then to $\dot{z}\,\dot{y}\,\dot{x}\;[\star]\;\texttt{dip}\;\star$. Now we are done simplifying $\star$ and can evaluate the `dip` that came from the B. We arrive at $\dot{z}\,\dot{y}\,\star\,\dot{x}\,\star$, which matches $(x\,(y\,z))$. The proof of the other basic combinators follows a similar structure:

$$\begin{array}{lcl} C\;x\;y\;z & \Rightarrow & \dot{z}\;\dot{y}\;\dot{x}\;[\texttt{swap}]\;\texttt{dip}\;\star\;\star \\ ((x\,z)\;y) & \Rightarrow & \dot{y}\;\dot{z}\;\dot{x}\;\star\;\star \\[6pt] W\;x\;y & \Rightarrow & \dot{y}\;\dot{x}\;[\texttt{dup}]\;\texttt{dip}\;\star\;\star \\ ((x\,y)\;y) & \Rightarrow & \dot{y}\;\dot{y}\;\dot{x}\;\star\;\star \\[6pt] K\;x\;y & \Rightarrow & \dot{y}\;\dot{x}\;[\texttt{zap}]\;\texttt{dip} \\ x & \Rightarrow & \dot{x} \\[6pt] I\;x & \Rightarrow & \dot{x} \\ x & \Rightarrow & \dot{x} \end{array}$$

Lastly, the inductive case covers a reduction inside a sub-term of an application. As our call-by-value reduction does not mandate a left-to-right or right-to-left order, we can have either $(\alpha\,\gamma) \longrightarrow (\beta\,\gamma)$ or $(\gamma\,\alpha) \longrightarrow (\gamma\,\beta)$. Let $\alpha \Rightarrow \mathsf{P}$, $\beta \Rightarrow \mathsf{Q}$, and $\gamma \Rightarrow \mathsf{R}$. Applying the induction hypothesis to $\alpha \longrightarrow \beta$ gives us $\mathsf{P} \mapsto^{*} \mathsf{Q}$, which gets us to the intended destination.

$$\begin{array}{lcl@{\qquad\qquad}lcl} \alpha\;\gamma & \Rightarrow & \mathsf{P}\;\mathsf{R}\;\star & \gamma\;\alpha & \Rightarrow & \mathsf{R}\;\mathsf{P}\;\star \\ \beta\;\gamma & \Rightarrow & \mathsf{Q}\;\mathsf{R}\;\star & \gamma\;\beta & \Rightarrow & \mathsf{R}\;\mathsf{Q}\;\star \end{array}$$

□

4.3 Static call-by-value

The dynamic translation added subscripted quotations. To avoid doing that, we need to be able to infer for each $\star$, whether it will turn into a `cons` or a `call`.

That is possible if we restrict ourselves to simply-typed combinators. We describe such a method in Figure 10. The relation $\alpha : \tau \Rightarrow \mathsf{P}$ means that the combinator α has type τ and compiles to P. Our typing rules are the usual ones, except that we label the arrow types with either `call` or `cons`. For each combinator, the rightmost arrow is labelled with a red `call`, which indicates that we reached the last argument and we may now reduce the combinator. The other arrows are labelled with a blue `cons` to indicate that there are more arguments to come. The purple arrows inside argument types are labeled with the variables x and y. They may be either `call` or `cons`, as determined by the type of that argument. The compiled program uses these variables to decide whether to use `call` or `cons` for that sub-expression. For example, consider the type of the B combinator. If the y label says `call`, then we generate a `[call] dip`.

$$B : (b \xrightarrow{x} c) \xrightarrow{\texttt{cons}} (b \xrightarrow{y} a) \xrightarrow{\texttt{cons}} a \xrightarrow{\texttt{call}} c \Rightarrow [\;[\texttt{y}]\;\texttt{dip}\;\texttt{x}\;]$$

$$C : (a \xrightarrow{x} b \xrightarrow{y} c) \xrightarrow{\texttt{cons}} b \xrightarrow{\texttt{cons}} a \xrightarrow{\texttt{call}} c \Rightarrow [\;[\texttt{swap}]\;\texttt{dip}\;\texttt{x}\;\texttt{y}\;]$$

$$W : (a \xrightarrow{x} a \xrightarrow{y} b) \xrightarrow{\texttt{cons}} a \xrightarrow{\texttt{call}} b \Rightarrow [\;[\texttt{dup}]\;\texttt{dip}\;\texttt{x}\;\texttt{y}\;]$$

$$K : a \xrightarrow{\texttt{cons}} b \xrightarrow{\texttt{call}} a \Rightarrow [\;[\texttt{zap}]\;\texttt{dip}\;] \qquad I : a \xrightarrow{\texttt{call}} a \Rightarrow [\,]$$

$$\frac{\alpha : a \xrightarrow{x} b \Rightarrow \mathsf{P} \qquad \beta : a \Rightarrow \mathsf{Q}}{\alpha\;\beta : b \Rightarrow \mathsf{Q}\;\mathsf{P}\;\texttt{x}}$$

Figure 10. Inferring cons and call

In Figure 11, we show a concrete example using the combinator BKI, the same one from Figure 9. For brevity, we only show the compiled program on the last line. We color and number the arrows to show where each `call` and `cons` label comes from. The type inference begins from the basic combinators. These arrows are red and blue and have an individual number from 1 to 6. All the remaining `call` and `cons` labels are inferred from these. For instance, in the first line of the derivation, the first argument of B has type $\alpha \rightarrow \beta$. This comes from the type of K so we infer $a \overset{\text{cons 2}}{\longrightarrow} (b \overset{\text{call 6}}{\longrightarrow} a)$. Now, let's turn our attention back to the generated concatenative program. Remember that the purple arrows from Figure 10 dictate whether the resulting program uses `cons` or `call`. We represent them here with bold labels and parenthesis around the numbers. For example, the `call`(1) and `cons`(2) in the type of B dictate that we should compile it to `[ [call] dip cons ]`. The compiled program has exactly the same shape from the dynamic version from Figure 9 except that each one of the six $\star$ became a `cons` or a `call`.

An optimization. Our algorithm only generates a `call` when there are enough arguments for the quotation. Therefore, any occurrences of `cons call` are redundant and can be optimized to just `call`.

Limitations of simple types. One important limitation of simply-typed combinators is that they are strongly normalizing and always reduce to a normal form after a finite number of steps. Thus, this type system cannot support looping combinators such as $\Omega = WI(WI)$ or $Yf = f\,(Yf)$.

Another limitation is that a simple type system is not polymorphic for cons-call. The combinators must decide at compile-time whether they will use `call` or `cons`, exclusively. Such polymorphism may be necessary if a combinator is duplicated. For example, consider $WIBKxIy$. In the first reduction step, W duplicates B. The first copy wants to use `cons` to partially apply (Kx), while the second wants to use

$$\dfrac{\dfrac{\dfrac{\dfrac{B : (a \xrightarrow{\texttt{cons}\mathbf{(2)}} (b \xrightarrow{\texttt{call}\,6} a)) \xrightarrow{\texttt{cons}\mathbf{(3)}} (a \xrightarrow{\texttt{call}\mathbf{(1)}} a) \xrightarrow{\texttt{cons}\,4} a \xrightarrow{\texttt{call}\,5} (b \xrightarrow{\texttt{call}\,6} a) \qquad K : a \xrightarrow{\texttt{cons}\,2} b \xrightarrow{\texttt{call}\,6} a}{B\,K : (a \xrightarrow{\texttt{call}\,1} a) \xrightarrow{\texttt{cons}\mathbf{(4)}} a \xrightarrow{\texttt{call}\,5} (b \xrightarrow{\texttt{call}\,6} a)} \qquad I : a \xrightarrow{\texttt{call}\,1} a}{B\,K\,I : a \xrightarrow{\texttt{call}\mathbf{(5)}} (b \xrightarrow{\texttt{call}\,6} a)} \qquad x : a}{B\,K\,I\,x : b \xrightarrow{\texttt{call}\mathbf{(6)}} a} \qquad y : b}{B\,K\,I\,x\,y : a \Rightarrow y\ x\ \texttt{[] [[zap] dip] [[call}_{(1)}\texttt{] dip cons}_{(2)}\texttt{] cons}_{(3)}\texttt{ cons}_{(4)}\texttt{ call}_{(5)}\texttt{ call}_{(6)}}$$

Figure 11. Inferring `cons` and `call` for $B\,K\,I\,x\,y$

`call` to apply $(I y)$. The original B cannot satisfy both.

$$\begin{array}{l} W\,I\,B\,K\,x\,I\,y \\ I\,B\,B\,K\,x\,I\,y \\ B\,B\,K\,x\,I\,y \\ B\,(K\,x)\,I\,y \\ (K\,x)\;(I\,y) \end{array}$$

5 Conclusion

We connected combinatory logic and concatenative calculus, two styles of tacit programming. We showed how to convert between them using semantics-preserving simulations.

The translation from combinatory to concatenative uses continuation-passing style. Our main contribuition was a model that can handle call-by-value evaluation order. The key obstacle was the impedance mismatch between curried combinators and n-ary concatenative programs.

Our translation from concatenative to combinatory uses direct style, without continuations. It relates regular combinators to short concatenative programs. We also extended the notion of regular combinators to encompass embedded constants and higher-order programming.

Related Work. Another formalization of the concatenative calculus was developed by Kleffner [4]. His typed concatenative calculus features lambdas, recursive fixpoint operators and polymorphic type inference.

Our compilation from combinatory to concatenative is reminiscent of lambda calculus stack machines such as the SECD machine [5]. The main difference is that the concatenative calculus must distinguish between partial and non-partial application.

Simulation relations are a standard technique for showing that compilation preserves the semantics of the source languages. An example is the CompCert verified compiler [6].

Future Work. In this paper, we used the *BCKWI* basis. It would be interesting to do the same for *SKI*, but for that we would first need to define regular combinators for that basis.

A limitation of our work is that our proofs were done in pen-and-paper. In the future, we plan to mechanize them using a proof assistant.

Acknowledgments

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001

References

[1] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. 1958. *Combinatory logic*. Vol. 1. North-Holland Amsterdam. (Regular combinators are discussed in chapter 5).

[2] Daniel Kiyoshi Hashimoto Vouzella de Andrade. 2024. *From Combinatory to Concatenative and Back Again*. Bachelor's Thesis. Universidade Federal do Rio de Janeiro. http://hdl.handle.net/11422/22871.

[3] Brent Kerby. 2002. The Theory of Concatenative Combinators. Published online at http://tunes.org/~iepos/joy.html. Accessed 05/09/2023.

[4] Robert Kleffner. 2017. *A Foundation for Typed Concatenative Languages*. Master's thesis. Northeastern University.

[5] Peter J Landin. 1964. The mechanical evaluation of expressions. *The computer journal* 6, 4 (1964), 208–230. https://doi.org/10.1093/comjnl/6.4.308

[6] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[7] Charles H Moore and Geoffrey C Leach. 1970. Forth–a language for interactive computing. *Amsterdam: Mohasco Industries Inc* (1970).

[8] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. 2010. Factor: A dynamic stack-based programming language. *ACM SIGPLAN Notices* 45, 12 (2010), 43–58.

[9] Adolfo Piperno. 1989. Abstraction problems in combinatory logic: a compositive approach. *Theoretical computer science* 66, 1 (1989), 27–43.

[10] Elizabeth D Rather, Donald R Colburn, and Charles H Moore. 1996. The evolution of Forth. In *History of programming languages—II*. 625–670.

[11] Moses Schönfinkel. 1924. Über die Bausteine der mathematischen Logik. *Mathematische annalen* 92, 3-4 (1924), 305–316.

[12] Manfred von Thun. 1994. Mathematical foundations of Joy. Published online at http://www.latrobe.edu.au/phimvt/joy/j02maf.html. Archived in 2011 at https://web.archive.org/web/20111007025556/http://www.latrobe.edu.au/phimvt/joy/j02maf.html.

[13] Manfred von Thun and Reuben Thomas. 2001. Joy: Forth's Functional Cousin. In *Proceedings of the 17th EuroForth Conference*.