UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DANIEL KIYOSHI HASHIMOTO VOUZELLA DE ANDRADE

From Combinators to Concatenative and Back Again

RIO DE JANEIRO
2024

DANIEL KIYOSHI HASHIMOTO VOUZELLA DE ANDRADE

From Combinators to Concatenative and Back Again

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Hugo Musso Gualandi

RIO DE JANEIRO

2024

CIP - Catalogação na Publicação

H348f

Hashimoto Vouzella de Andrade, Daniel Kiyoshi
From Combinators to Concatenative and Back Again (Indo e Voltando de Combinadores para Concatenativo) / Daniel Kiyoshi Hashimoto Vouzella de Andrade. -- Rio de Janeiro, 2024.
39 f.

Orientador: Hugo Musso Gualandi.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2024.

1. Cálculo Concatenativo. 2. Lógica Combinatória. 3. Programação Tácita. 4. Programação Point-free. 5. Linguagens de Pilha. I. Musso Gualandi, Hugo, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

DANIEL KIYOSHI HASHIMOTO VOUZELLA DE ANDRADE

From Combinators to Concatenative and Back Again

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 06 de Fevereiro de 2024

BANCA EXAMINADORA:

Prof. Hugo Musso Gualandi
Instituto de Computação – UFRJ

Prof. Daniel Chicayban Bastos
Instituto de Computação – UFRJ

Prof. Hugo de Holanda Cunha Nobrega
Instituto de Computação – UFRJ

Prof. João Antonio Recio Paixão
Instituto de Computação – UFRJ

"Any sufficiently advanced technology
is indistinguishable from magic."

Arthur C. Clarke

RESUMO

A programação tácita ou *point-free* é um estilo de programação que evita nomear variáveis, através do uso de combinadores para compor funções menores. Dois modelos de programação tácita são o cálculo concatenativo e a lógica combinatória. O cálculo concatenativo está relacionado às linguagens de programação baseadas em pilha, estas usadas em diversos contextos, incluindo *bytecode* para máquinas virtuais e sistemas embarcados. A lógica combinatória é um modelo computacional aplicativo mais semelhante ao cálculo lambda. Ambos estes modelos tácitos são frequentemente comparados entre si. Os combinadores elementares da lógica combinatória são comumente casados com instruções de manipulação de pilha: por exemplo, C se parece com o $swap$, e W se parece com o dup. Neste trabalho, nós generalizamos esta conexão para todos os combinadores e todos os programas de pilha. Nós descrevemos três algorítmos que convertem de um modelo tácito para o outro. O primeiro algorítmo traduz qualquer combinador da lógica combinatória para uma expressão do cálculo concatenativo usando ou a estratégia de redução *call-by-name* ou a *call-by-value*. O segundo algorítmo é especializado para um subconjunto de combinadores que recebe uma função *callback* como seu primeiro argumento. Ele produz programas de pilha menores e mais intuitivos. Por último, o terceiro algorítmo é uma versão reversa do segundo, e converte qualquer programa de pilha de volta para uma expressão da lógica combinatória. Como nossos algorítmos preservam a ordem da redução, nós mostramos que cada modelo é capaz de simular o outro.

Palavras-chave: linguagens de programação; programação tácita; programação point-free; cálculo concatenativo; linguagens de pilha; lógica combinatoria.

ABSTRACT

Tacit or *point-free* programming is a programming style which avoids named variables, by using combinators to compose smaller functions. Two models of tacit programming are concatenative calculus and combinatory logic. The concatenative calculus is related to stack-based languages, who are used in various contexts, including bytecode for virtual machines and embedded systems. Combinatory logic is an applicative computation model more similar to the lambda calculus. These two tacit models are frequently compared to each other. The elementary combinators of combinatory logic are often paired to stack-manipulation instructions: for instance, C looks similar to $swap$, and W is similar to dup. In this work, we generalize this connection to all combinator expressions and stack programs. We describe three algorithms to convert between the two tacit models. The first translates any combinatory logic expression into a concatenative expression using either the call-by-name or the call-by-value evaluation order. The second algorithm is specialized to a subset of combinators that receives a callback function as the first argument. It produces shorter and more intuitive stack programs. Lastly, the third algorithm is a backwards version of the second one, converting any concatenative calculus program back to a combinatory logic expression. Because our algorithms preserve the reduction order, we show that each model is able to simulate the other.

Keywords: programming languages; tacit programming; pointfree programming; concatenative calculus; stack languages; combinatory logic.

SUMÁRIO

1 INTRODUCTION

Tacit programming, also known as *point-free* programming, is a programming style that avoids naming variables and intermediary values. It focuses on combining smaller functions in different ways to construct bigger functions. One could think about it as building functions using pipelines and similar techniques. This style may be used to make code easier to read or to make the data flow more explicit.

The simplest tool for tacit programming is function composition. For instance, instead of $h(x) = f(g(x))$, we could say $h = f \circ g$ without mentioning x. Function composition appears in many languages and contexts. One of the most common examples is Unix shell, which permits piping one program's output into another program's input without naming a variable or a temporary file (Listing 1).

Haskell is another interesting language example for tacit programming, because its standard library includes several higher-order combinator functions, such as `flip`, `on` and function composition. For instance, in Listing 2 we use these combinators to define a function `isSameLength` that tests whether the lengths of two strings are the same, without naming any variables.

APL (IVERSON, 1962) and other Iversonian array languages, such as J and BQN, heavily encourage tacit programming. They feature several operators designed for tacit programming. A customary example is the function for computing the average of a list. In J, this averaging function is written `+/ % #`. This is close to the intuitive english definition: "the sum (`+/`) divided (`%`) by the length (`#`) of the list (the unnamed argument)".

Another tacit language group are the stack-based languages. These languages use reverse polish notation: the arguments come before functions. A stack-shuffling combinator, such as `swap` and `dup`, can be inserted between the arguments and the function to reorder or change the arguments. For instance, `10 10 +` could be written as `10 dup +`,

```
ls | grep '.pdf$' | sed 's/\.pdf$//' | less
```

Listing 1 – A shell pipeline example that lists PDF filenames

```
ghci> isSameLength = isEqual `on` length
ghci> -- Alternative definition:
ghci> -- isSameLength xs ys = isEqual (length xs) (length ys)
ghci> isSameLength "hello" "world"
True
ghci> isSameLength "tacit" "programming"
False
```

Listing 2 – Higher order combinators in Haskell

```
flip f x y          # Haskell
x f passive y       # J
y x swap f          # Stack languages
```

Listing 3 – Three different ways of building $f\ y\ x$

which can be tacitly factored into `dup +`.

Stack-based computing is common in lower level languages, often appearing as bytecode for virtual machines, such as Java's JVM, Erlang's BEAM, and Web Assembly[1]. There are also stack-based languages designed for humans. Some of the earliest are FORTH (MOORE; LEACH, 1970; RATHER; COLBURN; MOORE, 1996) and Postscript. Joy introduces a higher order aspect to stack-based programming, with the notion of quotations (THUN, 1994; THUN; THOMAS, 2001). Factor is one of the many languages inspired by Joy and introduces dynamic typing (a great improvement from mere arity checking), object orientation and other features (PESTOV; EHRENBERG; GROFF, 2010). Cat and Kitten introduce static type checking and type inference. The concatenative programming community created a wiki to catalog these and other concatenative languages[2]. The term *concatenative* was coined recently, appearing around Joy's creation time.

1.1 THE PROBLEM

We are interested in the different manifestations of tacit programming. In particular, we want to relate the applicative higher-order approach (used by Haskell and J) to the concatenative approach (of stack languages). In Listing 3, we illustrate this by comparing how these languages swap arguments of a function. We want to be able to say that `flip` and `passive` are analogous to `swap`, and also we would like to extend this analogy to bigger programs with more combinators.

To study this connection, we will need formal models for the applicative and concatenative setting. To model stack languages we will employ the concatenative calculus of Thun and Kerby (THUN, 1994; KERBY, 2002). This theory will be explained in Chapter 3.

The main model for the applicative setting is combinatory logic. It was started in the beginning of the twentieth century by Schönfinkel, who highlighted the advantages of eliminating variables from logic (SCHÖNFINKEL, 1924). Inspired by that, Curry and others further developed combinatory logic into a formal logic system without variables (CURRY et al., 1958). Curry's tradition of combinatory logic features the higher-order functions B, C, K, S, W, I, which perform elementary operations, such as composition, reordering, etc. We will cover combinatory logic in Chapter 2.

[1] and my personal favorite: uxn/varvara with tal language
[2] https://concatenative.org/

The analogy of swapping arguments in three different notations, previously discussed in Listing 3, leads to some questions. The combinator C is a function similar to Haskell's `flip`, which is in turn related to `swap`, but what about the others? As we will see later, K is analogous to `zap` and W to `dup`. But what about B and I? What about non-elementary combinators, such as S, $BC(BC)$ and $B(BS)B$? Can we properly describe how they relate to stack combinators? More concretely, can we look at a combinatory logic program and understand it as a concatenative calculus program, and vice-versa?

To address those questions, we will produce algorithms to convert from one theory to the other and vice-versa. For example, in combinatory logic, the combinator BCW maps $f\ x\ y$ to $f\ y\ y\ x$. As we will see in Chapter 2, the evaluation of $BCW\ f\ x\ y$ reduces to $W\ f\ y\ x$ and finally to $f\ y\ y\ x$. First, x and y swapped places, then y was duplicated. This behaves similarly to $swap\ dup$ from stack languages.

But we are not merely interested in the final result. For instance, the combinator BCC is a weird identity function. It maps $f\ x\ y$ to $f\ x\ y$ by swapping x and y twice, back to where they started. Therefore, we may not convert it to the no-op identity function. We would prefer $swap\ swap$, thus preserving the reductions steps. In technical terms, we want the conversion to map every input into an output with the following property: each reduction step in the input side have a matching step in the output side. If the input-output pair has this property, we say that the output is a **simulation** of the input. Observe that the number of reduction steps for the input may be smaller than the number of steps in the output.

1.2 SUMMARY

The remainder of this work is organized as follows: in Chapter 2, we review the lambda calculus and combinatory logic. We show the notation we will use and an algorithm to translate from lambda calculus to combinatory logic. In Chapter 3, we describe the concatenative calculus. In Chapter 4, we present our first conversion from combinatory logic to concatenative calculus. It accepts any combinatory logic expression, but its output is hard to read. In this chapter we also discuss the call-insertion problem. In Chapter 5, we present the second algorithm, specialized to convert regular combinators to concatenative calculus. Its output is more idiomatic than the general algorithm. In Chapter 6, we go in the opposite direction, converting from concatenative calculus back to combinatory logic. We also extend the algorithm to work with improper combinators (proper combinators are defined in Section 2.2). In Chapter 7, we discuss our final conclusions and ideas for future work.

2 LAMBDAS AND COMBINATORS

In this chapter, we will briefly revisit lambda calculus and combinatory logic and take the opportunity to establish a notation we will use for them. The concatenative calculus will be introduced more slowly and with more details in the next chapter.

2.1 LAMBDA CALCULUS

The lambda calculus is a minimalist logic and computation system, based on function definition and function calls. A lambda calculus's expression is defined by the grammar in Figure 1. Variables are denoted by lowercase roman letters (a, b, c, ...), represented in the grammar by v. Function applications are denoted by juxtaposing two expressions. Lambda functions have a variable (its argument) and an expression (its body). An expression is either a variable, a function application, or a lambda function and parenthesis may be used to disambiguate an expression. Expressions are denoted by uppercase roman letters (N, P, Q, ...).

On top of that, it is usual to add three rules to ease reading: application associates to the left; lambdas extend as far to the right as possible; a lambda with many variables means many lambdas with one variable. We show those rules in Figure 2.

Computation is achieved by β-reducing expressions. A **reducible expression** (redex) is an application where the left sub-expression is a lambda. The **β-reduction** reduces a redex to the body of the lambda on the left with its argument substituted by the sub-expression on the right. One expression, might contain more than one redex, in which case one must choose which to reduce first. In Figure 3, we show two possible reduction sequences from the same starting expression, each line is separated by a β-reduction.

There are many strategies for choosing which redex to β-reduce first (or if we want to reduce a redex or stop), each having different advantages and disadvantages. **Call-by-name** evaluation order always reduces the first outermost redex. It was shown on

$$
\begin{array}{rcll}
E & := & & \textsc{Expressions} \\
 & & v & \textit{variables} \\
 & | & E\ E & \textit{application} \\
 & | & \lambda\ v\ .\ E & \textit{lambda}
\end{array}
$$

Figure 1 – Lambda calculus

$$
\begin{aligned}
x\ y\ z &= (x\ y)\ z \\
\lambda\ x\ .\ x\ y &= \lambda\ x\ .\ (x\ y) \\
\lambda\ x\ y\ .\ x &= \lambda\ x\ .\ \lambda\ y\ .\ x
\end{aligned}
$$

Figure 2 – Abbreviations

$(\lambda\ x\ y\ .\ x)\ a\ ((\lambda z\ .\ c)\ b)$
$(\lambda\ y\ .\ a)\ ((\lambda z\ .\ c)\ b)$
a

(a) Call-by-name

$(\lambda\ x\ y\ .\ x)\ a\ ((\lambda z\ .\ c)\ b)$
$(\lambda\ y\ .\ a)\ ((\lambda z\ .\ c)\ b)$
$(\lambda\ y\ .\ a)\ c$
a

(b) Call-by-value

Figure 3 – Reduction strategies

$(\lambda\ x\ .\ (\lambda\ y\ .\ y\ x))\ y\ z$
$(\lambda\ y\ .\ y\ y)\ z \qquad (*)$
$z\ z$

(a) Bad evaluation

$(\lambda\ x\ .\ (\lambda\ y\ .\ y\ x))\ y\ z$
$(\lambda\ x\ .\ (\lambda\ a\ .\ a\ x))\ y\ z \qquad (\star)$
$(\lambda\ a\ .\ a\ y)\ z$
$z\ y$

(b) Good evaluation

Figure 4 – α-renaming

the left (Figure 3a). **Call-by-value** evaluation order always fully reduces the argument before applying it to a lambda. It was shown on the right (Figure 3b).

Sometimes, an expression cannot be reduced because name conflicts might change the meaning of the expression. In such cases, we can α-rename a lambda to continue to reduce. The **α-rename** changes the names of variables without affecting its meaning. We show an example of this in Figure 4. On the left (Figure 4a), there is an illegal reduction, marked with a $(*)$. This reduction is illegal because it changes the meaning of the argument y: before, it was a free variable; after, it turned into a variable bound by a inner lambda. On the right (Figure 4b), there is a correct reduction, α-renaming was applied on the line marked with a $(\star)$. These variable scoping and renaming complications motivate combinatory logic, which avoids the usage of variables.

Lastly, **η-reduction** is a simplification done to a lambda that receives an argument and only applies that argument to a function. For example, $(\lambda\ x\ .\ N\ x)$ can be η-reduced to N. For those who want to learn more about lambda calculus, a good reference is Barendregt (BARENDREGT et al., 1984).

2.2 COMBINATORY LOGIC

The combinatory logic is another minimalist model of logic and computation. It was created to not require variables and therefore avoid issues with variable scoping and substitution. Compared to lambda calculus, it does not have lambdas. It defines functions by combining and composing smaller functions.

A **pure combination** is a mathematical function that receives some variables and returns an expression that combines them using application. Variables may appear more than once or be ignored. The functions in Figure 5 are pure combinations. Everything

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
B\ f\ g\ x & := & f\ (g\ x) & S\ f\ g\ x & := & f\ x\ (g\ x) \\
C\ f\ x\ y & := & f\ y\ x & W\ f\ x & := & f\ x\ x \\
K\ x\ y & := & x & I\ x & := & x
\end{array}
$$

Figure 5 – Pure combinations

$$
\begin{array}{rcll}
E & := & & \textsc{Expressions} \\
 & & B \mid C \mid K \mid S \mid W \mid I & \textit{base combinator} \\
 & \mid & E\ E & \textit{application}
\end{array}
$$

Figure 6 – Combinators

that appears to the right of the $:=$ are variables that appear on the left side of the equation.

A **combinator** is an implementations of a combination. Syntactically, a combinator expression can be a base combinator or an application of two combinators (Figure 6). Combinators are traditionally written in uppercase letters. Function applications are denoted by juxtaposing two expressions, the same way as lambda calculus.

Similarly to lambda calculus, we can reduce expressions. Below, we apply the definition of B, K and W, in this order.

$$B\ K\ W\ x\ y\ z\ \Rightarrow\ K\ (W\ x)\ y\ z\ \Rightarrow\ W\ x\ z\ \Rightarrow\ x\ z\ z$$

A **proper combinator** implements a pure combination. The result only contains variables, not other combinators. All of the combinators shown in Figure 5 are proper. An example of a **improper** combinator is TK, with $T\ x\ y := y\ x$. The problem is that it leaves a K in its result.

$$T\ K\ x\ \Rightarrow\ x\ K$$

Another improper combinator is Y, the famous fixpoint combinator.

$$Y\ f\ \Rightarrow\ f\ (Y\ f)$$

There are many versions of combinatory logic, depending on the chosen set of base combinators. We could have chosen other bases instead of *BCKSWI*, such as *SKI* or *BCKW*. *BCKSWI* is a **complete base**, which means that all of the other combinators can be constructed from the base. For example, $T = CI$ and $Y = B(WI)(BWB)$. One of the smallest complete bases is *SK*. For instance, $B = S(KS)K$ and $I = SKK$.

This work less concerned with a small base. We will use *BCKWI* because each combinator does only one **elementary effect**: B composes and groups with parenthesis; C reorders; K discards; W duplicates; I identifies. That's opposed to S which is capable of composing, reordering, duplicating and identifying (with K's help). The importance of these effects are discussed by Curry (CURRY et al., 1958). Another motivation for preferring *BCKWI* is our goal of relating combinatory logic to stack operations such as $swap$, zap, dup.

$$
\begin{aligned}
[\, v \,] &:= v \\
[\, P\ Q \,] &:= [\, P \,]\,[\, Q \,] \\
[\, \lambda\, x\ .\ v \,] &:= \begin{cases} K\ v & \text{, if } x \neq v \\ I & \text{, if } x = v \end{cases} \\
[\, \lambda\, x\ .\ P\ x \,] &:= \begin{cases} [\, P \,] & \text{, if } x \notin \mathrm{FV}(P) \\ W\ [\, \lambda\, x\ .\ P \,] & \text{, if } x \in \mathrm{FV}(P) \end{cases} \\
[\, \lambda\, x\ .\ P\ Q \,] &:= \begin{cases} K\ [\, P\ Q \,] & \text{, if } x \notin \mathrm{FV}(P) \wedge x \notin \mathrm{FV}(Q) \\ B\ [\, P \,]\ [\, \lambda\, x\ .\ Q \,] & \text{, if } x \notin \mathrm{FV}(P) \wedge x \in \mathrm{FV}(Q) \\ C\ [\, \lambda\, x\ .\ P \,]\ [\, Q \,] & \text{, if } x \in \mathrm{FV}(P) \wedge x \notin \mathrm{FV}(Q) \\ S\ [\, \lambda\, x\ .\ P \,]\ [\, \lambda\, x\ .\ Q \,] & \text{, if } x \in \mathrm{FV}(P) \wedge x \in \mathrm{FV}(Q) \end{cases} \\
[\, \lambda\, x\ .\ \lambda\, y\ .\ P \,] &:= \begin{cases} [\, \lambda\, x\ .\ [\, \lambda\, y\ .\ P \,] \,] & \text{, if } x \neq y \\ K\ [\, \lambda\, y\ .\ P \,] & \text{, if } x = y \end{cases}
\end{aligned}
$$

Table 1 – Conversion from lambda calculus to $BCKSWI$ base

2.3 FROM LAMBDA CALCULUS TO COMBINATORS

In this section, we will show a standard conversion algorithm from lambda calculus to combinatory logic. More specifically, the algorithm will return a combinator using *BCKSWI* base. This conversion will not be used in this work, but it is included to help to create combinatory expression examples. During this research, it helped with testing and intuition building. Most of the initial test cases were first lambda expressions, then converted to combinatory expressions. The algorithm massages a lambda in such way that η-reduction can be applied.

$$
\begin{aligned}
&\lambda\, x\ .\ f\ (g\ x) \\
&\lambda\, x\ .\ B\ f\ g\ x \\
&B\ f\ g
\end{aligned}
$$

We use $[\, P \,]$ to convert the lambda calculus expression P. All the cases are represented in Table 1, where P and Q are lambda calculus expressions and v is a single variable. The relation $x \in \mathrm{FV}(P)$ means that x is a free variable of P, and therefore P uses the variable x. And $x \notin \mathrm{FV}(P)$ means the opposite: P does not use the variable x.

This conversion splits into many cases in order to make it clearer that all the possible cases are covered by the conversion algorithm. One might feel intrigued to see K appearing three times. All of those cases are the same, they could be summarized as

$$
[\, \lambda\, x\ .\ P \,] := K\ P \quad \text{, if } x \notin \mathrm{FV}(P)
$$

Using only *SKI* (another complete basis) would be sufficient for the conversion to work (CURRY et al., 1958). Recall that a complete base is a set of combinators that can create any other combinator, or in this case, any lambda function. The inclusion of extra rules for *BCW* and η-reduction reduces the size of the converted combinator, in the general case. The rule that uses a W is less common in the literature.

3 CONCATENATIVE CALCULUS

The concatenative calculus is another minimalist model of logic and computation, based on concatenation/composition of functions. Differently from the others, there is a distinction from executing functions (programs) and value functions (called quotations). Because of that, it has two syntaxes, one for instructions and another for values. The grammars in Figure 7 are a simplification of Kleffner's (KLEFFNER, 2017) grammars, replacing arbitrary lambda functions with primitive instructions and removing some constructions, such as fixpoint, name binding, numbers, and others. The simplification is made to match the subset of combinatory logic that we are working with.

With this simplification, the only values are quotations, equivalent to functions as values, represented by programs between square brackets. In the grammar, $\vec{I}$ represents a program: a possibly empty sequence of instructions. In our rules, uppercase variables (A, B, C, ...) stand for sequences of instructions. An empty sequence is represented by ε. Instructions are primitive functions, written in lowercase ($swap$, zap, dup, ...) or values/quotation (`[ ]`, `[` $swap\ dup$ `]`, ...). Similarly to combinatory logic, we could have chosen a different set of base instructions.

One way of evaluating a concatenative program is to use a stack-machine. For our notation, execution steps occur near the vertical bar (|). The next instruction to run is the first to the right of | and it will manipulate the values on the top of the stack, which are on the left, nearest to the |.

$$\cdots\ x_1\ x_0 \mid i_0\ i_1\ \cdots$$

A push instruction pushes a value onto the stack. It is represented by a quotation literal. Note that `[` A `]` may represent either a push instruction (if on the right side of |) or a quotation value (if on the left side of |).

$$\cdots\ x_0\ \mid\ \texttt{[}\ A\ \texttt{]}\quad\Rightarrow\quad\cdots\ x_0\ \texttt{[}\ A\ \texttt{]}\ \mid$$

Instructions affect the top values of the stack and leave the remaining values unchanged. So, we can write the previous rule in a simpler way, omitting the rest of the stack. The Table 2 shows the base instructions and a few others below a horizontal rule.

V	:=		VALUES
		`[` $\vec{I}$ `]`	*quotation*
I	:=		INSTRUCTION
		$swap \mid zap \mid dup \mid cons \mid sons \mid call \mid dip$	*base instruction*
	|	V	*push value*

Figure 7 – Concatenative calculus

	$[\ A\ ]$	$\Rightarrow$	$[\ A\ ]$	
$y\ x$	$swap$	$\Rightarrow$	$x\ y$	
x	dup	$\Rightarrow$	$x\ x$	
x	zap	$\Rightarrow$		
$[\ A\ ]$	$call$	$\Rightarrow$		A
$x\ [\ A\ ]$	$cons$	$\Rightarrow$	$[\ x\ A\ ]$	
$x\ [\ A\ ]$	$sons$	$\Rightarrow$	$[\ x\ A\ ]\ x$	
$x\ [\ A\ ]$	dip	$\Rightarrow$		$A\ x$
$[\ B\ ]\ [\ A\ ]$	cat	$\Rightarrow$	$[\ B\ A\ ]$	
x	$unit$	$\Rightarrow$	$[\ x\ ]$	
$x\ [\ A\ ]$	sip	$\Rightarrow$	x	$A\ x$
$x\ [\ A\ ]$	$take$	$\Rightarrow$	$[\ A\ x\ ]$	
$x\ [\ A\ ]$	$cake$	$\Rightarrow$	$[\ x\ A\ ]\ [\ A\ x\ ]$	

Table 2 – Concatenative instructions

Some of the instructions in Table 2 are elementary, in an analogous way of elementary for combinators, each only does one kind of effect on the stack. The effects are: reorder ($swap$), duplicate (dup), discard (zap), unquote ($call$), concatenate (cat), and quote ($unit$). Many of those instructions have alternative names: $drop$ and pop for zap; $apply$ for $call$; $concat$ and $compose$ for cat; and $quote$ for $unit$.

An execution starts with a (possibly empty) stack holding the inputs. The instructions are executed one by one. When there are no more instructions to execute, the (possibly empty) stack is the output of the execution. For example, let's run $zap\ unit\ dup\ cat$ with x, y and z on the stack. By the end, z is untouched; that is fine.

$$\begin{aligned} z\ y\ x &\mid zap\ unit\ dup\ cat \\ z\ y &\mid unit\ dup\ cat \\ z\ [\ y\ ] &\mid dup\ cat \\ z\ [\ y\ ]\ [\ y\ ] &\mid cat \\ z\ [\ y\ y\ ] &\mid \end{aligned}$$

In following chapters, we are going to have many repeated instructions next to each other. Because of that, we will abbreviate a instruction i repeated n times to i^n. For instance, using $call$ as the repeated instruction.

$$\begin{aligned} call^0 &:= \varepsilon \\ call^1 &:= call \\ call^2 &:= call\ call \\ call^3 &:= call\ call\ call \end{aligned}$$

3.1 CONCATENATIVE FUNCTIONS

Functions from concatenative calculus may use zero or more values from the stack and leave zero or more values after execution. Note that a push instruction is an example

of function that receives zero values. When they are called, they must receive at least all of their expected arguments. In other words, differently from lambda calculus and combinatory logic, functions are not automatically curried.

As a consequence of quotations not dequoting by themselves, we need to explicitly add $call$ instructions to run them. If the called function does not have enough arguments in the stack the computation gets stuck. For instance, $swap$ needs at least two arguments to execute. While, in the next example, dup needs only one, but none was given; it got stuck because of the lack of arguments.

Not stuck	Stuck
$y\ x \mid [\ swap\ ]\ call$	$\mid [\ dup\ ]\ call$
$y\ x\ [\ swap\ ] \mid call$	$[\ dup\ ] \mid call$
$x\ y \mid$	$\mid dup$

Quotations are not automatically curried, but currying is still possible. The instruction $cons$ does exactly that, it represents currying in the concatenative world. Another way of currying is to quote the argument x with $unit$, and cat it to the function $[\ A\ ]$.

$$
\begin{aligned}
x\ [\ A\ ] &\mid cons \\
[\ x\ A\ ] &\mid
\end{aligned}
$$

$$
\begin{aligned}
x\ [\ A\ ] &\mid swap\ unit\ swap\ cat \\
[\ A\ ]\ x &\mid unit\ swap\ cat \\
[\ A\ ]\ [\ x\ ] &\mid swap\ cat \\
[\ x\ ]\ [\ A\ ] &\mid cat \\
[\ x\ A\ ] &\mid
\end{aligned}
$$

Concatenative functions may receive more arguments then they expect. When it happens, the extra arguments are not touched. This idea explains semantically what consing extra values do. In the following example, dup takes one argument, but two values got curried/consed into it. When the resulting quotation gets called, dup will leave the second consed value y untouched.

$$
\begin{aligned}
y\ x\ [\ dup\ ] &\mid cons\ cons\ call \\
y\ [\ x\ dup\ ] &\mid cons\ call \\
[\ y\ x\ dup\ ] &\mid call \\
&\mid y\ x\ dup \\
y\ x &\mid dup \\
y\ x\ x &\mid
\end{aligned}
$$

The empty sequence of instructions (ε) is the identity function of the concatenative world. The function ε is similar to $dup\ zap$, except that the latter expects one argument on the stack, while the former expects none. In other words, when there are no values on the stack, $dup\ zap$ gets stuck; while ε still works in that case. If, for example, we cat these identity-like functions with $[\ A\ ]$ and $call$ the result, $\varepsilon\ A$ will get stuck on A (same

as if we were only running A), while $dup\ zap$ will get stuck on dup. This highlights the relation between composition and concatenation in concatenative languages: composing with identity is the same as concatenating with empty.

$$\begin{aligned} \texttt{[}\ \varepsilon\ \texttt{]}\ \texttt{[}\ A\ \texttt{]} \mid\ & cat\ call \\ \texttt{[}\ \varepsilon\ A\ \texttt{]} \mid\ & call \\ \mid\ & \varepsilon\ A \\ \mid\ & A \end{aligned} \qquad\qquad \begin{aligned} \texttt{[}\ dup\ zap\ \texttt{]}\ \texttt{[}\ A\ \texttt{]} \mid\ & cat\ call \\ \texttt{[}\ dup\ zap\ A\ \texttt{]} \mid\ & call \\ \mid\ & dup\ zap\ A \end{aligned}$$

3.2 REACHING DEEPER INTO THE STACK

The instructions shown so far can only access the top two values on the stack. Assuming that we have three values x, y and z on the stack, how can we, for example, $cat\ x$ and z? We could introduce a family of instructions rot in order to access deeper values. The instruction rot_n moves the n-th value to the top of the stack. rot_1 does not do much. rot_2 is the same as $swap$. But with rot_3 onwards, we could pull all of wanted values up, work with them, then put them back down.

$$\begin{aligned} x_1 \mid rot_1 &\Rightarrow x_1 \mid \\ x_2\ x_1 \mid rot_2 &\Rightarrow x_1\ x_2 \mid \\ x_3\ x_2\ x_1 \mid rot_3 &\Rightarrow x_2\ x_1\ x_3 \mid \\ x_n\ x_{n-1}\ \cdots\ x_1 \mid rot_n &\Rightarrow x_{n-1}\ \cdots\ x_1\ x_n \mid \end{aligned}$$

We could implement the rot instructions ourselves in terms of $swap$, $call$, cat and $unit$. One strategy is to accumulate a quotation which has the stack values with a $swap$ between them to bubble the desired value up to the top of the stack. For example, let's build rot_3. We want to put one $swap$ after x and after y. We will start by pushing a `[ ]` and we will put $x\ swap$ inside it with $swap\ unit$ `[` $swap$ `]` $cat\ swap\ cat$. The first part $swap\ unit$ `[` $swap$ `]` cat swaps x on top of the stack and replaces it with `[` $x\ swap$ `]`. The second part $swap\ cat$ swaps out accumulator back to the top and joins it with `[` $x\ swap$ `]`. We can repeat it once more to join `[` $y\ swap$ `]` to our accumulator quotation, resulting in `[` $y\ swap\ x\ swap$ `]`. Finally, we $call$ our quotation and watch z get bubbled up to the top of the stack.

$$
\begin{aligned}
z\ y\ x &\mid \texttt{[ ]}\ (swap\ unit\ \texttt{[}\ swap\ \texttt{]}\ cat\ swap\ cat)^2\ call \\
z\ y\ x\ \texttt{[ ]} &\mid (swap\ unit\ \texttt{[}\ swap\ \texttt{]}\ cat\ swap\ cat)^2\ call \\
z\ y\ \texttt{[}\ x\ swap\ \texttt{]} &\mid (swap\ unit\ \texttt{[}\ swap\ \texttt{]}\ cat\ swap\ cat)^1\ call \\
z\ \texttt{[}\ y\ swap\ x\ swap\ \texttt{]} &\mid call \\
z &\mid y\ swap\ x\ swap \\
z\ y &\mid swap\ x\ swap \\
y\ z &\mid x\ swap \\
y\ z\ x &\mid swap \\
y\ x\ z &\mid
\end{aligned}
$$

But instead, what if we could *magically* run A in the middle of the stack? This is the alternative we will be adopt. We will use the instruction dip for that. One could describe it as saving the top value (removing it from the stack), running a function, then restoring the saved value to the top. It might be helpful to think that a quotation followed by a dip is a single instruction.

$$x \mid \texttt{[}\ A\ \texttt{]}\ dip \quad \Rightarrow \quad \mid A\ x$$

Using dip makes things simpler than using the rot family of instructions or some $unit$-cat magic. Also, it will make a nice parallel to the B combinator in the upcoming chapters. We can use $swap$ and dip to implement rot_3 and rot_4. rot_2 ($swap$) was used to implement rot_3, rot_3 was used to implement rot_4 and rot_n could be easily implemented in terms of rot_{n-1}.

rot_3 implementation

$$
\begin{aligned}
z\ y\ x &\mid \texttt{[}\ swap\ \texttt{]}\ dip\ swap \\
z\ y &\mid swap\ x\ swap \\
y\ z &\mid x\ swap \\
y\ z\ x &\mid swap \\
y\ x\ z &\mid
\end{aligned}
$$

rot_4 implementation

$$
\begin{aligned}
z\ y\ x\ w &\mid \texttt{[}\ rot_3\ \texttt{]}\ dip\ swap \\
z\ y\ x &\mid rot_3\ w\ swap \\
y\ x\ z &\mid w\ swap \\
y\ x\ z\ w &\mid swap \\
y\ x\ w\ z &\mid
\end{aligned}
$$

Embracing the idea of unifying a quotation followed by a dip to a single instruction, we can chain dip instructions to save multiple values at once. This is useful to dig the stack and work directly in the middle of it. The examples dig the stack to zap the second or third value.

zap third value

$$
\begin{aligned}
z\ y\ x &\mid \texttt{[[}\ zap\ \texttt{]}\ dip\ \texttt{]}\ dip \\
z\ y &\mid \texttt{[}\ zap\ \texttt{]}\ dip\ x \\
z &\mid zap\ y\ x \\
&\mid y\ x \\
y\ x &\mid
\end{aligned}
$$

zap fourth value

$$
\begin{aligned}
z\ y\ x\ w &\mid \texttt{[[[}\ zap\ \texttt{]}\ dip\ \texttt{]}\ dip\ \texttt{]}\ dip \\
z\ y\ x &\mid \texttt{[[}\ zap\ \texttt{]}\ dip\ \texttt{]}\ dip\ w \\
&\vdots \\
y\ x &\mid w \\
y\ x\ w &\mid
\end{aligned}
$$

We abbreviate those chains of dips with dip_n, which saves n values from the stack before running the quotation. Under this definition, dip_0 and $call$ are the same, and dip_1 is plain dip.

$$\begin{array}{rcl} \mid \texttt{[}\ A\ \texttt{]}\ dip_0 & \Rightarrow & \mid A \\ x_0 \mid \texttt{[}\ A\ \texttt{]}\ dip_1 & \Rightarrow & \mid A\ x_0 \\ x_{n-1}\ \cdots\ x_0 \mid \texttt{[}\ A\ \texttt{]}\ dip_n & \Rightarrow & \mid A\ x_{n-1}\ \cdots\ x_0 \end{array}$$

The instruction sip is similar to dip, but it leaves a copy of the value on the stack for the quotation to use. Its definition is in Table 2, and it can be implemented in terms of dup and dip:

$$\begin{aligned} \texttt{[}\ A\ \texttt{]}\ sip &= dup\ \texttt{[}\ A\ \texttt{]}\ dip \\ sip &= \texttt{[}\ dup\ \texttt{]}\ dip\ dip \end{aligned}$$

The instruction $sons$, also defined in Table 2, is similar to sip and to $cons$. Imagine a $cons$ that keeps a copy of the argument on top of the stack. We did not find the instruction $sons$ in literature. We invented it to help with the conversions. $sons$ is named after its similarity with sip and $cons$ and it can be implemented in terms of them:

$$sons = \texttt{[}\ cons\ \texttt{]}\ sip$$

3.3 EQUATIONAL REASONING

There are two useful properties of concatenative calculus, namely concatenation and split. Concatenation property says that you can concatenate two functions to get a new function and it works like composition. Split property says that a function can be separated in an instruction boundary to get two functions: one from the start to this boundary and from the boundary to the end. Kleffner proves that the type system which he defined preserves those important properties (KLEFFNER, 2017).

Those properties can be used to separate some part of the function, change it for something equivalent and then stitch them back together. In the following example, the split property is used twice to isolate the $x\ \texttt{[}\ A\ \texttt{]}\ cons$ subfunction. We know what the function $x\ \texttt{[}\ A\ \texttt{]}\ cons$ does: it takes no arguments from the stack, and pushes $\texttt{[}\ x\ A\ \texttt{]}$. So, we can swap it for the function that just pushes the quotation and stitch them back up, the resulting function will do the same thing.

$$\begin{aligned} &swap\ dup\ (x\ \texttt{[}\ A\ \texttt{]}\ cons)\ dip \\ &swap\ dup\ (\texttt{[}\ x\ A\ \texttt{]})\ dip \\ &swap\ dup\ \texttt{[}\ x\ A\ \texttt{]}\ dip \end{aligned}$$

We just used an equivalence that is true by definition. But other non-obvious equivalences can be made. For instance, the following equivalences describe how $call$, dip and sip distribute over concatenation.

$$[\ A\ B\ ]\ call \Leftrightarrow [\ A\ ]\ call\ [\ B\ ]\ call$$
$$[\ A\ B\ ]\ dip \Leftrightarrow [\ A\ ]\ dip\ [\ B\ ]\ dip$$
$$[\ A\ B\ ]\ sip \Leftrightarrow [\ A\ ]\ sip\ [\ B\ ]\ dip$$

Other equivalences may need extra requirements, such as at least n extra values on the stack. In this case, these extra arguments remain in the same position by the end of the function but are needed for the execution to not get stuck. We write the extra arguments to the left of a vertical bar.

$y\ x$	$swap\ swap \Leftrightarrow \varepsilon$
x	$dup\ zap \Leftrightarrow \varepsilon$
x	$unit\ call \Leftrightarrow \varepsilon$
x	$[\ ]\ cat \Leftrightarrow \varepsilon$
x	$[\ ]\ dip \Leftrightarrow \varepsilon$
x	$f\ cons\ call \Leftrightarrow f\ call$
$x_n\ \cdots\ x_0$	${rot_n}^n \Leftrightarrow \varepsilon$

Because this is a reasoning of equivalent functions, not a reasoning of reduction rules, any equivalence can be used. In the next example, we will use the $swap\ swap \Leftrightarrow \varepsilon$ equivalence. The function $dup\ [\ A\ ]\ cons$ expects one argument and leaves two on the stack, so we are sure that we use this equivalence. We use split to isolate $swap\ swap$ and substitute it with ε. When we try to join all of the parts, we notice that ε disappears because concatenating a sequence with empty results in the sequence.

$$dup\ [\ A\ ]\ cons\ swap\ swap\ dip$$
$$dup\ [\ A\ ]\ cons\ (swap\ swap)\ dip$$
$$dup\ [\ A\ ]\ cons\ (\varepsilon)\ dip$$
$$dup\ [\ A\ ]\ cons\ dip$$

If we have that $A \Leftrightarrow B$, can we say that $[\ A\ ] \Leftrightarrow [\ B\ ]$? The answer for this question is not simple and it may vary in different flavors of concatenative worlds (maybe someone could ask how many instructions there are inside a quotation). But if the context guarantees that these quotations are only dequoted, the answer is yes. For instance, it is ok to substitute in $[\ A\ ]\ dip$. For this work, we will always have this guarantee. We will often use them to use single variables instead of $[\ A\ ]$. For example, if $f = [\ A\ ]$, we might want to write $f\ call$ in place of A.

$$x\ [\ A\ ]\ cons \Leftrightarrow [\ x\ A\ ] \Leftrightarrow [\ x\ [\ A\ ]\ call\ ] \Leftrightarrow [\ x\ f\ call\ ] \Leftrightarrow x\ f\ cons$$

4 FROM ARBITRARY COMBINATORS TO CONCATENATIVE

Now that we have finished the review of the existing work, we can finally get to the conversion algorithms between combinatory logic and concatenative calculus. In this chapter, we describe the first conversion which translates any combinator expression into an equivalent stack program.

One obstacle is that combinatory logic leaves the reduction order unspecified, while concatenative calculus is fully deterministic. Because of that, we will discuss two versions for this conversion. The first reduces the combinator in call-by-name order. The second reduces the combinator in call-by-value order. In Figure 8, we show an example that compares these two orders.

While researching this topic, we discovered that the call-by-name strategy is simpler and more uniform than the call-by-value counterpart. Because of that, we will show call-by-name first, in Section 4.1. In Section 4.2, we will show steps towards a call-by-value conversion.

4.1 THE CALL-BY-NAME CONVERSION

The general algorithm for our call-by-name conversion can be separated in two steps. First, we translate the input combinatory logic expression to an analogous concatenative calculus quotation. After this step, every expression turns into a quotation (function). In the second half, we insert $call$ instructions in order to run the quoted expressions. We insert one $call$ instruction for each reduction step done by the original combinatory logic expression.

Let's begin by comparing what the basic combinators and their analogous counterparts in concatenative logic. In Figure 9, the left column shows combinatory expressions. The top one is the initial expression and the bottom one is its result. The right arrow translates the combinatory expression to our desired stack program. The $\ulcorner\alpha\urcorner$ represents the translation of the combinator α. Parenthesis on the left translate to quotations on the right. This is how concatenative calculus implements partial evaluation (currying).

$B\ (B\ C)\ K\ f\ x\ y\ z$	$B\ (B\ C)\ K\ f\ x\ y\ z$
$B\ C\ (K\ f)\ x\ y\ z$	$B\ C\ (K\ f)\ x\ y\ z$
$C\ (K\ f\ x)\ y\ z$	$C\ (K\ f\ x)\ y\ z$
$K\ f\ x\ z\ y$	$C\ f\ y\ z$
$f\ z\ y$	$f\ z\ y$
(a) Call-by-name	(b) Call-by-value

Figure 8 – Different evaluation orders for combinatory logic

$$
\begin{array}{rcrl|rcrl}
B\ f\ g\ x & \Rightarrow & x\ g\ f\ \ulcorner B\urcorner & | \mathit{calls} & S\ f\ g\ x & \Rightarrow & x\ g\ f\ \ulcorner S\urcorner & | \mathit{calls} \\
f\ (g\ x) & \Rightarrow & [\ x\ g\ \mathit{call}\ ]\ f & | & f\ x\ (g\ x) & \Rightarrow & [\ x\ g\ \mathit{call}\ ]\ x\ f & | \\
\\
C\ f\ x\ y & \Rightarrow & y\ x\ f\ \ulcorner C\urcorner & | \mathit{calls} & W\ f\ x & \Rightarrow & x\ f\ \ulcorner W\urcorner & | \mathit{calls} \\
f\ y\ x & \Rightarrow & x\ y\ f & | & f\ x\ x & \Rightarrow & x\ x\ f & | \\
\\
K\ x\ y & \Rightarrow & y\ x\ \ulcorner K\urcorner & | \mathit{calls} & I\ x & \Rightarrow & x\ \ulcorner I\urcorner & | \mathit{calls} \\
x & \Rightarrow & x & | & x & \Rightarrow & x & |
\end{array}
$$

Figure 9 – Intuition for the conversion

$\ulcorner B\urcorner$	$:=$	$[\ [\ \mathit{cons}\ ]\ \mathit{dip}\ ]$
$\ulcorner C\urcorner$	$:=$	$[\ [\ \mathit{swap}\ ]\ \mathit{dip}\ ]$
$\ulcorner K\urcorner$	$:=$	$[\ [\ \mathit{zap}\ ]\ \mathit{dip}\ ]$
$\ulcorner S\urcorner$	$:=$	$[\ [\ \mathit{sons}\ ]\ \mathit{dip}\ ]$
$\ulcorner W\urcorner$	$:=$	$[\ [\ \mathit{dup}\ ]\ \mathit{dip}\ ]$
$\ulcorner I\urcorner$	$:=$	$[\ [\ \varepsilon\ ]\ \mathit{dip}\ ]$
$\ulcorner \alpha\ \beta\urcorner$	$:=$	$[\ \ulcorner\beta\urcorner\ \ulcorner\alpha\urcorner\ \mathit{call}\ ]$

Table 3 – The call-by-name conversion

On the right column, the bottom expression is also the top one's result. This follows our goal of simulating every reduction step of the combinatory expression. Remember that we do not need to be conserned about the amount of reduction steps on the translated side; it is sufficient for the top expression to be able to reach the bottom one. There is left to tell how to implement the basic combinators in terms of basic stack instructions and where calls should be inserted.

In Table 3, we show how to convert combinatory expressions in terms of basic stack instructions. All of the basic combinators have a dip, which jumps over the f argument and executes some code on the other arguments. The combinators C, K and W run swap, zap and dup, respectively. The combinator B introduces parenthesis, grouping two sub-expressions into one. The way to group the two sub-expressions is to run cons to introduce a quotation. We do not use call because in call-by-name semantics it is not the time to run that quotation yet. Something similar happens to S, who also introduces parenthesis. Here we needed to invent sons, who is analogous to the more famous sip, in the same way cons is analogous to call.

Our translation for the combinator I runs nothing after the f argument. We remark that ε is the empty sequence, so $[\ ]$ and $[\ \varepsilon\ ]$ are syntactically the same. One might have tried to define $\ulcorner I\urcorner$ as plain $[\ ]$, without dip. However, we do not that because I expects an argument. We do not want to allow I to reduce without one. The chosen definition with dip also fits the pattern.

Lastly, we come to the rule for function application. We translate the application $\ulcorner\alpha\beta\urcorner$ to $\ulcorner\beta\urcorner\ \ulcorner\alpha\urcorner\ \mathit{cons}$. Similarly to what was done with B, we are partially applying β to α. However in Table 3, we write the cons as the equivalent $[\ \ulcorner\beta\urcorner\ \ulcorner\alpha\urcorner\ \mathit{call}\ ]$ because

$$
\begin{array}{lcl}
(B\ (B\ K)\ C\ f\ x\ y\ z) & \Rightarrow & \texttt{[}\ z\ y\ x\ f\ c\ \texttt{[}\ k\ b\ call\ \texttt{]}\ b\ call\ \texttt{]}\ |\ \textbf{call}^4 \\
B\ (B\ K)\ C\ f\ x\ y\ z & \Rightarrow & z\ y\ x\ f\ c\ \texttt{[}\ k\ b\ call\ \texttt{]}\ b\ |\ call\ \textbf{call}^3 \\
(B\ K)\ (C\ f)\ x\ y\ z & \Rightarrow & z\ y\ x\ \texttt{[}\ f\ c\ call\ \texttt{]}\ \texttt{[}\ k\ b\ call\ \texttt{]}\ |\ \textbf{call}^3 \\
B\ K\ (C\ f)\ x\ y\ z & \Rightarrow & z\ y\ x\ \texttt{[}\ f\ c\ call\ \texttt{]}\ k\ b\ |\ call\ \textbf{call}^2 \\
K\ (C\ f\ x)\ y\ z & \Rightarrow & z\ y\ \texttt{[}\ x\ f\ c\ call\ \texttt{]}\ k\ |\ \textbf{call}^2 \\
(C\ f\ x)\ z & \Rightarrow & z\ \texttt{[}\ x\ f\ c\ call\ \texttt{]}\ |\ \textbf{call}^1 \\
C\ f\ x\ z & \Rightarrow & z\ x\ f\ c\ |\ call \\
f\ z\ x & \Rightarrow & x\ z\ f\ |
\end{array}
$$

Figure 10 – All the required calls can be inserted at the end

that is already in a quotation form.

Now we turn our attention to the call-insertion issue. We translate expressions to quotations, but, in concatenative calculus, those quotations will not run until they are called. Therefore, we need to insert enough calls in the right places to fully evaluate the translated expression.

In Figure 10, we show an example of call insertion. To reduce clutter, we abbreviate conversions of the basic combinators: $b = \ulcorner B \urcorner$, $c = \ulcorner C \urcorner$, and so on. On the left column, we show the reduction steps of $B(BK)C$. In gray, we added some phantom expressions with extra parenthesis. They align precisely with the stack programs on the right. If we only count the actual expressions in black, we have 4 reduction steps. Therefore, we need to insert 4 calls in the first line. Each actual reduction step inserts one call. The inserted calls are written in bold. In our call-by-name conversion, all of the calls can go at the end of the stack program because all unevaluated expressions are represented as quotations. We remark that some reductions steps were skipped on the right side and there is no problem. The important property for the simulation is that all steps in the combinatory side, have a matching step in the concatenative side. This concludes the call-by-name conversion algorithm.

Our lowercase combinators $bckswi$ are inspired by similar combinators created by Kerby (KERBY, 2002). One difference is that, in Kerby's approach, these lowercase combinators also dequote the first argument. For instance, in Kerby's system, $\texttt{[}\ B\ \texttt{]}\ \texttt{[}\ A\ \texttt{]}\ k$ reduces to A. This avoids the need to insert calls. At first, Kerby's version might appear strictly superior to ours. The main reason why we chose to add explicit calls was because it will allow us to model other reduction strategies, such as call-by-value.

4.2 THE CALL-BY-VALUE CONVERSION

The call-by-value reduction strategy reduces the arguments as much as possible, before it reduces the function. This poses a problem for us. When an expression has more than one reduction point, we have to know which one to reduce. For example, in $C(Kfx)yz$, should we reduce the C or the K first? Back in the call-by-name version, the

$$
\begin{array}{lcrll}
B\ (B\ C)\ K\ f\ x\ y\ z & \Rightarrow & z\ y\ x\ f\ k\ [\ c\ b^{\dagger}\ call\ ]\ b \mid \mathbf{call}^{3} & & \mathbf{(0)} \\
B\ C\ (K\ f)\ x\ y\ z & \Rightarrow & z\ y\ x\ [\ f\ k\ call\ ]\ c\ b^{\dagger} \mid \mathbf{call}^{2} & & \mathbf{(1)} \\
C\ (K\ f\ x)\ y\ z & \Rightarrow & z\ y\ x\ f\ k \mid call\ c\ \mathbf{call}^{1} & & \mathbf{(2)} \\
C\ f\ y\ z & \Rightarrow & z\ y\ f\ c \mid \mathbf{call}^{1} & & \mathbf{(3)} \\
f\ z\ y & \Rightarrow & y\ z\ f \mid & & \mathbf{(4)}
\end{array}
$$

Figure 11 – The need for b and $b^{\dagger}$

$$
\begin{array}{rcccl}
b^{\dagger} & := & [\ [\ call\ ]\ dip\ ] & = & [\ [\ cons\ ]\ dip\ [\ \mathbf{call}\ ]\ dip\ ] \\
s^{\dagger} & := & [\ [\ sip\ ]\ dip\ ] & = & [\ [\ sons\ ]\ dip\ [\ \mathbf{call}\ ]\ dip_2\ ]
\end{array}
$$

Figure 12 – Definition of $b^{\dagger}$ and $s^{\dagger}$

$$
\begin{array}{rcccl}
b & := & [\ [\ cons\ ]\ dip\ ] & = & [\ [\ cons\ ]\ dip\ [\ ]\ dip\ ] \\
s & := & [\ [\ sons\ ]\ dip\ ] & = & [\ [\ sons\ ]\ dip\ [\ ]\ dip_2\ ]
\end{array}
$$

Figure 13 – Alternative definition of b and s

answer was easy: we always reduce the outermost redex (in this case, C). But in the call-by-value version, it depends whether the argument is reducible (a redex).

Now we will see how the problem manifests in the conversion procedure. For instance, consider the combinator $B(BC)K$, shown in Figure 11. Let's pay attention to those B, because they are the ones that potentially creates a reducible sub-expression in argument position. In line (0), we see that the first B behaves like a $cons$, because Kf is not yet reducible. But the second B, in line (1), creates the reducible sub-expression Kfx. When translating this B to stack instructions it should perform a $call$, instead of a $cons$. Therefore, we need alternative versions of b and s. We will call them $b^{\dagger}$ and $s^{\dagger}$. As observed in Figure 12, $b^{\dagger}$ performs a $call$, and $s^{\dagger}$ performs a sip.

Back to Figure 11, pay attention to the number of calls. As we did before in call-by-name version, we need to insert one call for each reduction in the combinatory logic site. But for call-by-value we also count each † as an inserted **call**. Thus, 3 **calls** plus one † matches the 4 reductions on the left. Another way to look at it is that we could define b and s with a slot to insert a call, shown in Figure 13. In this version, to go from b to $b^{\dagger}$ is literally to insert a **call**.

But wait, there is more! Sometimes a reducible sub-expression might reduce more than once. In those cases, we need to insert a call inside the quotation, so that once that quotation is called, it has enough call-fuel to reduce all the way. In Figure 14, we show an example of this. After step (1), the sub-expression $CKxy$ can be reduced twice, to Kyx then y. By default, $b^{\dagger}$ gives one call worth of fuel for the quotation to reduce. Thus, to enable the second reduction, we must to insert a second call there, inside the quotation.

To conclude, the structure of both conversions, call-by-name and call-by-value, are similar. The difference lies in the call-insertion procedure. The call-by-name version

$$
\begin{array}{lcrll}
B\ (B\ W)\ (C\ K)\ x\ y\ z & \Rightarrow & z\ y\ x\ \texttt{[}\ k\ c\ \mathit{call}\ \textbf{call}\ \texttt{]}\ \texttt{[}\ w\ b^{\dagger}\ \mathit{call}\ \texttt{]}\ b & \mid \textbf{call}^{3} & \textbf{(0)} \\
B\ W\ (C\ K\ x)\ y\ z & \Rightarrow & z\ y\ \texttt{[}\ x\ k\ c\ \mathit{call}\ \textbf{call}\ \texttt{]}\ w\ b^{\dagger} & \mid \mathit{call}\ \textbf{call}^{1} & \textbf{(1)} \\
W\ (C\ K\ x\ y)\ z & \Rightarrow & z\ y\ x\ k\ c & \mid \mathit{call}\ \textbf{call}\ w\ \textbf{call}^{1} & \textbf{(2)} \\
W\ (K\ y\ x)\ z & \Rightarrow & z\ x\ y\ k & \mid \textbf{call}\ w\ \textbf{call}^{1} & \textbf{(3)} \\
W\ y\ z & \Rightarrow & z\ y\ w & \mid \textbf{call}^{1} & \textbf{(4)} \\
y\ z\ z & \Rightarrow & z\ z\ y & \mid & \textbf{(5)}
\end{array}
$$

Figure 14 – $b^{\dagger}$ with reducing multiple times

always inserts calls outside at the end of the program. On the other hand, the call-by-value version may insert some calls inside, in quotations or as a †. The tricky part is to know when these internal insertions are required. What we presented needs to reduce both expressions—combinatory and concatenative—side by side. A possible alternative could be to introduce a type system and a type-aware conversion, but we have not investigated that yet.

5 FROM REGULAR COMBINATORS TO CONCATENATIVE

As mentioned in the introduction, in some sense C corresponds to $swap$ and K corresponds to zap. But how does this correspondence extends to non-elementary combinators, such as $B(BK)C$, $BC(BC)$ and $B(BS)B$? In this chapter, we will tackle the first direction of this question: how to convert from combinators to stack programs in a way that complements this correspondence. We will identify that we cannot do that in the general case, but there is a special subset of combinators which allows us to translate to smaller and more direct stack programs. This special subset are the regular combinators, defined in Section 5.2.

We will start with a concrete example: the combinator $B(BK)C$. In Figure 15, we show its translation $\ulcorner B(BK)C \urcorner$, using the general call-by-name conversion from Section 4.1. But observing its evaluation, shown in the left half of Figure 17, we see that this combinator keeps f in place and only touches the other arguments. Can we take advantage of this property to make a more direct translation, where we shuffle the non-f arguments, then follow with an implicit f $call$? In this case, the stack shuffling would be only [zap] dip $swap$, as shown in Figure 16.

Both the general and direct versions of $B(BK)C$ are equivalent and have the same number of useful elementary operations, namely a single $swap$ and a single zap. But the general one is less intuitive, because of the conses, calls, quotations and dips used to carry the f around. The direct one can be simpler because it makes the f implicit. We can do that because the f argument is a continuation in $B(BK)C$.

A **continuation** is a function argument that denotes the remaining of the computation to be executed; functions that receive a continuation return their result values by passing them to the continuation instead of just returning them. **Continuation-passing style programs** are programs built using only continuations. **Direct programs** where the continuation is implicit are shorter and more intuitive than the counterpart in explicit continuation-passing style. So, in the following sections, we will start naming the continuation as q instead of f to show that it is special. (We choose q because c and k were already taken.)

Previously, we promised that our conversions would preserve the reduction steps of the original combinator. Figure 17 shows that this is still the case for the direct

[[[$swap$] dip] [[[[zap] dip] [[$cons$] dip] $call$] [[$cons$] dip] $call$] $call$]

Figure 15 – General conversion: $\ulcorner B(BK)C \urcorner$

[zap] dip $swap$

Figure 16 – Direct conversion: $|B(BK)C\rangle$

$B\ (B\ K)\ C\ f\ x\ y\ z$	$z\ y\ x \mid [\ zap\ ]\ dip\ swap\ f\ call$
$B\ K\ (C\ f)\ x\ y\ z$	$z\ y\ x \mid [\ zap\ ]\ dip\ swap\ f\ call$
$K\ (C\ f\ x)\ y\ z$	$z\ y \mid zap\ x\ swap\ f\ call$
$(C\ f\ x)\ z$	$z \mid x\ swap\ f\ call$
$C\ f\ x\ z$	$z\ x \mid swap\ f\ call$
$f\ z\ x$	$x\ z \mid f\ call$

Figure 17 – The direct conversion $|B(BK)C\rangle$ also preserves reduction steps

$q\ \ulcorner B\ (B\ K)\ C \urcorner\ call^5$	general conversion	**(0)**
$q\ [\ c\ [\ [\ k\ b\ call\ ]\ b\ call\ ]\ call\ ]\ call^5$	definition of $call$	
$q\ c\ [\ [\ k\ b\ call\ ]\ b\ call\ ]\ call\ call^4$	definition of $call$	
$q\ c\ [\ k\ b\ call\ ]\ b\ call\ call^4$	definition of $b\ call$	
$q\ c\ cons\ [\ k\ b\ call\ ]\ call^4$	definition of $cons$	
$[\ q\ c\ call\ ]\ [\ k\ b\ call\ ]\ call^4$	definition of $call$	
$[\ q\ c\ call\ ]\ k\ b\ call\ call^3$	expand $b\ call$	**(1)**
$[\ q\ c\ call\ ]\ cons\ k\ call^3$	expand $c\ call$	
$[\ swap\ q\ ]\ cons\ k\ call^3$	expand $k\ call$	
$[\ swap\ q\ ]\ cons\ [\ zap\ ]\ dip\ call^2$	cons-dip lemma	**(2)**
$[\ zap\ ]\ dip\ [\ swap\ q\ ]\ call\ call$	definition of $call$	
$[\ zap\ ]\ dip\ swap\ q\ call$	direct conversion	
$\|B\ (B\ K)\ C\rangle\ q\ call$		**(3)**

Figure 18 – Steps from general version to direct version

conversion. The first B, the outermost one, only touches the continuation f, but not the other arguments. Now that f is implicit, this step disappears. The second B, the one inside parenthesis, behaves like a dip, hiding x from the upcoming K. Finally, K zaps y and C swaps x and z.

5.1 EQUIVALENCE OF GENERAL AND DIRECT VERSIONS

In this section, we will prove that general and direct translations of $B(BK)C$ are indeed equivalent, via equational reasoning. We will use a three-step strategy, shown in Figure 18. In the first step, from (0) to (1), we start with the general version, add q to the left end, and then apply definitions of b, c, k and $cons$ as many times as we can. Because of the nature of the general conversion, we also need to add a sufficient amount of calls to the right end to be able to run it. Recall that exponentiation stands for repetition.[1] In the second step, from (1) to (2), we expand the abbreviations b, c, k, leaving only primitive stack operations.

Once we are done simplifying, in line (2), the two interesting stack instructions are still present: $swap\ q$ inside a quotation, and $[\ zap\ ]\ dip$ towards the end. To arrive at

[1] Also, recall that the definitions for instructions may be found in Table 2 at Chapter 3 and definitions for b, c, k in Table 3 at Section 4.1

$$f\ cons\ g\ dip\ call \Leftrightarrow g\ dip\ f\ call$$

(a) The equivalence

$f\ cons\ g\ dip\ call$	add an arbitrary a to the left
$a\ f\ cons\ g\ dip\ call$	definition of $cons$
$[\ a\ f\ call\ ]\ g\ dip\ call$	definition of dip
$g\ call\ [\ a\ f\ call\ ]\ call$	definition of $call$
$g\ call\ a\ f\ call$	reverse definition of dip
$a\ g\ dip\ f\ call$	remove a on the left
$g\ dip\ f\ call$	

(b) The proof

Figure 19 – The cons-dip lemma

$$f\ cons^n\ g\ dip\ call \Leftrightarrow g\ dip_n\ f\ call$$

(a) The equivalence

$f\ cons^0\ g\ dip\ call$	definition of $cons^0$
$f\ g\ dip\ call$	definition of dip
$g\ call\ f\ call$	reverse definition of dip_0
$g\ dip_0\ f\ call$	

(b) The base case

$f\ cons^{n+1}\ g\ dip\ call$	add arbitrary a to the left
$a\ f\ cons^{n+1}\ g\ dip\ call$	definition of $cons$
$[\ a\ f\ call\ ]\ cons^n\ g\ dip\ call$	inductive hipothesis
$g\ dip_n\ [\ a\ f\ call\ ]\ call$	definition of $call$
$g\ dip_n\ a\ f\ call$	reverse definition of dip
$a\ [\ g\ dip_n\ ]\ dip\ f\ call$	reverse definition of dip_{n+1}
$a\ g\ dip_{n+1}\ f\ call$	remove arbitrary a on the left
$g\ dip_{n+1}\ f\ call$	

(c) The inductive case

Figure 20 – The general cons-dip theorem

our goal, what is left is to move the q all the way to the right. For that, we will use the cons-dip lemma, which is shown in Figure 19.

We prove the lemma in Figure 19b, using f and g as arbitrary quotations. The last step of the proof removes the generic a from the left side of the program. This removal is analogous to η-reduction in lambda calculus. The cons-dip lemma can also be generalized for any amount of conses. The general cons-dip theorem (Figure 20) can be proved by induction in the number of conses.

Finally, from (2) to (3), we can continue with the third and last step: move q to the right side of the program. We use the cons-dip lemma with $f = [\ swap\ q\ ]$ and $g = [\ zap\ ]$. After removing one last quotation, the result is the direct version of the

stack combinator.

5.2 REGULAR COMBINATORS

Now we will extend what we have done to $B(BK)C$ in the previous section to other combinators. When can we take the general translation of a combinator and massage it to its direct version? What kind of combinator does it work with? Which combinators allows us to fully move q to the right? In a specification level, we want a combinator whose first argument is a special function q, the continuation, and it results in q applied to zero or more terms without q. The combinator may reorder, ignore, duplicate and apply the non-q arguments onto themselves. This kind of combinator is known as a **regular combinator**. Let's see some examples. All of the chosen basic combinators are regular: q appears only once in the result, in head position.

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

Some combinators are not regular. For instance, CI, KI, SI and CB are not regular, because q does not appear in head position, and therefore it is not the continuation. CI, SI and WI break the rule because q appears as one of the values. By the way, CI is also known as T, CB is also known as Q and WI is also known as M.

$$\begin{array}{rcl} CI\ q\ x & = & x\ q \\ KI\ q\ x & = & x \\ SI\ q\ x & = & x\ (q\ x) \\ CB\ q\ x\ y & = & x\ (q\ y) \\ WI\ q & = & q\ q \end{array}$$

Lastly, here are some examples of non-elementary regular combinators:

$$\begin{array}{rrcl} BBB & q\ x\ y\ z & = & q\ (x\ y\ z) \\ BBC & q\ x\ y\ z & = & q\ z\ (x\ y) \\ BC(BC) & q\ x\ y\ z & = & q\ y\ z\ x \\ B(BK)C & q\ x\ y\ z & = & q\ z\ x \\ B(BS)B & q\ x\ y\ z & = & q\ (x\ z)\ (y\ z) \end{array}$$

5.3 REGULAR BY CONSTRUCTION COMBINATORS

In the previous section, the definition for a regular combinator only cares about the end result of the combinator. This is not sufficient for us: we reiterate that we are still interested in distinguishing combinators that reach the same result through distinct

$W\ (B\ (B\ (B\ C))\ K)\ q\ x\ y\ z$
$(B\ (B\ (B\ C))\ K)\ q\ q\ x\ y\ z$
$(B\ (B\ C))\ (K\ q)\ q\ x\ y\ z$ $\quad$ $B\ C\ q\ x\ y\ z$
$(B\ C)\ (K\ q\ q)\ x\ y\ z$ $\quad$ $C\ (q\ x)\ y\ z$
$C\ (K\ q\ q\ x)\ y\ z$ $\quad$ $q\ x\ z\ y$
$(K\ q\ q\ x)\ z\ y$
$q\ x\ z\ y$

Figure 21 – Reduction steps of $W(B(B(BC))K)$ and BC

R	$:=$	Regular by construction combinator
	$B \mid C \mid K \mid S \mid W \mid I$	*basic combinators*
	$\mid\ B\ \alpha$	*B with one argument*
	$\mid\ B\ \alpha\ \beta$	*B with two arguments*

Figure 22 – Regular by construction combinators

paths. For example, BCC is a roundabout identity function that maps $f\ x\ y$ to $f\ x\ y$ but we want to translate it as $swap\ swap$, not as a no-op.

In addition to that, we want to avoid the messy cases where a combinator makes a mess with the continuation q only to clean it up later. This would make it more difficult to make q implicit. For instance, take $W(B(B(BC))K)$. In Figure 21, we see that it duplicates q than in a later moment discards one of the copies. To avoid these messy combinators, we will only accept well-behaved ones, such as BC.

Well-behaved regular combinators never duplicate the continuation q, nor do they zap it or swap it out of head position. We can create rules for constructing combinators which only produce well-behaved ones. Let a combinator be **regular by construction** if it has one of the forms in Figure 22, where α and β are also regular by construction. Reducible combinators, such as $K\alpha\varphi = \alpha$ and $WB\alpha = B\alpha\alpha$, are not considered regular by construction, even if they reduce to one.

The base cases in Figure 22 are the basic combinators *BCKSWI*, which are all regular. We can create larger combinators by using the inductive rules $B\alpha$ and $B\alpha\beta$. Let's begin with the first, $B\alpha$, which serves a similar purpose as dip. The $B\alpha qx = \alpha(qx)$ does whatever α does, but, before it, passes the x to q so that α can *dig deeper* and work on the latter arguments. Back in our $B(BK)C$ example, this applied to the B in BK, which turned into `[` zap `]` dip.

The $B\alpha\beta$ case composes regular combinators. We have $B\alpha\beta q = \alpha(\beta q)$. Because the first argument of a regular combinator is a continuation, $\alpha(\beta q)$ first executes α with βq as its continuation. This in turn, will run β receiving α's result and with q as its continuation. Back in our $B(BK)C$ example, this case happened to the outside B, which arranged to run BK before C.

There are no more cases with B because $B\alpha\beta\gamma$ would be reducible. There are no cases for KSW either. $K\alpha q = \alpha$ discards q. and $S\alpha qx = \alpha x(qx)$ moves q out of head

$B\ B\ B\ q\ x\ y\ z$	$B\ C\ (B\ C)\ q\ x\ y\ z$	$B\ (B\ S)\ B\ q\ x\ y\ z$
$B\ (B\ q)\ x\ y\ z$	$C\ (B\ C\ q)\ x\ y\ z$	$B\ S\ (B\ q)\ x\ y\ z$
$B\ q\ (x\ y)\ z$	$B\ C\ q\ y\ x\ z$	$B\ q\ x\ z\ (y\ z)$
$q\ (x\ y\ z)$	$q\ y\ z\ x$	$q\ (x\ z)\ (y\ z)$

Figure 23 – Examples of well-behaved combinators

$W\ K\ q$	$C\ (C\ I)\ q\ x$	$S\ (K\ S)\ K\ q\ x\ y$
$K\ q\ q$	$C\ I\ x\ q$	$K\ S\ q\ (K\ q)\ x\ y$
q	$I\ q\ x$	$S\ (K\ q)\ x\ y$
	$q\ x$	$K\ q\ y\ (x\ y)$
		$q\ (x\ y)$

Figure 24 – Examples of messy regular combinators

position. $S\alpha\beta q = \alpha q(\beta q)$ and $W\alpha q = \alpha qq$ both duplicate q.

The only combinator left is C. $C\alpha qx = \alpha xq$ does not work. The only possibility that might is $C\alpha\varphi q = \alpha q\varphi$. It does whatever α does to q but inserts a new argument φ to αq. Within this context, φ can be anything. Using this form introduces a new element in the argument list, which makes it an improper combinator. For now, we will prohibit the $C\alpha\varphi$ case, but we will come back to it in Section 6.2.

All of the examples of non-elementary regular combinators from the last section are by construction. In Figure 23, we reduce of some of them. Note that q never mixes itself with the other arguments.

Surely, the definition of regular by construction combinators is more restrictive than the definition from Section 5.3. As shown previously, that definition allows regular combinators that break the special treatment of q and fix it in a latter point. We show some more examples of messy combinators in Figure 24. WK and $C(CI)$ are weird identity functions. In the middle of evaluation WK duplicates q then immediately discards one of the copies. $C(CI)$ takes q out of head position, then puts it back. $S(KS)K$ is an example of the B combinator written using the *SK* base. It is not regular by construction because it duplicates q. Many of the *SKI* combinators exhibit this sort of behavior, where K discards the thing that S just duplicated.

To finish this section we will make a small diversion about the multiple definitions of regular combinator. We found two definitions in the literature. The newest one, which is similar to the one we made in Section 5.2, can be found in Curry's 1958 treatise (CURRY et al., 1958). The older definition is the *Reguläre Kombinatoren* of Curry's thesis (CURRY, 1930; CURRY, 1932). This definition is syntactic and matches our definition of regular by construction.

$\lvert B\rangle$	$:=$	$cons$	$\lvert S\rangle$	$:=$	$sons$	$\lvert B\ \alpha\rangle$	$:=$	$[\ \lvert\alpha\rangle\ ]\ dip$
$\lvert C\rangle$	$:=$	$swap$	$\lvert W\rangle$	$:=$	dup	$\lvert B\ \alpha\ \beta\rangle$	$:=$	$\lvert\alpha\rangle\ \lvert\beta\rangle$
$\lvert K\rangle$	$:=$	zap	$\lvert I\rangle$	$:=$	ε			

Table 4 – The regular conversion

5.4 REGULAR CONVERSION

Finally after all of that, we can derive a direct conversion from regular by construction combinators to concatenative calculus. Given a regular by construction combinator γ, then $\lvert\gamma\rangle$ is a sequence of stack instructions analogous to γ. The rules are shown in Table 4. The $\lvert I\rangle$ case maps to ε, the empty sequence. All the other base cases have a direct mapping to a single instruction in concatenative calculus. It means that, for these cases, one reduction step in combinatory logic stands for exactly one reduction step in concatenative logic. This is what we wanted all along! The translation done in Chapter 4 mapped a single combinator to a quotation with multiple instructions inside.

In the first inductive case, $\lvert B\alpha\rangle$ maps to a dip helper. When reducing that, we will use one step to push the quotation, another one to run dip, and then the steps that run $\lvert\alpha\rangle$. And, lastly, we finish it with by pushing the previously saved stack value. This simulates the reduction of the original combinator which runs B (the dip) before running α. In the other inductive case, $\lvert B\alpha\beta\rangle$ maps to a concatenation, running α first, then β. This also preserves the original combinator's reduction order.

At last, we will show some conversion examples of the combinators previously mentioned in this chapter. Pay attention to B, it has three different meanings in different contexts. In $B(BK)C$, the outside B composes BK with C; the inner B translates to a dip. In BBB, the first B is a composition, while the others are $cons$. The others base combinators are simpler: C always translates to $swap$, and S always translates to $sons$.

$\lvert B\ (B\ K)\ C\rangle$	$\lvert B\ B\ B\rangle$	$\lvert B\ C\ (B\ C)\rangle$	$\lvert B\ (B\ S)\ B\rangle$
$\lvert B\ K\rangle\ \lvert C\rangle$	$\lvert B\rangle\ \lvert B\rangle$	$\lvert C\rangle\ \lvert B\ C\rangle$	$\lvert B\ S\rangle\ \lvert B\rangle$
$[\ \lvert K\rangle\ ]\ dip\ swap$	$cons\ cons$	$swap\ [\ \lvert C\rangle\ ]\ dip$	$[\ \lvert S\rangle\ ]\ dip\ cons$
$[\ zap\ ]\ dip\ swap$		$swap\ [\ swap\ ]\ dip$	$[\ sons\ ]\ dip\ cons$

6 BACK FROM CONCATENATIVE TO COMBINATORS

In this chapter we will introduce three algorithms to translate concatenative programs back to combinatory logic. The first one converts from first-order concatenative programs (defined in Section 6.1) to regular combinators. The second one extends the first, to allow programs that push values to the stack. This extension might produce improper combinators. The third and last algorithm allows higher order programs, featuring $call$ and standalone dip. This allows to use concatenative calculus to its full extent, but the resulting combinators may no longer be regular.

Our algorithms are almost a backwards version of the algorithm from Chapter 5. The only problem is that the aforementioned conversion is not a bijection: some combinatory expressions map to the same concatenative program. The blame lies on the composition done by B, which forms a monoid (it is associative and it has the neutral element). Associativity is a problem because $|B(B\alpha\beta)\gamma\rangle$ and $|B\alpha(B\beta\gamma)\rangle$ both map to $|\alpha\rangle\ |\beta\rangle\ |\gamma\rangle$. The combinator I being the neutral element of composition is a problem because $|I\rangle = \varepsilon$. All of $B\alpha I$, α and $BI\alpha$ will map to the same term $|\alpha\rangle$. These equivalences are shown in Figures 25 and 26. To work around this issue, our algorithms will produce an arbitrary combinator from the equivalence class.

6.1 FROM FIRST-ORDER CONCATENATIVE TO COMBINATORS

The conversion from Chapter 5 only generates a subset of concatenative programs. In these programs, dip are always preceded by a quotation and quotations are always immediately followed by a dip. We will say that programs in this subset are **first-order**. Their grammar is shown in Table 27. Operations are the non-dequoting instructions or a quotation followed by dip. A program is a list of operations associated to the right. One weird choice we made is that this list is not nil-terminated, instead we have separated it into empty and non-empty programs. The reason is that we do not want the result of

$\|B\ (B\ \alpha\ \beta)\ \gamma\rangle$	$\|B\ \alpha\ (B\ \beta\ \gamma)\rangle$
$\|B\ \alpha\ \beta\rangle\ \|\gamma\rangle$	$\|\alpha\rangle\ \|B\ \beta\ \gamma\rangle$
$\|\alpha\rangle\ \|\beta\rangle\ \|\gamma\rangle$	$\|\alpha\rangle\ \|\beta\rangle\ \|\gamma\rangle$

Figure 25 – B is associative

$\|B\ \alpha\ I\rangle$	$\|\alpha\rangle$	$\|B\ I\ \alpha\rangle$
$\|\alpha\rangle\ \|I\rangle$	$\|\alpha\rangle$	$\|I\rangle\ \|\alpha\rangle$
$\|\alpha\rangle\ \varepsilon$	$\|\alpha\rangle$	$\varepsilon\ \|\alpha\rangle$
$\|\alpha\rangle$	$\|\alpha\rangle$	$\|\alpha\rangle$

Figure 26 – I is the neutral element of composition

$$
\begin{array}{lll}
O & := & \textsc{Operation} \\
 & \quad cons \mid swap \mid zap \mid sons \mid dup & \textit{base instruction} \\
 & \mid \texttt{[}\ P\ \texttt{]}\ dip & \textit{dip subprogram} \\
 & & \\
N & := & \textsc{Non-empty program} \\
 & \mid O & \textit{single operation} \\
 & \mid O\ N & \textit{restricted concatenation} \\
 & & \\
P & := & \textsc{First-order program} \\
 & \quad \varepsilon & \textit{empty program} \\
 & \mid N & \textit{non-empty program}
\end{array}
$$

Figure 27 – First-order concatenative programs

$\{cons\}$	$:=$	B	$\{sons\}$	$:=$	S	$\{\varepsilon\}$	$:=$	I
$\{swap\}$	$:=$	C	$\{dup\}$	$:=$	W			
$\{zap\}$	$:=$	K	$\{\texttt{[}\ P\ \texttt{]}\ dip\}$	$:=$	$B\ \{P\}$	$\{O\ N\}$	$:=$	$B\ \{O\}\ \{N\}$

Table 5 – The back-conversion for first-order

our translation to produce an I at the end every time.

The conversion algorithm for first-order concatenative program is described in Table 5. It is exactly the reverse of Table 4 from Section 5.4. Basic instructions translate to basic combinators. A dipped quotation becomes a single-argument B. A concatenation converts to a two-argument B. Because our structural definition of N nests to the right, the resulting combinator will also nests the Bs to the right. Here is an example:

$$
\begin{array}{l}
\{zap\ \texttt{[}\ dup\ \texttt{]}\ dip\ swap\} \\
B\ \{zap\}\ \{\texttt{[}\ dup\ \texttt{]}\ dip\ swap\} \\
B\ K\ (B\ \{\texttt{[}\ dup\ \texttt{]}\ dip\}\ \{swap\}) \\
B\ K\ (B\ (B\ \{dup\})\ C) \\
B\ K\ (B\ (B\ W)\ C)
\end{array}
$$

6.2 PUSHING VALUES

In this section, we will introduce a push-value operation, which was missing. This allows pushing constant values onto the stack, such as numbers and other literals. Because our calculus does not have numbers, quotations are the only values we have. Back in the previous section, we only allowed quotations together with a dip, but now we will allow pushing quotations by themselves. At first, this might not seam that useful because we do not have a $call$ operation yet. We will add call-like operations in the next section. However, remember that pushing would also be useful for numbers, if we had them.

$$(\varphi\ \{v\})\ q\ x\ y\ z\ \ldots$$
$$q\ \{v\}\ x\ y\ z\ \ldots$$

Figure 28 – Intuition for push operations

$\{[\ A\ ]\}$	$:=$	$T\ \{A\}$

Table 6 – The back-conversion for push

$$\{call\}\ q\ \alpha\ x\ y\ z\ \ldots \qquad \{dip\}\ q\ \alpha\ x\ y\ z\ \ldots$$
$$\alpha\ q\ x\ y\ z\ \ldots \qquad \alpha\ (q\ x)\ y\ z\ \ldots$$

Figure 29 – Intuition for $call$ and dip

In Figure 28, we show how we want the push operation to behave as a translated combinator. In the top line, we have the pushing combinator φ parameterized by the translated value $\{v\}$. The instantiated combinator $(\varphi\ \{v\})$ receives the continuation q and the remaining stack arguments x, y and z. The desired result should call the continuation with the value pushed on the top of the stack.

Which combinator is this φ? Observing this reduction rule we can recognize that our old friend T, also known as CI. (Recall that $Tvq = qv$.) In Table 6, we add this step to the conversion algorithm from stack to combinators. Observe that T is not regular, and $T\alpha$ is not proper.

6.3 HIGHER-ORDER STACK PROGRAMS

In this section, we will introduce the dequoting operations: $call$ and dip. These are the concatenative instructions left to translate. It will allow us to build higher-order programs, when functions are first-class values. It means that we can push them to the stack and run them in a later opportunity.

In Figure 29, we show how we want $call$ and dip to behave in combinatory logic. First, let's observe $\{call\}$. The instruction $call$ runs the first argument from the stack. So, its translation should call the first argument α passing to it the continuation q and the remaining stack arguments. We can use the combinator T to implement $\{call\}$. Note that this time the T appears by itself, without the parameter v.

The instruction dip gets two arguments from the stack. The top one is the quotation to run, and the argument below it is the value to be put away for later. dip should run the quotation, then restore the value. Similarly, its translation should call the first argument α, but now we pass the continuation q applied to the second argument x and the remaining stack arguments. Appling x to the continuation is the combinatory version of restoring the argument. Remember that α is regular combinator. It means that in the expression $\alpha(qx)$ the combinator α does not touch x and reduces to qx applied to result of α. We can achieve this with Q (defined as CB), who is in the same family as T.

$\{call\}$	$:=$	T
$\{dip\}$	$:=$	Q

Table 7 – The back-conversion for call and dip

$$
\begin{array}{l}
B\ \{[\ A\ ]\}\ \{dip\}\ q\ x \\
B\ (T\ \alpha)\ Q\ q\ x \\
T\ \alpha\ (Q\ q)\ x \\
Q\ q\ \alpha\ x \\
\alpha\ (q\ x) \\
B\ \alpha\ q\ x \\
(B\ \alpha)\ q\ x \\
\{[\ A\ ]\ dip\}\ q\ x
\end{array}
$$

Figure 30 – First-order and higher-order dip are equivalent

We show how to fit $call$ and dip into our conversion algorithm in Table 7. Observe that neither T nor Q are regular. For every thing to run fine we have to promise that the quotation arguments given to $call$ and dip are in fact regular. Finally, in Figure 30, we prove that the higher-order definition of dip is compatible with its first-order counterpart from Section 6.1.

7 CONCLUSION

In this work, we highlighted the similarities between two theoretical tacit programming models: combinatory logic and concatenative calculus. We showed two reversible algorithms to convert between them. The first algorithm can take any combinatory logic expression and outputs a stack program that simulates the original expression. This translation results in concatenative programs written in a continuation-passing style way, with lots of quotation pushing and dequoting. The second algorithm is able to produce smaller stack programs because it restricts the input side to regular combinators and takes advantage of their continuation-passing style.

One interesting way of looking at these conversions is how they restrain its input and output domains. The first conversion's input domain is the full set of combinators, but it cannot reach all the possible stack programs in the output domain. Its output is restricted to explicit continuation-passing style stack programs. Conversely, the second conversion restrains its input domain to regular combinators but this time the output domain unrestricted.

During the development of this work we learned things we did not expect at first. The first of them is that B has three different interpretations in the concatenative calculus: $cons$, dip and composition. We also learned about the subtle problem of call-insertions, described in Section 4.2. This issue appeared when we tried to work with call-by-value reduction order. After that, we discovered that call-insertion is much simpler in the call-by-name setting.

7.1 FUTURE WORK

This work can be expanded in many different directions. The most obvious of them is to write more formal proofs for the various simulation theorems between combinatory logic and concatenative calculus.

In Section 4.2, the call-insertion problem for call-by-value semantics was left open. We still do not have a algorithmic way of knowing whether B or S will produce a redex in its sub-expression. Furthermore, how many reduction steps will the sub-expression need to fully reduce? Possibly, it can be solved by introducing some kind of type information to the original expression.

In Section 5.3, we introduced rules to construct well-behaved regular combinators, but a few questions remain to be answered. Given a messy regular combinator, can we always find a well-behaved equivalent? Can we expand the definition to include more combinators, such as T and Q? Can we describe a set of rules to construct well-behaved regular combinators using the *SKI* base?

BIBLIOGRAPHY

BARENDREGT, H. P. et al. **The lambda calculus**. [S.l.]: North-Holland Amsterdam, 1984. v. 3.

CURRY, H. B. Grundlagen der kombinatorischen logik. **American Journal of Mathematics**, JSTOR, v. 52, n. 4, p. 789–834, 1930. This was his PhD thesis.

CURRY, H. B. Some additions to the theory of combinators. **American Journal of Mathematics**, JSTOR, v. 54, n. 3, p. 551–558, 1932.

CURRY, H. B. et al. **Combinatory logic**. [S.l.]: North-Holland Amsterdam, 1958. v. 1. Regular combinators are discussed in chapter 5.

IVERSON, K. E. A programming language. In: **Proceedings of the May 1-3, 1962, spring joint computer conference**. [S.l.: s.n.], 1962. p. 345–351.

KERBY, B. **The Theory of Concatenative Combinators**. 2002. Published online at http://tunes.org/~iepos/joy.html. Accessed 05/09/2023.

KLEFFNER, R. **A Foundation for Typed Concatenative Languages**. Dissertação (Mestrado) — Northeastern University, Abril 2017.

MOORE, C. H.; LEACH, G. C. Forth–a language for interactive computing. **Amsterdam: Mohasco Industries Inc**, 1970.

PESTOV, S.; EHRENBERG, D.; GROFF, J. Factor: A dynamic stack-based programming language. **Acm Sigplan Notices**, ACM New York, NY, USA, v. 45, n. 12, p. 43–58, 2010.

RATHER, E. D.; COLBURN, D. R.; MOORE, C. H. The evolution of forth. In: **History of programming languages—II**. [S.l.: s.n.], 1996. p. 625–670.

SCHÖNFINKEL, M. Über die bausteine der mathematischen logik. **Mathematische annalen**, Springer, v. 92, n. 3-4, p. 305–316, 1924.

THUN, M. von. **Mathematical foundations of Joy**. 1994. 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. Mirror available at http://www.kevinalbrecht.com/code/joy-mirror/index.html.

THUN, M. von; THOMAS, R. Joy: Forth's functional cousin. In: **Proceedings of the 17th EuroForth Conference**. [S.l.: s.n.], 2001.