Logical Methods in Computer Science
Vol. 11(3:25)2015, pp. 1–31
www.lmcs-online.org

Submitted Jan. 13, 2014
Published Sep. 30, 2015

ELLIPSES AND LAMBDA DEFINABILITY

MAYER GOLDBERG

Department of Computer Science, Ben-Gurion University, Beer Sheva 8410501, Israel.

e-mail address: gmayer@cs.bgu.ac.il

Abstract. Ellipses are a meta-linguistic notation for denoting terms the size of which are specified by a meta-variable that ranges over the natural numbers. In this work, we present a systematic approach for encoding such meta-expressions in the λ-calculus, without ellipses: Terms that are parameterized by meta-variables are replaced with corresponding λ-abstractions over actual variables. We call such λ-terms *arity-generic*. Concrete terms, for particular choices of the parameterizing variable are obtained by applying an arity-generic λ-term to the corresponding numeral, obviating the need to use ellipses.

For example, to find the multiple fixed points of n equations, n different λ-terms are needed, every one of which is indexed by two meta-variables, and defined using three levels of ellipses. A single arity-generic λ-abstraction that takes two Church numerals, one for the number of fixed-point equations, and one for their arity, replaces all these multiple fixed-point combinators. We show how to define arity-generic generalizations of two historical fixed-point combinators, the first by Curry, and the second by Turing, for defining multiple fixed points. These historical fixed-point combinators are related by a construction due to Böhm: We show that likewise, their arity-generic generalizations are related by an arity-generic generalization of Böhm's construction.

We further demonstrate this approach to arity-generic λ-definability with additional λ-terms that create, project, extend, reverse, and map over ordered n-tuples, as well as an arity-generic generator for one-point bases.

1. Introduction

1.1. **Motivation.** This work is concerned with λ-terms that are written using the meta-language of ellipses: Terms such as, for example, the ordered n-tuple maker: $\lambda x_1 \cdots x_n \sigma.(\sigma\ x_1 \cdots x_n)$. As the use of ellipses indicates, the syntax for such λ-terms is described *for any given* n, in the meta-language of the λ-calculus, i.e., in the language in which we describe the syntax of λ-terms. The index n is thus a meta-variable. It is only after we have picked a natural number for n, that we can write down an actual λ-term, and it will be "hard-coded" for that specific n. For example, the ordered 5-tuple maker is defined

2012 ACM CCS: [**Theory of computation**]: Models of computation.

Key words and phrases: arity-generic expressions, bases, definability, fixed-point combinators, lambda calculus, LISP/Scheme, variadic functions.

DOI:10.2168/LMCS-11(3:25)2015

as $\lambda x_1 x_2 x_3 x_4 x_5 \sigma.(\sigma\ x_1\ x_2\ x_3\ x_4\ x_5)$, can be written without ellipses, and is "hard-coded" for $n = 5$. But what if we want n, which determines the syntactic structure of the λ-term, to be an argument in the language of the λ-calculus: How do we go from a λ-term whose syntax is indexed or parameterized by a meta-variable over the natural numbers in the meta-language of the λ-calculus to a corresponding λ-term parameterized by a Church numeral?

In this work, we present a systematic approach for encoding terms whose syntax is parameterized by a meta-variable and written using ellipses, to λ-terms that take a Church numeral c_n as an argument, and return the corresponding λ-term for that given n. We call such λ-terms *arity-generic*, following the work of Weirich and Casinghino on *Arity-Generic Datatype-Generic Programming* [31]. When we speak of an arity-generic λ-term E_{AG}, we require two things:

(1) We have in mind an n-ary term E_n in the meta-language of the λ-calculus, that is parameterized by a meta-variable $n \in \mathbb{N}$. For any specific value of n, E_n is a λ-term: E_1, E_3, etc., are all λ-terms.
(2) For all $n \in \mathbb{N}$, $(E_{\text{AG}}\ c_n) =_{\beta\eta} E_n$.

1.2. **Overview.** In Combinatory Logic, bases provide a standard approach to constructing inductively larger combinators from smaller combinators. We follow this approach by augmenting the standard of $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ basis introduced by Schönfinkel [24], Curry [9, 10], Turner [28], and many others, with arity-generic generalizations $\mathbf{K}_{\text{AG}}, \mathbf{S}_{\text{AG}}$ of the respective $\mathbf{K}, \mathbf{S}$ combinators. We then encode $\mathbf{K}_{\text{AG}}, \mathbf{S}_{\text{AG}}$ in terms of $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ (Section 2). $\mathbf{K}_{\text{AG}}, \mathbf{S}_{\text{AG}}$ can then be used to encode straightforwardly those parts of the term that use ellipses using an arity-generic generalization of the bracket-abstraction algorithm for the $\{\mathbf{K}, \mathbf{S}\}$ basis.

In principle, we could have stopped at this point, since $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}, \mathbf{K}_{\text{AG}}, \mathbf{S}_{\text{AG}}\}$ would already be sufficient to encode any arity-generic term. We chose, however, to use $\mathbf{K}_{\text{AG}}, \mathbf{S}_{\text{AG}}$ to define $\mathbf{B}_{\text{AG}}, \mathbf{C}_{\text{AG}}$, which are the arity-generic generalizations of $\mathbf{B}, \mathbf{C}$, because Turner's *bracket-abstraction algorithm* for the basis $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ extends naturally to the basis $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}, \mathbf{I}_{\text{AG}}, \mathbf{K}_{\text{AG}}, \mathbf{B}_{\text{AG}}, \mathbf{C}_{\text{AG}}, \mathbf{S}_{\text{AG}}\}$. This extended algorithm (Section 4) maintains the simplicity of Turner's original algorithm, and generates compact encodings for arity-generic λ-terms.

The second part of this work (Section 5) demonstrates how the new basis can be used to encode interesting arity-generic λ-terms, such as multiple fixed-point combinators.

1.3. **Terminology, notation and list of combinators.** For background material on the λ-calculus, we refer the reader to Church's original book on the λ-calculus, *The Calculi of Lambda Conversion* [7], Curry's two volumes *Combinatory Logic I, II* [9, 10], and Barendregt's encyclopedic textbook, *The Lambda Calculus: Its Syntax and Semantics* [4]. Here we briefly list the λ-terms and notation used throughout this work.

$\mathbf{I}$	$\lambda x.x$	*Identitätsfunktion* [24]
$\mathbf{K}$	$\lambda xy.x$	*Konstanzfunktion* [24]
$\mathbf{B}$	$\lambda xyz.(x\ (y\ z))$	*Zusammensetzungsfunktion* [24]
$\mathbf{C}$	$\lambda xyz.(x\ z\ y)$	*Vertauschungsfunktion* [24]

$\mathbf{S}$	$\lambda xyz.(x\ z\ (y\ z))$	*Verschmelzungsfunktion* [24]
$\langle x_1, \ldots, x_n \rangle$	$\lambda \sigma.(\sigma\ x_1 \cdots x_n)$	Ordered n-tuple [4]
$\langle \_, \ldots, \_ \rangle_n$	$\lambda x_1 \cdots x_n \sigma.(\sigma\ x_1 \cdots x_n)$	Ordered n-tuple maker [14]
σ^n_k	$\lambda x_1 \cdots x_n.x_k$	Selector: Returns the k-th of n arguments [4]
π^n_k	$\lambda x.(x\ \sigma^n_k)$	Projection: Returns the k-th projection of an ordered n-tuple [4]
c_n	$\lambda sz.(\underbrace{s\ (\cdots (s}_{n \text{ times}}\ z) \cdots))$	The n-th Church numeral [7]
S^+	$\lambda nsz.(s\ (n\ s\ z))$	Computes the *successor* on Church numerals [7]
$+$	$\lambda ab.(b\ S^+\ a)$	Computes addition on Church numerals [7]
P^-	$\lambda n.(\pi^2_1\ (n\ (\lambda p.\langle(\pi^2_2\ p), (S^+\ (\pi^2_2\ p))\rangle) \langle c_0, c_0 \rangle))$	Computes the *predecessor* on Church numerals [14], following Kleene's construction for the $\lambda\mathbf{I}\beta\eta$-calculus [17]
$\dot{-}$	$\lambda ab.(b\ P^-\ a)$	Computes the *monus* function on Church numerals [7]
False	$\lambda xy.y$	The Boolean value *False* [4]
True	$\lambda xy.x$	The Boolean value *True* [4]
Zero?	$\lambda n.(n\ (\lambda x.\mathbf{False})\ \mathbf{True})$	Computes the *zero-predicate* on Church numerals

For any λ-term P, the set of variables that occur freely in P is denoted by $\mathsf{FreeVars}(P)$. The $\equiv$ symbol denotes identity modulo α-conversion, the symbol $\twoheadrightarrow$ denotes reflexive and transitive closure of the $\beta\eta$ relation, The $=_\beta$ symbol denotes the equivalence relation induced by β-reduction. The $=_\eta$ symbol denotes the equivalence relation induced by η-reduction. The symbol $=_{\beta\eta}$, which is also abbreviated as $=$, denotes the equivalence relation induced by the $\beta\eta$ relation.

The *size* of a λ-term P, denoted by $|P|$, is the length of its abstract-syntax tree. For variable ν, and λ-terms P, Q, we have:

$$
\begin{aligned}
|\nu| &= 1 \\
|\lambda\nu.P| &= 1 + |P| \\
|(PQ)| &= 1 + |P| + |Q|
\end{aligned}
$$

The relationship between λ-terms A, B, and a function f, which maps the λ-term A to B is denoted by $A \stackrel{f}{\Longrightarrow} B$.

1.4. **The meta-language of ellipses.** The ellipsis is used extensively in the literature on the λ-calculus and combinatory logic: It appears in Church's original text on the λ-calculus [7], in Curry's texts on combinatory logic [9, 10], in Barendregt's text on the λ-calculus [4], and in many other books and articles.

As a meta-linguistic notational device, the ellipsis is very economical, but the economy often hides subtlety and complexity. For example, in the expression

$$P \equiv \lambda x.\underbrace{(S^+ \cdots (S^+}_{100 \text{ times}}\ x) \cdots)\,,$$

the ellipses serve to abbreviate an expression that would otherwise be cumbersome to write. Now consider the superficially-similar expression

$$Q_n \equiv \lambda x.\underbrace{(S^+ \cdots (S^+}_{n \text{ times}}\ x) \cdots)\,.$$

For specific values of n, the expression Q_n is a λ-expression: Q_1, Q_{23}, Q_{100}, etc., are all λ-expressions, and in fact, $Q_{100} \equiv P$. However, Q_n is not a λ-expression: Linguistically, n is a meta-variable in the meta-language of the λ-calculus, and so Q_n is rather a meta-expression.

Would it be possible to define a λ-expression that would, in some sense, "capture the essence" of Q_n? Since we use Church numerals in this paper, and since Church numerals are abstractions over the iterated composition of a function, it seems reasonable to argue that the expression $R = \lambda n.\lambda x.(n\ S^+\ x) =_\eta \lambda n.(n\ S^+)$ is our candidate: It takes a Church numeral n as an argument, and returns a function that applies to its argument the n-th composition of S^+. The relationship between Q_n and R is given by $Q_n = (R\ c_n)$. We can use this relationship, to replace a meta-expression with a λ-expression and a Church numeral, and in that sense, "eliminate" the use of ellipses.

In more complicated scenarios, ellipses and meta-variables can be combined to hide even greater complexity. For example, in Section 1.1, we described the n-tuple maker: $\langle \_, \ldots, \_ \rangle_n = \lambda x_1 \cdots x_n \sigma.(\sigma\ x_1 \cdots x_n)$. Ellipses now control the number of nested λ-abstractions, and the number of left-associated applications. How can these ellipses be eliminated? The "interface" to such a term, which we call $\langle \_, \ldots, \_ \rangle_{\text{AG}}$ would take a Church numeral c_n, and satisfy the relationship $(\langle \_, \ldots, \_ \rangle_{\text{AG}}\ c_n) = \langle \_, \ldots, \_ \rangle_n$.

Sections 2, 4, and 5 explore how all meta-linguistic ellipses can be removed from expressions in the meta-language of the λ-calculus. Put otherwise, the λ-calculus is sufficiently expressive so as to make the use of meta-linguistic ellipses unnecessary, even if they are still used as a matter of convenience.

2. Arity-generic generalizations of the $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ basis

Our goal is to define arity-generic versions of $\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$ combinators, which form the arity-generic part of a basis for arity-generic λ-expressions.

2.1. The arity-generic K combinator.

The $\mathbf{K}$ combinator, defined as $\lambda px.p$, abstracts a variable x over an expression in which x does not occur free. The n-ary generalization of $\mathbf{K}$ abstracts n variables, and is given by:

$$\mathbf{K}_n \quad \equiv \quad \lambda p x_1 \cdots x_n.p$$

Notice that $\mathbf{K}$ abstracts a single unused variable over its argument. Hence we may write:

$$\mathbf{K}_n \quad = \quad (c_n\ \mathbf{K})$$

We now define $\mathbf{K}_{\text{AG}}$ as follows:

$$\mathbf{K}_{\text{AG}} \quad \equiv \quad \lambda n.(n\ \mathbf{K})$$

This definition satisfies the requirement that $(\mathbf{K}_{\text{AG}}\ c_n) = \mathbf{K}_n$. Also note that $\mathbf{K}_0 = \mathbf{I}$, and $\mathbf{K}_1 = \mathbf{K}$.

2.2. **The arity-generic S combinator.** The $\mathbf{S}$ combinator, defined as $\lambda pqx.(p\ x\ (q\ x))$, abstracts a variable x over an application of two expressions, where x occurs free in both expressions. The n-ary generalization of $\mathbf{S}$ abstracts n variables, and is given by:[1]

$$\mathbf{S}_n \quad \equiv \quad \lambda pqx_1 \cdots x_n.(p\ x_1 \cdots x_n\ (q\ x_1 \cdots x_n))$$

We describe $\mathbf{S}_{n+1}$ in terms of $\mathbf{S}_n$:

$$\begin{aligned}
\mathbf{S}_{n+1} &= \lambda pqx_1 \cdots x_{n+1}.(p\ x_1 \cdots x_{n+1}\ (q\ x_1 \cdots x_{n+1})) \\
&= \lambda pqx_1 \cdots x_{n+1}.(\mathbf{S}\ (p\ x_1 \cdots x_n)\ (q\ x_1 \cdots x_n)\ x_{n+1}) \\
&=_\eta \lambda pqx_1 \cdots x_n.(\mathbf{S}\ (p\ x_1 \cdots x_n)\ (q\ x_1 \cdots x_n)) \\
&= \lambda pqx_1 \cdots x_n.(\mathbf{K}_n\ \mathbf{S}\ x_1 \cdots x_n\ (p\ x_1 \cdots x_n)\ (q\ x_1 \cdots x_n)) \\
&= \lambda pqx_1 \cdots x_n.(\mathbf{S}_n\ (\mathbf{K}_n\ \mathbf{S})\ p\ x_1 \cdots x_n\ (q\ x_1 \cdots x_n)) \\
&= \lambda pqx_1 \cdots x_n.(\mathbf{S}_n\ (\mathbf{S}_n\ (\mathbf{K}_n\ \mathbf{S})\ p)\ q\ x_1 \cdots x_n) \\
&=_\eta \lambda pq.(\mathbf{S}_n\ (\mathbf{S}_n\ (\mathbf{K}_n\ \mathbf{S})\ p)\ q) \\
&=_\eta \lambda p.(\mathbf{S}_n\ (\mathbf{S}_n\ (\mathbf{K}_n\ \mathbf{S})\ p)) \\
&= \lambda p.(\mathbf{B}\ \mathbf{S}_n\ (\mathbf{S}_n\ (\mathbf{K}_n\ \mathbf{S}))\ p) \\
&=_\eta (\mathbf{B}\ \mathbf{S}_n\ (\mathbf{S}_n\ (\mathbf{K}_n\ \mathbf{S}))) \\
&= ((\lambda s.(\mathbf{B}\ s\ (s\ (\mathbf{K}_n\ \mathbf{S}))))\ \mathbf{S}_n)
\end{aligned}$$

The λ-term f that takes a Church numeral c_n, and maps $\mathbf{S}_n \overset{f}{\Longrightarrow} \mathbf{S}_{n+1}$ is given by

$$\begin{aligned}
f &= \lambda ns.((\lambda s.(\mathbf{B}\ s\ (s\ (\mathbf{K}_{\text{AG}}\ n\ \mathbf{S}))))\ s) \\
&= \lambda ns.(\mathbf{B}\ s\ (s\ (\mathbf{K}_{\text{AG}}\ n\ \mathbf{S})))
\end{aligned}$$

The λ-term g such that $\langle c_n, \mathbf{S}_n \rangle \overset{g}{\Longrightarrow} \langle c_{n+1}, \mathbf{S}_{n+1} \rangle$ is given by:

$$g \quad = \quad \lambda p.\left\langle (S^+\ (\pi_1^2\ p)), (f\ (\pi_1^2\ p)\ (\pi_2^2\ p)) \right\rangle$$

Note that $\mathbf{S}_0$ abstracts over 0 arguments, so we have $\mathbf{S}_0 = \lambda pq.(pq) =_\eta \mathbf{I}$. We define $\mathbf{S}_{\text{AG}}$ by taking the n-th composition of g, applying it to $\langle c_0, \mathbf{S}_0 \rangle$, and taking the second projection:

$$\mathbf{S}_{\text{AG}} \quad \equiv \quad \lambda n.(\pi_2^2\ (n\ g\ \langle c_0, \mathbf{I} \rangle))$$

This definition satisfies the requirement that $(\mathbf{S}_{\text{AG}}\ c_n) = \mathbf{S}_n$. Also note that $\mathbf{S}_0 = \mathbf{I}$, and $\mathbf{S}_1 = \mathbf{S}$.

[1] Curry [9, page 169] uses the symbol $\mathbf{S}_n$ to denote the following generalization of $\mathbf{S}$, which is different from our own:

$$\mathbf{S}_n^{\text{Curry}} \quad \equiv \quad \lambda f g_1 \cdots g_n x.(f\ x\ (g_1\ x) \cdots (g_n\ x))$$

Nevertheless, we think that our generalization fits better here, because of the way the relevant rule in Turner's bracket-abstraction algorithm for the basis $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ generalizes to our definition of $\mathbf{S}_n$.

2.3. **The arity-generic I combinator.** The **I**-combinator is defined as $\lambda x.x$. The n-ary generalization of **I** is

$$\begin{array}{rcl}\mathbf{I}_n & \equiv & \lambda x_1 \cdots x_n.(x_1 \cdots x_n) \\ & =_\eta & \mathbf{I}\end{array}$$

Since $\mathbf{I}_n =_\eta \mathbf{I}$, this case is trivial. It is nevertheless necessary for completeness, to give the arity-generic extension of $\mathbf{I}_n$:

$$\begin{array}{rcl}\mathbf{I}_{\text{AG}} & \equiv & \lambda n.\mathbf{I} \\ & = & (\mathbf{K}\ \mathbf{I})\end{array}$$

This definition trivially satisfies the requirement that $(\mathbf{I}_{\text{AG}}\ c_n) = \mathbf{I}_n$, as $\mathbf{I}_n = \mathbf{I}$ holds trivially for all $n \in \mathbb{N}$.

2.4. **The arity-generic B combinator.** The **B** combinator, defined as $\lambda pqx.(p\ (q\ x))$ abstracts a variable x over an application of two expressions, where x occurs free in the second expression. The n-ary generalization of **B** abstracts n variables, and is given by:

$$\begin{array}{rcl}\mathbf{B}_n & \equiv & \lambda pqx_1 \cdots x_n.(p\ (q\ x_1 \cdots x_n)) \\ & = & \lambda pqx_1 \cdots x_n.(\mathbf{K}_n\ p\ x_1 \cdots x_n\ (q\ x_1 \cdots x_n)) \\ & = & \lambda pqx_1 \cdots x_n.(\mathbf{S}_n\ (\mathbf{K}_n\ p)\ q\ x_1 \cdots x_n) \\ & =_\eta & \lambda p.(\mathbf{S}_n\ (\mathbf{K}_n\ p)) \\ & = & \lambda p.(\mathbf{B}\ \mathbf{S}_n\ \mathbf{K}_n\ p) \\ & =_\eta & (\mathbf{B}\ \mathbf{S}_n\ \mathbf{K}_n)\end{array}$$

The arity-generic version of **B**, written as $\mathbf{B}_{\text{AG}}$ takes c_n and returns $\mathbf{B}_n$. We can define $\mathbf{B}_{\text{AG}}$ as follows:

$$\mathbf{B}_{\text{AG}} \equiv \lambda n.(\mathbf{B}\ (\mathbf{S}_{\text{AG}}\ n)\ (\mathbf{K}_{\text{AG}}\ n))$$

This definition satisfies the requirement that $(\mathbf{B}_{\text{AG}}\ c_n) = \mathbf{B}_n$. Also note that $\mathbf{B}_0 = \mathbf{I}$, and $\mathbf{B}_1 = \mathbf{B}$.

2.5. **The arity-generic C combinator.** The **C** combinator, defined as $\lambda pqx.(p\ x\ q)$ abstracts a variable x over an application of two expressions, where x occurs free in the first expression. The n-ary generalization of **C** abstracts n variables, and is given by:

$$\begin{array}{rcl}\mathbf{C}_n & = & \lambda pqx_1 \cdots x_n.(p\ x_1 \cdots x_n\ q) \\ & = & \lambda pqx_1 \cdots x_n.(p\ x_1 \cdots x_n\ (\mathbf{K}_n\ q\ x_1 \cdots x_n)) \\ & = & \lambda pqx_1 \cdots x_n.(\mathbf{S}_n\ p\ (\mathbf{K}_n\ q)\ x_1 \cdots x_n) \\ & =_\eta & \lambda pq.(\mathbf{S}_n\ p\ (\mathbf{K}_n\ q)) \\ & = & \lambda pq.(\mathbf{B}\ (\mathbf{S}_n\ p)\ \mathbf{K}_n\ q) \\ & = & \lambda pq.(\mathbf{B}\ \mathbf{B}\ \mathbf{S}_n\ p\ \mathbf{K}_n\ q) \\ & = & \lambda pq.(\mathbf{C}\ (\mathbf{B}\ \mathbf{B}\ \mathbf{S}_n)\ \mathbf{K}_n\ p\ q) \\ & =_\eta & (\mathbf{C}\ (\mathbf{B}\ \mathbf{B}\ \mathbf{S}_n)\ \mathbf{K}_n)\end{array}$$

The arity-generic version of $\mathbf{C}$, written as $\mathbf{C}_{\text{AG}}$ takes c_n and returns $\mathbf{C}_n$. We can define the $\mathbf{C}_{\text{AG}}$ as follows:

$$\mathbf{C}_{\text{AG}} \equiv \lambda n.(\mathbf{C}\ (\mathbf{B}\ \mathbf{B}\ (\mathbf{S}_{\text{AG}}\ n))\ (\mathbf{K}_{\text{AG}}\ n))$$

This definition satisfies the requirement that $(\mathbf{C}_{\text{AG}}\ c_n) = \mathbf{C}_n$. Also note that $\mathbf{C}_0 = \mathbf{I}$, and $\mathbf{C}_1 = \mathbf{C}$.

2.6. **Summary and Conclusion.** We have introduced n-ary and arity-generic generalizations of the combinators $\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$. These terms satisfy the property that for any $X \in \{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$, we have $(X_{\text{AG}}\ c_n) = X_n$, and in particular $(X_{\text{AG}}\ c_1) = X_1 = X$.

Encoding an n-ary extension of a λ-term parallels the case where $n = 1$, both in the steps as well as in the final encoding. For example, consider the parallel encoding of $\mathbf{B}$ and $\mathbf{B}_n$:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathbf{B}xyz & = & x(yz) & \mathbf{B}_n xyz_1 \cdots z_n & = & x(yz_1 \cdots z_n) \\
& = & \mathbf{K}xz(yz) & & = & \mathbf{K}_n xz_1 \cdots z_n(yz_1 \cdots z_n) \\
& = & \mathbf{S}(\mathbf{K}x)yz & & = & \mathbf{S}_n(\mathbf{K}_n x)yz_1 \cdots z_n \\
& = & \mathbf{KS}x(\mathbf{K}x)yz & & = & \mathbf{KS}_n x(\mathbf{K}_n x)yz_1 \cdots z_n \\
& = & \underline{\mathbf{S}(\mathbf{KS})\mathbf{K}}xyz & & = & \underline{\underline{\mathbf{S}(\mathbf{KS}_n)\mathbf{K}_n}}xyz_1 \cdots z_n
\end{array}$$

Hence we obtain an alternative encoding for an arity-generic extension of $\mathbf{B}$ as follows:

$$\mathbf{B}_{\text{AG}}^{\text{alt}} \equiv \lambda n.(\mathbf{S}\ (\mathbf{K}\ (\mathbf{S}_{\text{AG}}\ n))\ (\mathbf{K}_{\text{AG}}\ n))$$

Similarly, consider the parallel encoding of $\mathbf{C}$ and $\mathbf{C}_n$:

$$\begin{array}{rcl@{\qquad}rcl}
\mathbf{C}xyz & = & xzy & \mathbf{C}_n xyz_1 \cdots z_n & = & xz_1 \cdots z_n y \\
& = & xz(\mathbf{K}yz) & & = & xz_1 \cdots z_n(\mathbf{K}_n yz_1 \cdots z_n) \\
& = & \mathbf{S}x(\mathbf{K}y)z & & = & \mathbf{S}_n x(\mathbf{K}_n y)z_1 \cdots z_n \\
& = & \mathbf{K}(\mathbf{S}x)y(\mathbf{K}y)z & & = & \mathbf{K}(\mathbf{S}_n x)y(\mathbf{K}_n y)z_1 \cdots z_n \\
& = & \mathbf{S}(\mathbf{K}(\mathbf{S}x))\mathbf{K}yz & & = & \mathbf{S}(\mathbf{K}(\mathbf{S}_n x))\mathbf{K}_n yz_1 \cdots z_n \\
& = & \mathbf{S}(\mathbf{KK}x(\mathbf{S}x))\mathbf{K}yz & & = & \mathbf{S}(\mathbf{KK}x(\mathbf{S}_n x))\mathbf{K}_n yz_1 \cdots z_n \\
& = & \mathbf{KS}x(\mathbf{S}(\mathbf{KK})\mathbf{S}x)\mathbf{K}yz & & = & \mathbf{KS}x(\mathbf{S}(\mathbf{KK})\mathbf{S}_n x)\mathbf{K}_n yz_1 \cdots z_n \\
& = & \mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S})x\mathbf{K}yz & & = & \mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S}_n)x\mathbf{K}_n yz_1 \cdots z_n \\
& = & \mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S})x(\mathbf{KK}x)yz & & = & \mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S}_n)x(\mathbf{KK}_n x)yz_1 \cdots z_n \\
& = & \underline{\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S}))(\mathbf{KK})}xyz & & = & \underline{\underline{\mathbf{S}(\mathbf{S}(\mathbf{KS})(\mathbf{S}(\mathbf{KK})\mathbf{S}_n))(\mathbf{KK}_n)}}xyz_1 \cdots z_n
\end{array}$$

Hence we obtain an alternative encoding for an arity-generic extension of $\mathbf{C}$ as follows:

$$\mathbf{C}_{\text{AG}}^{\text{alt}} \equiv \lambda n.(\mathbf{S}\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{S})\ (\mathbf{S}\ (\mathbf{K}\ \mathbf{K})(\mathbf{S}_{\text{AG}}\ n)))\ (\mathbf{K}\ (\mathbf{K}_{\text{AG}}\ n)))$$

Arity-generic λ-terms can be encoded directly using $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}, \mathbf{K}_{\text{AG}}, \mathbf{S}_{\text{AG}}\}$ and Church numerals, similarly to how combinators are encoded using $\{\mathbf{K}, \mathbf{S}\}$, and we have done just that in encoding the $\mathbf{B}_{\text{AG}}, \mathbf{C}_{\text{AG}}$ combinators.

Our aim, however, was to extend the original $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ basis introduced by Schönfinkel, resulting in a more compact encoding, and in a smaller number of derivation steps.

3. Turner's Bracket-Abstraction Algorithm

A bracket-abstraction algorithm is an algorithm for translating a λ-expression into an equivalent expression that is generated by some basis, an expression that contains no λ-abstractions and no variables, and that is written using applications of the terms of the given basis. Thus a bracket-abstraction algorithm is specific to a given basis.

Turner's bracket-abstraction algorithm [28] is an algorithm for translating λ-expressions into the $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ basis. The algorithm, denoted by *double brackets* ($[\![\cdot]\!]$) is defined on the structure of the argument, and is described in several cases:

Original term	**Condition**	**Rewrite**
$M \in \mathsf{Vars}$		M
$M \in \{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$		M
$M = (P\ Q)$		$([\![P]\!]\ [\![Q]\!])$
$M = \lambda x.\lambda y.P$		$[\![\lambda x.\,[\![\lambda y.P]\!]]\!]$
$M = \lambda x.(P\ x)$	$x \notin \mathsf{FreeVars}(P)$	$[\![P]\!]$
$M = \lambda x.P$	$x \notin \mathsf{FreeVars}(P)$	$(\mathbf{K}\ [\![P]\!])$
$M = \lambda x.(P_x\ Q)$	$x \in \mathsf{FreeVars}(P_x), x \notin \mathsf{FreeVars}(Q)$	$(\mathbf{C}\ [\![\lambda x.P_x]\!]\ [\![Q]\!])$
$M = \lambda x.(P\ Q_x)$	$x \notin \mathsf{FreeVars}(P), x \in \mathsf{FreeVars}(Q_x)$	$(\mathbf{B}\ [\![P]\!]\ [\![\lambda x.Q_x]\!])$
$M = \lambda x.(P_x\ Q_x)$	$x \in \mathsf{FreeVars}(P_x), x \in \mathsf{FreeVars}(Q_x)$	$(\mathbf{S}\ [\![\lambda x.P_x]\!]\ [\![\lambda x.Q_x]\!])$

To give some intuition as to the rôle the different combinators of the basis play in the algorithm, let us analyze just one single case: Where $M = \lambda x.(P_x\ Q_x)$:

$$\lambda x.(P_x\ Q_x) \quad = \quad (\underbrace{\boxed{(\lambda pqx.(p\ x\ (q\ x)))}}_{\equiv \mathbf{S}}\ (\lambda x.P_x)\ (\lambda x.Q_x))$$

Hence we have the rule that

$$[\![\lambda x.(P_x\ Q_x)]\!] \quad = \quad (\mathbf{S}\ [\![\lambda x.P_x]\!]\ [\![\lambda x.Q_x]\!])$$

The correctness of this algorithm is shown by induction on the *length* of the term, rather than by structural induction, because, for example, while $|\lambda x.P_x| < |\lambda x.(P_x\ Q_x)|$, clearly $\lambda x.P_x$ is not a sub-expression of $\lambda x.(P_x\ Q_x)$, and the same holds for other cases in the proof.

Example: We demonstrate the bracket-abstraction algorithm by applying it to S^+:

$$\begin{aligned}
[\![S^+]\!] &\equiv [\![\lambda abc.(b\ (a\ b\ c))]\!] \\
&= [\![\lambda a.\,[\![\lambda b.\,[\![\lambda c.(b\ (a\ b\ c))]\!]]\!]]\!] \\
&= [\![\lambda a.\,[\![\lambda b.(\mathbf{B}\ [\![b]\!]\ [\![\lambda c.(a\ b\ c)]\!])]\!]]\!] \\
&=_\eta [\![\lambda a.\,[\![\lambda b.(\mathbf{B}\ b\ (a\ b))]\!]]\!] \\
&= [\![\lambda a.(\mathbf{S}\ [\![\lambda b.(\mathbf{B}\ b)]\!]\ [\![\lambda b.(a\ b)]\!])]\!] \\
&=_\eta [\![\lambda a.(\mathbf{S}\ \mathbf{B}\ a)]\!] \\
&=_\eta (\mathbf{S}\ \mathbf{B})
\end{aligned}$$

4. Extending Turner's Bracket-Abstraction Algorithm

4.1. **Extending the rule for I.** In Turner's original bracket-abstraction algorithm, the rule for $\mathbf{I}$ was a base case:

$$[\![\lambda x.x]\!] \quad = \quad \mathbf{I}$$

The n-ary and arity-generic generalization of the rule for $\mathbf{I}$ abstracts n variables $x_1, \ldots, x_n$, and is also a base case:

$$\begin{aligned} [\![\lambda x_1 \cdots x_n.(x_1 \cdots x_n)]\!] &= \mathbf{I}_n \\ &= (\mathbf{I}_{\text{AG}}\ c_n) \end{aligned}$$

4.2. **Extending the rule for K.** Note that $(\mathbf{K}\ P) = ((\lambda px.p)\ P) = \lambda x.P$. Accordingly, $\mathbf{K}$ is used in the original bracket-abstraction algorithm to abstract a variable x over an expression P, where $x \notin \mathsf{FreeVars}(P)$:

$$[\![\lambda x.P]\!] \quad = \quad (\mathbf{K}\ [\![P]\!])$$

The n-ary and arity-generic generalization of the rule for $\mathbf{K}$ allows for abstracting n variables $x_1, \ldots, x_n$ over an expression P, where $\{x_1, \ldots, x_n\} \cap \mathsf{FreeVars}(P) = \emptyset$:

$$\begin{aligned} [\![\lambda x_1 \cdots x_n.P]\!] &= (\mathbf{K}_n\ [\![P]\!]) \\ &= (\mathbf{K}_{\text{AG}}\ c_n\ [\![P]\!]) \end{aligned}$$

4.3. **Extending the rule for B.** Note that

$$\begin{aligned} (\mathbf{B}\ P\ (\lambda x.Q_x)) &= ((\lambda pqx.(p\ (q\ x)))\ P\ (\lambda x.Q_x)) \\ &= \lambda x.(P\ Q_x) \end{aligned}$$

where $x \in \mathsf{FreeVars}(Q_x)$. Accordingly, $\mathbf{B}$ is used in the original bracket-abstraction algorithm to abstract a variable x over an application $(P\ Q_x)$, where $x \in \mathsf{FreeVars}(Q_x)$:

$$[\![\lambda x.(P\ Q_x)]\!] \quad = \quad (\mathbf{B}\ [\![P]\!]\ [\![\lambda x.Q_x]\!])$$

The n-ary and arity-generic generalization of the rule for $\mathbf{B}$ allows for abstracting n variables $x_1, \ldots, x_n$ over an application $(P\ Q_{x_1,\ldots,x_n})$, where $\{x_1, \ldots, x_n\} \cap \mathsf{FreeVars}(P) = \emptyset$ and $\{x_1, \ldots, x_n\} \subseteq \mathsf{FreeVars}(Q_{x_1,\ldots,x_n})$:

$$\begin{aligned} [\![\lambda x_1 \cdots x_n.(P\ Q_{x_1,\ldots,x_n})]\!] &= (\mathbf{B}_n\ [\![P]\!]\ [\![\lambda x_1 \cdots x_n.Q_{x_1,\ldots,x_n}]\!]) \\ &= (\mathbf{B}_{\text{AG}}\ c_n\ [\![P]\!]\ [\![\lambda x_1 \cdots x_n.Q_{x_1,\ldots,x_n}]\!]) \end{aligned}$$

4.4. **Extending the rule for C.** Note that

$$\begin{aligned}(\mathbf{C}\ (\lambda x.P_x)\ Q) &= ((\lambda pqx.(p\ x\ q))\ (\lambda x.P_x)\ Q)\\ &= \lambda x.(P_x\ Q)\end{aligned}$$

where $x \in \mathsf{FreeVars}(P_x)$. Accordingly, $\mathbf{C}$ is used in the original bracket-abstraction algorithm to abstract a variable x over an application $(P_x\ Q)$, where $x \in \mathsf{FreeVars}(P_x)$:

$$[\![\lambda x.(P_x\ Q)]\!] = (\mathbf{C}\ [\![\lambda x.P_x]\!]\ [\![Q]\!])$$

The n-ary and arity-generic generalization of the rule for $\mathbf{C}$ allows for abstracting n variables $x_1, \ldots, x_n$ over an application $(P_{x_1,\ldots,x_n}\ Q)$, where $\{x_1, \ldots, x_n\} \subseteq \mathsf{FreeVars}(P_{x_1,\ldots,x_n}) \wedge \{x_1, \ldots, x_n\} \cap \mathsf{FreeVars}(Q) = \emptyset$:

$$\begin{aligned}[\![\lambda x_1 \cdots x_n.(P_{x_1,\ldots,x_n}\ Q)]\!] &= (\mathbf{C}_n\ [\![\lambda x_1 \cdots x_n.P_{x_1,\ldots,x_n}]\!]\ [\![Q]\!])\\ &= (\mathbf{C}_{\mathrm{AG}}\ c_n\ [\![\lambda x_1 \cdots x_n.P_{x_1,\ldots,x_n}]\!]\ [\![Q]\!])\end{aligned}$$

4.5. **Extending the rule for S.** Note that

$$\begin{aligned}(\mathbf{S}\ (\lambda x.P_x)\ (\lambda x.Q_x)) &= ((\lambda pqx.(p\ x\ (q\ x)))\ (\lambda x.P_x)\ (\lambda x.Q_x))\\ &= \lambda x.(P_x\ Q_x)\end{aligned}$$

where $x \in \mathsf{FreeVars}(P_x) \cap \mathsf{FreeVars}(Q_x)$. Accordingly, $\mathbf{S}$ is used in the original bracket-abstraction algorithm to abstract a variable x over an application $(P_x\ Q_x)$, where $x \in \mathsf{FreeVars}(P_x) \cap \mathsf{FreeVars}(Q_x)$:

$$[\![\lambda x.(P_x\ Q_x)]\!] = (\mathbf{S}\ [\![\lambda x.P_x]\!]\ [\![\lambda x.Q_x]\!])$$

The n-ary and arity-generic generalization of the rule for $\mathbf{S}$ allows for abstracting n variables $x_1, \ldots, x_n$ over an application $(P_{x_1,\ldots,x_n}\ Q_{x_1,\ldots,x_n})$, where $\{x_1, \ldots, x_n\} \subseteq \mathsf{FreeVars}(P_{x_1,\ldots,x_n}) \cap \mathsf{FreeVars}(Q_{x_1,\ldots,x_n})$:

$$\begin{aligned}[\![\lambda x_1 \cdots x_n.(P_{x_1,\ldots,x_n}\ Q_{x_1,\ldots,x_n})]\!] &= (\mathbf{S}_n\ [\![\lambda x_1 \cdots x_n.P_{x_1,\ldots,x_n}]\!]\\ &\qquad [\![\lambda x_1 \cdots x_n.Q_{x_1,\ldots,x_n}]\!])\\ &= (\mathbf{S}_{\mathrm{AG}}\ c_n\ [\![\lambda x_1 \cdots x_n.P_{x_1,\ldots,x_n}]\!]\\ &\qquad [\![\lambda x_1 \cdots x_n.Q_{x_1,\ldots,x_n}]\!])\end{aligned}$$

4.6. **Summary and Conclusion.** In Turner's bracket-abstraction algorithm, each of the combinators $\mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$ is used to encode an abstraction of a variable over an expression: The $\mathbf{K}$ combinator is used when the variable does not occur freely in the expression. The $\mathbf{B}, \mathbf{C}, \mathbf{S}$ combinators are used when the variable abstracts over an application of two expressions, and correspond to the situations where the given variable occurs freely in one or in both expressions.

We extended Turner's bracket-abstraction algorithm by introducing four additional rules for $\mathbf{I}_{\mathrm{AG}}, \mathbf{K}_{\mathrm{AG}}, \mathbf{B}_{\mathrm{AG}}, \mathbf{C}_{\mathrm{AG}}, \mathbf{S}_{\mathrm{AG}}$, corresponding to the abstraction of a *sequence of variables* of an expression. The extended algorithm shares the simplicity of Turner's original algorithm, and generates compact encodings for arity-generic λ-terms.

In those situations where $\{x_1, \ldots, x_n\} \cap \mathsf{FreeVars}(P) \neq \emptyset \wedge \{x_1, \ldots, x_n\} \not\subseteq \mathsf{FreeVars}(P)$, we can use the **K**-introduction rule to obtain from P a β-equal expression P' for which $\{x_1, \ldots, x_n\} \subseteq \mathsf{FreeVars}(P')$.

Proposition 4.1. *For any n-ary λ-expression $\mathcal{E}_n$ that is written with ellipses, a corresponding arity-generic λ-expression $\mathcal{E}_{\text{AG}}$ can be defined, such that for any natural number n, we have $(\mathcal{E}_{\text{AG}}\ c_n) = \mathcal{E}_n$.*

Sketch of proof: By induction on the length of $\mathcal{E}_n$, a corresponding rule can be applied in the extended algorithm, so that the rewritten expression is arity-generic and satisfies the above relation to $\mathcal{E}_n$. □

Example: Church [7] introduces the λ-expression $\mathbf{D} = \lambda x.(x\ x)$, which is encoded via Turner's algorithm as $(\mathbf{S\ I\ I})$. How would the n-ary and arity-generic extensions be encoded?

The n-ary extension:

$$\begin{aligned} [\![\mathbf{D}_n]\!] &= [\![\lambda x_1 \cdots x_n.(x_1 \cdots x_n\ (x_1 \cdots x_n))]\!] \\ &= (\mathbf{S}_n\ [\![\lambda x_1 \cdots x_n.(x_1 \cdots x_n)]\!]\ [\![\lambda x_1 \cdots x_n.(x_1 \cdots x_n)]\!]) \\ &= (\mathbf{S}_n\ \mathbf{I}_n\ \mathbf{I}_n) \end{aligned}$$

The arity-generic extension:

$$\mathbf{D}_{\text{AG}} = \lambda n.(\mathbf{S}_{\text{AG}}\ n\ (\mathbf{I}_{\text{AG}}\ n)\ (\mathbf{I}_{\text{AG}}\ n))$$

So as we can see, the extended basis $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}, \mathbf{I}_{\text{AG}}, \mathbf{K}_{\text{AG}}, \mathbf{B}_{\text{AG}}, \mathbf{C}_{\text{AG}}, \mathbf{S}_{\text{AG}}\}$ provides a natural extension of the original basis for encoding arity-generic λ-expressions.

5. n-ARY AND ARITY-GENERIC EXPRESSIONS

5.1. **The arity-generic selector combinators.** The *selector* combinators return one of their arguments. For n, k such that $0 \leq k \leq n$, the selector that returns the k-th of its $n+1$ arguments is defined as follows:

$$\sigma_k^n \equiv \lambda x_0 \cdots x_n.x_k$$

An arity-generic version of the selector, which we write as σ_{AG}, would take Church numerals k, n and return σ_k^n. We generate σ_k^n in two states: First, we generate a selector in which only the first argument is returned:

$$\lambda x_k \cdots x_n.x_k$$

We then tag on k additional abstractions.

Suppose we have P, Q that are defined as follows:

$$\begin{aligned} P &= \lambda x_0 \cdots x_n.x_0 \\ Q &= \lambda x_0 \cdots x_n x_{n+1}.x_0 \end{aligned}$$

We define the λ-term f to map $P \overset{f}{\Longrightarrow} Q$ for all n. The relationship between P and Q is given by $Q = \lambda x.(P\ (\lambda z.x))$, and so:

$$f = \lambda px.(p\ (\lambda z.x))$$

We use f to generate $\lambda x_k \cdots x_n.x_k$ by applying the $(n-k)$-th composition of f to the identity combinator $\mathbf{I}$. From this we obtain σ_k^n by $k+1$ applications of $\mathbf{K}$. We can now define σ_{AG} as follows:

$$\sigma_{\text{AG}} \equiv \lambda kn.(P^- \ k \ \mathbf{K} \ (\dot{-} \ n \ k \ f \ \mathbf{I}))$$

This definition satisfies the requirement that $(\sigma_{\text{AG}} \ c_k \ c_n) = \sigma_k^n$.

5.2. **The arity-generic projections.** The *projection* combinators take an n-tuple and return the respective projection:

$$(\pi_k^n \ \langle x_1, \ldots, x_n \rangle) = x_k$$

The standard way of defining projections is to take an n-tuple and apply it to the corresponding selector:

$$\pi_k^n \equiv \lambda x.(x \ \sigma_k^n)$$

The definition of the arity-generic extension π_{AG} can be written in terms of σ_{AG}:

$$\pi_{\text{AG}} \equiv \lambda knx.(x \ (\sigma_{\text{AG}} \ k \ n))$$

This definition satisfies the requirement that $(\pi_{\text{AG}} \ c_k \ c_n) = \pi_k^n$.

5.3. **The arity-generic, ordered n-tuple maker.** In his textbook *The Lambda Calculus: Its Syntax and Semantics* [4, pages 133-134], Barendregt introduces one of the standard constructions for n-tuples[2]:

$$\langle E_1, \ldots, E_n \rangle \equiv \lambda z.(z \ E_1 \cdots E_n)$$

The ordered n-tuple maker $\langle \_, \ldots, \_ \rangle_n$ takes n λ-terms and returns their ordered tuple. Although most texts on the λ-calculus use it implicitly by using ordered tuples as generalizations to the syntax of the λ-calculus, it is easily definable:

$$\begin{aligned} \langle \_, \ldots, \_ \rangle_n &\equiv \lambda x_1 \cdots x_n. \langle x_1, \ldots, x_n \rangle \\ &= \lambda x_1 \cdots x_n z.(z \ x_1 \ \cdots x_n) \end{aligned}$$

We wish to define the arity-generic generalization of the n-tuple maker $\langle \_, \ldots, \_ \rangle_{\text{AG}}$, such that:

$$(\langle \_, \ldots, \_ \rangle_{\text{AG}} \ c_n) = \langle \_, \ldots, \_ \rangle_n$$

We relate $\langle \_, \ldots, \_ \rangle_n$ with $\langle \_, \ldots, \_ \rangle_{n+1}$ as follows:

$$\begin{aligned} \langle \_, \ldots, \_ \rangle_{n+1} &= \lambda x_1 \cdots x_n x_{n+1} z.(z \ x_1 \cdots x_n \ x_{n+1}) \\ &= \lambda x_1 \cdots x_n x_{n+1} z.(\langle \_, \ldots, \_ \rangle_n \ x_1 \cdots x_n \ z \ x_{n+1}) \\ &= \lambda x_1 \cdots x_n x_{n+1} z.(\mathbf{C} \ (\langle \_, \ldots, \_ \rangle_n \ x_1 \cdots x_n) \ x_{n+1} \ z) \\ &=_\eta \lambda x_1 \cdots x_n.(\mathbf{C} \ (\langle \_, \ldots, \_ \rangle_n \ x_1 \cdots x_n)) \\ &= \lambda x_1 \cdots x_n.(\mathbf{B}_{\text{AG}} \ \mathbf{C} \ \langle \_, \ldots, \_ \rangle_n \ x_1 \cdots x_n) \\ &=_\eta (\mathbf{B}_{\text{AG}} \ \mathbf{C} \ \langle \_, \ldots, \_ \rangle_n) \end{aligned}$$

[2]This construction appears, for ordered pairs and triples, in Church's book *The Calculi of Lambda Conversion* [7].

Using this relation, we define the λ-term f to map $\langle c_n, \langle \_, \ldots, \_\rangle_n\rangle \overset{f}{\Longrightarrow} \left\langle c_{n+1}, \langle \_, \ldots, \_\rangle_{n+1}\right\rangle$ for all n:

$$\begin{array}{rcl} f & = & \lambda p.\langle (S^+ \ (\pi_1^2 \ p)), \\ & & \quad (\mathbf{B}_{\text{AG}} \ (\pi_1^2 \ p) \ \mathbf{C} \ (\pi_2^2 \ p))\rangle \end{array}$$

Notice that $\langle \_, \ldots, \_\rangle_0 = \mathbf{I}$, so we can obtain $\langle \_, \ldots, \_\rangle_n$ by applying the n-th composition of f to $\langle c_0, \mathbf{I}\rangle$. We define $\langle \_, \ldots, \_\rangle_{\text{AG}}$ as follows:

$$\langle \_, \ldots, \_\rangle_{\text{AG}} \ \equiv \ \lambda n.(\pi_2^2 \ (n \ f \ \ \langle c_0, \mathbf{I}\rangle))$$

This definition satisfies the requirement that $(\langle \_, \ldots, \_\rangle_{\text{AG}} \ c_n) = \langle \_, \ldots, \_\rangle_n$, so for example, $(\langle \_, \ldots, \_\rangle_{\text{AG}} \ c_3 \ E_1 \ E_2 \ E_3) = \langle E_1, E_2, E_3\rangle$.

The task of defining the $\langle \_, \ldots, \_\rangle_n$ combinator is given as an exercise in the author's course notes on the λ-calculus [14], where the combinator is referred to as *malloc*, in a tongue-in-cheek reference to the C library function for allocating blocks of memory.

5.4. **Applying λ-terms.** A useful property of our representation of ordered n-tuples, is that it gives us *left-associated* applications immediately:

$$(\langle E_1, \ldots, E_n\rangle \ \ P) \ = \ (P \ E_1 \cdots E_n)$$

This behavior can be used to apply some expression to its arguments, where these arguments are passed in an n-tuple, in much the same way as the `apply` procedure in LISP [21], which takes a procedure and a list of arguments, and applies the procedure to these arguments. One notable difference though, is that the λ-calculus does not have a notion of arity-generic procedures, and so the procedure we wish to apply must "know" now many arguments to expect. We can thus define:

$$\text{Apply} \ \equiv \ \lambda f v.(v \ f)$$

Because functions in the λ-calculus are Curried, and therefore applications associate to the left, Apply combinator proides for *left-associated* applications. For *right-associated* applications, we would like to have an arity-generic version of the following n-ary λ-term:

$$\text{RightApplicator}_n \ \equiv \ \lambda x_1 \cdots x_n z.(x_1 \ (x_2 \cdots (x_n \ z) \cdots))$$

We begin by writing $\text{RightApplicator}_{n+1}$ in terms of RightApplicator_n:

$$\begin{array}{rcl} \text{RightApplicator}_{n+1} & = & \lambda x_1 \cdots x_n x_{n+1} z.(\text{RightApplicator}_n \ x_1 \cdots x_n \ (x_{n+1} \ z)) \\ & = & \lambda x_1 \cdots x_n x_{n+1} z.(\mathbf{B} \ (\text{RightApplicator}_n \ x_1 \cdots x_n) \ x_{n+1} \ z) \\ & = & \lambda x_1 \cdots x_n x_{n+1} z.(\mathbf{B}_n \ \mathbf{B} \ \text{RightApplicator}_n \ x_1 \cdots x_n \ x_{n+1} \ z) \\ & =_\eta & (\mathbf{B}_n \ \mathbf{B} \ \text{RightApplicator}_n) \end{array}$$

The λ-term f such that $\langle c_n, \text{RightApplicator}_n\rangle \overset{f}{\Longrightarrow} \left\langle c_{n+1}, \text{RightApplicator}_{n+1}\right\rangle$ is given by:

$$\begin{array}{rcl} f & \equiv & \lambda p.\langle (S^+ \ (\pi_1^2 \ p)), \\ & & \quad (\mathbf{B}_{\text{AG}} \ (\pi_1^2 \ p) \ \mathbf{B} \ (\pi_2^2 \ p))\rangle \end{array}$$

Notice that $\text{RightApplicator}_0 = \lambda z.z = \mathbf{I}$, so we can obtain RightApplicator_n by applying the n-th composition of f to $\langle c_0, \mathbf{I}\rangle$. We define $\text{RightApplicator}_{\text{AG}}$ as follows:

$$\text{RightApplicator}_{\text{AG}} \ \equiv \ \lambda n.(\pi_2^2 \ (n \ f \ \ \langle c_0, \mathbf{I}\rangle))$$

This definition satisfies the requirement that $(\text{RightApplicator}_{\text{AG}}\ c_n) = \text{RightApplicator}_n$.

5.5. **Extending n-tuples.** Applying $\langle \_, \ldots, \_ \rangle_{n+1}$ to n arguments results in a λ-term that takes an argument and returns an $n+1$-tuple, in which the given argument is the $n+1$-st projection. We use this fact to extend an n-tuple by an additional $n+1$-st element:

$$\text{Extend}_{\text{AG}} \equiv \lambda nva.(v\ (\langle \_, \ldots, \_ \rangle_{\text{AG}}\ (S^+\ n))\ a)$$

We can use it as follows:

$$(\text{Extend}_{\text{AG}}\ c_n\ \ \langle x_1, \ldots, x_n \rangle\ \ x_{n+1}) = \langle x_1, \ldots, x_n, x_{n+1} \rangle$$

Similarly, we can define the λ-term Catenate, for creating an $n{+}k$-tuple given an n-tuple and a k-tuple:

$$\text{Catenate} \equiv \lambda nvkw.(w\ (v\ (\langle \_, \ldots, \_ \rangle_{\text{AG}}\ (+\ n\ k))))$$

For example:

$$(\text{Catenate}\ c_3\ \ \langle E_1, E_2, E_3 \rangle\ \ c_2\ \ \langle F_1, F_2 \rangle) = \langle E_1, E_2, E_3, F_1, F_2 \rangle$$

5.6. **Iota.** When working with indexed expressions, it is convenient to have the *iota*-function (written as the Greek letter ι, and pronounced "yota"), which maps the number N to the vector $\langle 0, \ldots, N-1 \rangle$. Iota was introduced by Kenneth Iverson first in the APL notation [15], and then in the APL programming language [22].

We implement the ι combinator to take a Church numeral c_n and return the ordered n-tuple $\langle c_0, \ldots, c_{n-1} \rangle$. Given the standard definition of ordered n-tuples, it is natural to define $(\iota\ c_0) = \mathbf{I}$.

We know that

$$\begin{aligned} (\iota\ c_n) &= \langle c_0, \ldots, c_{n-1} \rangle \\ &= \lambda z.(z\ c_0 \cdots c_{n-1}\ c_n) \end{aligned}$$

So the λ-term f such that

$$(\iota\ c_n) \stackrel{(f\ c_n)}{\Longrightarrow} (\iota\ c_{n+1})$$

can be characterized as follows:

$$(\lambda z.(z\ c_0 \cdots c_{n-1})) \stackrel{(f\ c_n)}{\Longrightarrow} (\lambda z.(z\ c_0 \cdots c_{n-1}\ c_n))$$

We define f as follows:

$$f = \lambda niz.(i\ z\ n)$$

The λ-term g such that

$$\langle n, i \rangle \stackrel{g}{\Longrightarrow} \big\langle (S^+\ n), (f\ n\ i) \big\rangle$$

is defined as follows:

$$\begin{aligned} g = \lambda p.\langle &(S^+\ (\pi_1^2\ p)), \\ &(f\ (\pi_1^2\ p)\ (\pi_2^2\ p)) \rangle \end{aligned}$$

We now define ι as follows:

$$\iota \quad \equiv \quad \lambda n.(\pi_2^2 \ (n \ g \ \langle c_0, \mathbf{I}\rangle))$$

This definition satisfies the requirement that $(\iota \ c_{n+1}) = \lambda z.(z \ c_0 \cdots c_n)$.

5.7. **Reversing.** It is often useful to be able to reverse the arguments to a function or an n-tuple. We can define an n-ary reversal combinator R_n as follows:

$$R_n \quad \equiv \quad \lambda x_1 \cdots x_n w.(w \ x_n \cdots x_1)$$

R_n can be used in two ways:

(1) We can use it to reverse an ordered n tuple:

$$(\langle E_1, \ldots, E_n\rangle \ R_n) \quad = \quad \langle E_n, \ldots, E_1\rangle$$

(2) We can use it to take n arguments are return their n-tuple, in reverse order:

$$(R_n \ E_1 \cdots E_n) \quad = \quad \langle E_n, \ldots, E_1\rangle$$

We would like to define R_{AG}, the arity-generic generalization of R_n, such that $(R_{\text{AG}} \ c_n) = R_n$. We start by writing R_{n+1} in terms of R_n:

$$\begin{array}{rcl}
R_{n+1} & = & \lambda x_1 \cdots x_n x_{n+1} w.(w \ x_1 \cdots x_{n+1}) \\
& = & \lambda x_1 \cdots x_n x_{n+1} w.(R_n \ x_1 \cdots x_n \ (w \ x_{n+1})) \\
& = & \lambda x_1 \cdots x_n x_{n+1} w.(\mathbf{B} \ (R_n \ x_1 \cdots x_n) \ w \ x_{n+1}) \\
& = & \lambda x_1 \cdots x_n x_{n+1} w.(\mathbf{C} \ (\mathbf{B} \ (R_n \ x_1 \cdots x_n)) \ x_{n+1} \ w) \\
& =_\eta & \lambda x_1 \cdots x_n.(\mathbf{C} \ (\mathbf{B} \ (R_n \ x_1 \cdots x_n))) \\
& = & \lambda x_1 \cdots x_n.(\mathbf{C} \ (\mathbf{B}_n \ \mathbf{B} \ R_n \ x_1 \cdots x_n)) \\
& = & \lambda x_1 \cdots x_n.(\mathbf{B}_n \ \mathbf{C} \ (\mathbf{B}_n \ \mathbf{B} \ R_n) \ x_1 \cdots x_n) \\
& =_\eta & (\mathbf{B}_n \ \mathbf{C} \ (\mathbf{B}_n \ \mathbf{B} \ R_n))
\end{array}$$

The λ-term f that takes a Church numeral c_n, and maps $R_n \overset{(f \ c_n)}{\Longrightarrow} R_{n+1}$, is given by:

$$f \quad \equiv \quad \lambda nr.(\mathbf{B}_{\text{AG}} \ n \ \mathbf{C} \ (\mathbf{B}_{\text{AG}} \ n \ \mathbf{B} \ r))$$

The λ-term g such that $\langle c_n, R_n\rangle \overset{g}{\Longrightarrow} \langle c_{n+1}, R_{n+1}\rangle$ is given by:

$$g \quad \equiv \quad \lambda p.\left\langle (S^+ \ (\pi_1^2 \ p)), (f \ (\pi_1^2 \ p) \ (\pi_2^2 \ p))\right\rangle$$

Notice that $R_0 = \mathbf{I}$, so we can obtain R_n by applying the n-th composition of g to $\langle c_0, \mathbf{I}\rangle$. We define R_{AG} as follows:

$$R_{\text{AG}} \quad \equiv \quad \lambda n.(\pi_2^2 \ (n \ g \ \langle c_0, \mathbf{I}\rangle))$$

Note that $(R_{\text{AG}} \ c_n) = R_n$.

Below are examples of two slightly different ways of using R_{AG}:

$$\begin{array}{rcl}
(R_{\text{AG}} \ c_3 \ E_1 \ E_2 \ E_3) & = & (R_3 \ E_1 \ E_2 \ E_3) \\
& = & \langle E_3, E_2, E_1\rangle \\
(\langle E_1, E_2, E_3, E_4\rangle \ (R_{\text{AG}} \ c_4)) & = & (\langle E_1, E_2, E_3, E_4\rangle \ R_4) \\
& = & \langle E_4, E_3, E_2, E_1\rangle
\end{array}$$

5.8. **Mapping.** We would like to define the combinator $\mathrm{Map}_{\mathrm{AG}}$, such that:

$$(\mathrm{Map}_{\mathrm{AG}}\ c_n\ f\ \langle x_1, \ldots, x_n\rangle) \quad = \quad \langle (f\ x_1), \ldots (f\ x_n)\rangle$$

Let:

$$\begin{array}{rcl} Q_n & \equiv & \lambda x_1 \cdots x_n z.(z\ (f\ x_1) \cdots (f\ x_n)) \\ \mathrm{Map}_n & \equiv & \lambda f v.(v\ Q_n) \end{array}$$

We define Q_{n+1} in terms of Q_n:

$$\begin{array}{rcl} Q_{n+1} & = & \lambda x_1 \cdots x_n w z.(Q_n\ x_1\ \cdots x_n\ z\ (f\ w)) \\ & = & \lambda x_1 \cdots x_n w z.(\mathbf{C}\ (Q_n\ x_1 \cdots x_n)\ (f\ w)\ z) \\ & =_\eta & \lambda x_1 \cdots x_n w.(\mathbf{C}\ (Q_n\ x_1 \cdots x_n)\ (f\ w)) \\ & = & \lambda x_1 \cdots x_n w.(\mathbf{B}\ (\mathbf{C}\ (Q_n\ x_1 \cdots x_n))\ f\ w) \\ & =_\eta & \lambda x_1 \cdots x_n.(\mathbf{B}\ (\mathbf{C}\ (Q_n\ x_1 \cdots x_n))\ f) \\ & = & \lambda x_1 \cdots x_n.(\mathbf{C}\ \mathbf{B}\ f\ (\mathbf{C}\ (Q_n\ x_1 \cdots x_n))) \\ & = & \lambda x_1 \cdots x_n.(\mathbf{B}\ (\mathbf{C}\ \mathbf{B}\ f)\ \mathbf{C}\ (Q_n\ x_1 \cdots x_n)) \\ & = & \lambda x_1 \cdots x_n.(\mathbf{B}_n\ (\mathbf{B}\ (\mathbf{C}\ \mathbf{B}\ f)\ \mathbf{C})\ Q_n\ x_1 \cdots x_n) \\ & =_\eta & (\mathbf{B}_n\ (\mathbf{B}\ (\mathbf{C}\ \mathbf{B}\ f)\ \mathbf{C})\ Q_n) \end{array}$$

Using this relation, we define the λ-term g to map $\langle c_n, Q_n\rangle \overset{g}{\Longrightarrow} \langle c_{n+1}, Q_{n+1}\rangle$ for all n:

$$\begin{array}{rcl} g & \equiv & \lambda p.\langle (S^+\ (\pi_1^2\ p)), \\ & & \quad (\mathbf{B}_{\mathrm{AG}}\ (\pi_1^2\ p)\ (\mathbf{B}\ (\mathbf{C}\ \mathbf{B}\ f)\ \mathbf{C})\ (\pi_2^2\ p))\rangle \end{array}$$

Notice that $Q_0 = \mathbf{I}$, so we can obtain Q_n by applying the n-th composition of g to $\langle c_0, \mathbf{I}\rangle$. We define Q_{AG} as follows:

$$Q_{\mathrm{AG}} \quad \equiv \quad \lambda n.(\pi_2^2\ (n\ g\ \langle c_0, \mathbf{I}\rangle))$$

We now define $\mathrm{Map}_{\mathrm{AG}}$ as follows:

$$\mathrm{Map}_{\mathrm{AG}} \quad \equiv \quad \lambda n f v.(v\ (Q_{\mathrm{AG}}\ n))$$

This definition satisfies the requirement that $(\mathrm{Map}_{\mathrm{AG}}\ c_n) = \mathrm{Map}_n$.

5.9. **Arity-generic, multiple fixed-point combinators.** By now we have the tools needed to construct arity-generic, multiple fixed-point combinators in the λ-calculus. Fixed-point combinators are used to solve fixed-point equations, resulting in a single solution that is the *least* in a lattice-theoretic sense. When moving to n multiple fixed-point equations, multiple fixed-point combinators are needed to solve the system, giving a set of n solutions, that once again, are the least in the above-mentioned lattice-theoretic sense.

A set of n multiple fixed-point combinators are λ-terms $F_1^n, \ldots, F_n^n$, such that for any n λ-terms $x_1, \ldots, x_n$, and $k = 1, \ldots, n$ we have:

$$(F_j^n\ x_1 \cdots x_n) \quad = \quad (x_j\ (F_1^n\ x_1 \cdots x_n) \cdots (F_n^n\ x_1 \cdots x_n))$$

Brevity is one motivation for the construction of an arity-generic fixed-point combinator. Using ordinary multiple fixed-point combinators, n combinators are needed for *any* choice of n, which means that if we wish to solve several such systems of equations, we need a great many number of multiple fixed-point combinators. In contrast, an arity-generic fixed-point combinator can be used to find *any* multiple fixed-point in a system of *any* size: It takes as arguments two Church numerals c_n, c_k, which specify the size of the system, and the specific multiple fixed-point, and returns the specific multiple fixed-point combinator of interest.

Other reasons for using an arity-generic fixed-point combinator have to do with the size of the multiple fixed-point combinators and their correctness: The size of the n-ary extensions of Curry's and Turing's historical fixed-point combinators is *quadratic* to the number of equations, or $O(n^2)$. Specifying such large terms, be in on paper, in LaTeX, or in a computerized reduction system is unwieldy and prone to errors. An arity-generic fixed-point combinator is surprisingly compact, because the size of the system is specified as an argument.

5.9.1. *An arity-generic generalization of Curry's fixed-point combinator for multiple fixed points.* Recall Curry's single fixed-point combinator:

$$\begin{array}{rcl} \mathrm{Y}_{\textsc{Curry}} & \equiv & \lambda f.((\lambda x.(f\ (x\ x))) \\ & & \quad\quad (\lambda x.(f\ (x\ x)))) \end{array}$$

Generalizing Curry's single fixed-point combinator to n multiple fixed-point equations yields a sequence $\{\Phi_k^n\}_{k=1}^n$ of n multiple fixed-point combinators, where Φ_k^n is defined as follows:

$$\begin{array}{rcl} \Phi_k^n & \equiv & \lambda f_1 \cdots f_n.((\lambda x_1 \cdots x_n.(f_k\ (x_1\ x_1 \cdots x_n) \cdots (x_n\ x_1 \cdots x_n))) \\ & & \quad\quad\quad\quad (\lambda x_1 \cdots x_n.(f_1\ (x_1\ x_1 \cdots x_n) \cdots (x_n\ x_1 \cdots x_n))) \\ & & \quad\quad\quad\quad\quad\quad \vdots \\ & & \quad\quad\quad\quad (\lambda x_1 \cdots x_n.(f_n\ (x_1\ x_1 \cdots x_n) \cdots (x_n\ x_1 \cdots x_n)))) \end{array}$$

Given the system of fixed-point equations $\{(F_k\ x_1 \cdots x_n) = x_k\}_{k=1}^n$, the k-th multiple fixed-point is given by $(\Phi_k^n\ F_1\ \cdots F_n)$.

Our inductive definition (on the syntax of λ-calculus) is sufficiently precise and well-defined that we can construct, for any given $n \in \mathbb{N}$, a set of multiple fixed-point combinators. But if n is a variable, rather than a constant, then this will not do.

Let $v_x = \langle x_1, \ldots, x_n \rangle$. Starting with the inner common sub-expression $\langle (x_k\ x_1 \cdots x_n) \rangle_{k=1}^n$, we note that:

$$\langle (x_k\ x_1 \cdots x_n) \rangle_{k=1}^n \quad = \quad (\mathrm{Map}_{\mathrm{AG}}\ c_n\ (\lambda x_k.(x_k\ v_x))\ v_x)$$

The arity-generic fixed-point combinator Φ_{AG} takes c_n, c_k, and returns Φ_k^n, which is the fixed-point combinator that takes n generating functions, and returns the k-th of n multiple fixed-points:

$$\begin{array}{rcl} (\Phi_{\mathrm{AG}}\ c_k\ c_n) & = & \lambda f_1 \cdots f_n.((\lambda v_f.((\lambda w.(\pi_{\mathrm{AG}}\ c_k\ c_n\ w\ w)) \\ & & \quad\quad\quad\quad (\mathrm{Map}_{\mathrm{AG}}\ c_n \\ & & \quad\quad\quad\quad\quad\quad (\lambda f_j v_x.(\mathrm{Map}_{\mathrm{AG}}\ c_n \\ & & \quad\quad\quad\quad\quad\quad\quad\quad (\lambda x_k.(x_k\ v_x)) \\ & & \quad\quad\quad\quad\quad\quad\quad\quad v_x\ f_j)) \\ & & \quad\quad\quad\quad\quad\quad v_f))) \\ & & \quad\quad (\langle \llcorner, \ldots, \llcorner \rangle_{\mathrm{AG}}\ c_n\ f_1 \cdots f_n) \end{array}$$

$$
\begin{array}{rl}
= & (\mathbf{B}_{\mathrm{AG}}\ c_n \\
& \quad (\lambda v_f.((\lambda w.(\pi_{\mathrm{AG}}\ c_k\ c_n\ w\ w)) \\
& \qquad (\mathrm{Map}_{\mathrm{AG}}\ c_n \\
& \qquad\quad (\lambda f_j v_x.(\mathrm{Map}_{\mathrm{AG}}\ c_n \\
& \qquad\qquad\qquad (\lambda x_k.(x_k\ v_x)) \\
& \qquad\qquad\qquad v_x\ f_j)) \\
& \qquad\quad v_f))) \\
& \quad (\langle \llcorner, \ldots, \llcorner \rangle_{\mathrm{AG}}\ c_n))
\end{array}
$$

Abstracting the variables k, n over c_k, c_n respectively, we define the arity-generic extension of Curry's multiple fixed-point combinator:

$$
\begin{array}{rcl}
\Phi_{\mathrm{AG}} & \equiv & \lambda kn.(\mathbf{B}_{\mathrm{AG}}\ n \\
& & \quad (\lambda v_f.((\lambda w.(\pi_{\mathrm{AG}}\ k\ n\ w\ w)) \\
& & \qquad (\mathrm{Map}_{\mathrm{AG}}\ n \\
& & \qquad\quad (\lambda f_j v_x.(\mathrm{Map}_{\mathrm{AG}}\ n \\
& & \qquad\qquad\qquad (\lambda x_k.(x_k\ v_x)) \\
& & \qquad\qquad\qquad v_x\ f_j)) \\
& & \qquad\quad v_f))) \\
& & \quad (\langle \llcorner, \ldots, \llcorner \rangle_{\mathrm{AG}}\ n))
\end{array}
$$

This definition satisfies the requirement that $(\Phi_{\mathrm{AG}}\ c_k\ c_n) = \Phi_k^n$.

5.9.2. *An arity-generic generalization of Turing's fixed-point combinator for multiple fixed points.* Recall Turing's single fixed-point combinator:

$$
\begin{array}{rcl}
\mathrm{Y}_{\mathrm{TURING}} & \equiv & ((\lambda xf.(f\ (x\ x\ f))) \\
& & \ (\lambda xf.(f\ (x\ x\ f))))
\end{array}
$$

Generalizing Turing's single fixed-point combinator to n multiple fixed-point equations yields a sequence $\{\Psi_k^n\}_{k=1}^n$ of n multiple fixed-point combinators, where Ψ_k^n is defined as follows:

$$
\begin{array}{rcl}
\Psi_k^n & \equiv & ((\lambda x_1 \cdots x_n f_1 \cdots f_n.(f_k\ (x_1\ x_1 \cdots x_n\ f_1 \cdots f_n) \cdots (x_n\ x_1 \cdots x_n\ f_1 \cdots f_n))) \\
& & \ (\lambda x_1 \cdots x_n f_1 \cdots f_n.(f_1\ (x_1\ x_1 \cdots x_n\ f_1 \cdots f_n) \cdots (x_n\ x_1 \cdots x_n\ f_1 \cdots f_n))) \\
& & \ \cdots \\
& & \ (\lambda x_1 \cdots x_n f_1 \cdots f_n.(f_n\ (x_1\ x_1 \cdots x_n\ f_1 \cdots f_n) \cdots (x_n\ x_1 \cdots x_n\ f_1 \cdots f_n))))
\end{array}
$$

Our construction follows similar lines as with the n-ary generalization of $\mathrm{Y}_{\mathrm{CURRY}}$. For a given n, the ordered n-tuples v_x, v_f are defined as follows:

$$
\begin{array}{rcl}
v_x & \equiv & \langle x_1, \ldots, x_n \rangle \\
v_f & \equiv & \langle f_1, \ldots, f_n \rangle
\end{array}
$$

respectively.

As before, we begin by encoding a common sub-expression $\langle (x_k\ x_1 \cdots x_n\ f_1 \cdots f_n) \rangle_{k=1}^n$, as follows:

$$
\langle (x_k\ x_1 \cdots x_n\ f_1 \cdots f_n) \rangle_{k=1}^n \quad = \quad (\mathrm{Map}_{\mathrm{AG}}\ c_n\ (\lambda x_k.(x_k\ v_x\ v_f))\ v_x)
$$

The arity-generic generalization of Turing's multiple fixed-point combinator is given by:

$$
\begin{array}{lcl}
(\Psi_{\mathrm{AG}}\ c_k\ c_n) & = & \lambda f_1 \cdots f_n.((\lambda v_f.((\lambda w.(\pi_{\mathrm{AG}}\ c_k\ c_n\ w\ w\ v_f)) \\
& & \qquad\qquad (\mathrm{Map}_{\mathrm{AG}}\ c_n \\
& & \qquad\qquad\qquad (\lambda j v_x v_f.(\mathrm{Map}_{\mathrm{AG}}\ c_n \\
& & \qquad\qquad\qquad\qquad (\lambda x_k.(x_k\ v_x\ v_f)) \\
& & \qquad\qquad\qquad\qquad v_x \\
& & \qquad\qquad\qquad\qquad (\pi_{\mathrm{AG}}\ (S^+\ j)\ c_n\ v_f))) \\
& & \qquad\qquad\qquad (\iota\ c_n) \\
& & \qquad ((\langle \text{\textvisiblespace}, \ldots, \text{\textvisiblespace} \rangle_{\mathrm{AG}}\ c_n\ f_1 \cdots f_n)) \\
& = & (\mathbf{B}_{\mathrm{AG}}\ c_n \\
& & \quad (\lambda v_f.((\lambda w.(\pi_{\mathrm{AG}}\ c_k\ c_n\ w\ w\ v_f)) \\
& & \qquad (\mathrm{Map}_{\mathrm{AG}}\ c_n \\
& & \qquad\quad (\lambda j v_x v_f.(\mathrm{Map}_{\mathrm{AG}}\ c_n \\
& & \qquad\qquad (\lambda x_k.(x_k\ v_x\ v_f)) \\
& & \qquad\qquad v_x \\
& & \qquad\qquad (\pi_{\mathrm{AG}}\ (S^+\ j)\ c_n\ v_f))) \\
& & \qquad\quad (\iota\ c_n) \\
& & \quad ((\langle \text{\textvisiblespace}, \ldots, \text{\textvisiblespace} \rangle_{\mathrm{AG}}\ c_n))
\end{array}
$$

We define Ψ_{AG} by abstracting c_k, c_n over the above, to get:

$$
\begin{array}{lcl}
\Psi_{\mathrm{AG}} & \equiv & \lambda kn.(\mathbf{B}_{\mathrm{AG}}\ n \\
& & \quad (\lambda v_f.((\lambda w.(\pi_{\mathrm{AG}}\ k\ n\ w\ w\ v_f)) \\
& & \qquad (\mathrm{Map}_{\mathrm{AG}}\ n \\
& & \qquad\quad (\lambda j v_x v_f.(\mathrm{Map}_{\mathrm{AG}}\ n \\
& & \qquad\qquad (\lambda x_k.(x_k\ v_x\ v_f)) \\
& & \qquad\qquad v_x \\
& & \qquad\qquad (\pi_{\mathrm{AG}}\ (S^+\ j)\ n\ v_f))) \\
& & \qquad\quad (\iota\ n) \\
& & \quad ((\langle \text{\textvisiblespace}, \ldots, \text{\textvisiblespace} \rangle_{\mathrm{AG}}\ n))
\end{array}
$$

This definition satisfies the requirement that $(\Psi_{\mathrm{AG}}\ c_k\ c_n) = \Psi_k^n$.

5.9.3. *An arity-generic generalization of Böhm's construction.* In Sections 5.9.1 and 5.9.2 we introduced n-ary generalizations of Curry's and Turing's fixed-point combinator for solving systems of multiple fixed-point equations. The goal of this section is to show that these generalizations are, in a precise sense, natural, and obey a well-known relation that holds between the two original, single fixed-point combinators.

In his textbook *The Lambda Calculus: Its Syntax and Semantics* [4, page 143], Barendregt mentions, in the proof of Proposition 6.5.5, a result due to Böhm, that relates Curry's and Turing's fixed-point combinators:

Let $M \equiv \lambda \phi x.(x\ (\phi\ x)) = (\mathbf{S}\ \mathbf{I})$. We have:

$$(\mathrm{Y}_{\textsc{Curry}}\ M) \ \twoheadrightarrow\ \mathrm{Y}_{\textsc{Turing}}$$

To understand whence this λ-term M comes, consider the definition of a fixed-point combinator: A term Φ, such that for all x, $(\Phi\ x)$ is a fixed point of x, and so we have:

$$(\Phi\ x) \quad = \quad (x\ (\Phi\ x))$$

Abstracting over x, we get a recursive definition for Φ, that can be rewritten as a fixed-point equation:

$$\begin{aligned} \Phi &= \lambda x.(x\ (\Phi\ x)) \\ &= ((\lambda \phi x.(x\ (\phi\ x)))\ \Phi) \\ &= (M\ \Phi) \end{aligned}$$

We can solve this fixed-point equation using *any* fixed-point combinator. If Φ is a fixed-point combinator, then $(\Phi\ M)$ is also a fixed-point combinator. After we prove these to be distinct in the $\beta\eta$ sense, we can define an infinite chain of distinct fixed-point combinators. Furthermore, M relates $\mathrm{Y}_{\text{Curry}}$ and $\mathrm{Y}_{\text{Turing}}$ in an interesting way: $(\mathrm{Y}_{\text{Curry}}\ M) \twoheadrightarrow \mathrm{Y}_{\text{Turing}}$, which is a stronger relation than $=$.

For the purpose of this work, we consider n-ary generalizations of $\mathrm{Y}_{\text{Curry}}$ and $\mathrm{Y}_{\text{Turing}}$ to be natural if they satisfy a corresponding n-ary generalization of the above relation.

We now define n-ary generalizations of the above term M. If $\Theta_1, \ldots, \Theta_n$ are a set of n multiple fixed-point combinators, then for any $x_1, \ldots, x_n$ and $k = 1, \ldots, n$, it satisfies:

$$\begin{aligned} (\Theta_k^n\ x_1 \cdots x_n) &= (x_k\ (\Theta_1^n\ x_1 \cdots x_n) \cdots (\Theta_n^n\ x_1 \cdots x_n)) \\ &= ((\lambda \phi_1 \cdots \phi_n x_1 \cdots x_n.(x_k\ (\phi_1\ x_1 \cdots x_n) \cdots (\phi_n\ x_1 \cdots x_n))) \\ &\quad\ \Theta_1 \cdots \Theta_n) \\ &= (M_k^n\ \Theta_1 \cdots \Theta_n) \end{aligned}$$

where $M_k^n \equiv \lambda \phi_1 \cdots \phi_n x_1 \cdots x_n.(x_k\ (\phi_1\ x_1 \cdots x_n) \cdots (\phi_n\ x_1 \cdots x_n))$.

The n-ary generalizations of $\mathrm{Y}_{\text{Curry}}, \mathrm{Y}_{\text{Turing}}$ are given by Φ_k^n, Ψ_k^n, respectively, for all $k = 1, \ldots, n$.

Proposition 5.1. *For any $n > 0$ and each $k = 1, \ldots n$, we have $(\Phi_k^n\ M_1^n \cdots M_n^n) \twoheadrightarrow \Psi_k^n$.*

Proof.

$$\begin{aligned} &(\Phi_k^n\ M_1^n \cdots M_n^n) \\ &\quad \twoheadrightarrow \quad ((\lambda z_1 \cdots z_n.(M_k^n\ (z_1\ z_1 \cdots z_n) \cdots (z_n\ z_1 \cdots z_n))) \\ &\qquad\qquad (\lambda z_1 \cdots z_n.(M_1^n\ (z_1\ z_1 \cdots z_n) \cdots (z_n\ z_1 \cdots z_n))) \\ &\qquad\qquad \cdots \\ &\qquad\qquad (\lambda z_1 \cdots z_n.(M_n^n\ (z_1\ z_1 \cdots z_n) \cdots (z_n\ z_1 \cdots z_n)))) \\ &\quad \twoheadrightarrow \quad ((\lambda z_1 \cdots z_n x_1 \cdots x_n.(x_k\ (z_1\ z_1 \cdots z_n\ x_1 \cdots x_n) \cdots (z_n\ z_1 \cdots z_n\ x_1 \cdots x_n))) \\ &\qquad\qquad (\lambda z_1 \cdots z_n x_1 \cdots x_n.(x_1\ (z_1\ z_1 \cdots z_n\ x_1 \cdots x_n) \cdots (z_n\ z_1 \cdots z_n\ x_1 \cdots x_n))) \\ &\qquad\qquad \cdots \\ &\qquad\qquad (\lambda z_1 \cdots z_n x_1 \cdots x_n.(x_n\ (z_1\ z_1 \cdots z_n\ x_1 \cdots x_n) \cdots (z_n\ z_1 \cdots z_n\ x_1 \cdots x_n)))) \\ &\quad \equiv \quad \Psi_k^n \end{aligned}$$

□

We would like to define the combinator M_{AG}, which is the arity-generic generalization of the M_k^n, such that:

$$(M_{\text{AG}}\ c_k\ c_n) \quad = \quad M_k^n$$

We start with M_k^n:

$$
\begin{array}{lcl}
M_k^n & \equiv & \lambda\phi_1\cdots\phi_n x_1\cdots x_n.(x_k\ (\phi_1\ x_1\cdots x_n)\cdots(\phi_n\ x_1\cdots x_n)) \\
& = & \lambda\phi_1\cdots\phi_n x_1\cdots x_n.(\sigma_{\text{AG}}\ c_k\ c_n\ x_1\cdots x_n\ (\phi_1\ x_1\cdots x_n)\cdots(\phi_n\ x_1\cdots x_n)) \\
& = & \lambda\phi_1\cdots\phi_n.(\mathbf{S}_{\text{AG}}\ c_n \\
& & \qquad\qquad (\lambda x_1\cdots x_n.(\sigma_{\text{AG}}\ c_k\ c_n\ x_1\cdots x_n\ (\phi_1\ x_1\cdots x_n)\cdots(\phi_{n-1}\ x_1\cdots x_n))) \\
& & \qquad\qquad \phi_n) \\
& = & \lambda\phi_1\cdots\phi_n.(\mathbf{S}_{\text{AG}}\ c_n \\
& & \qquad\qquad (\mathbf{S}_{\text{AG}}\ c_n \\
& & \qquad\qquad\quad (\lambda x_1\cdots x_n.(\sigma_{\text{AG}}\ c_k\ c_n\ x_1\cdots x_n\ (\phi_1\ x_1\cdots x_n)\cdots(\phi_{n-2}\ x_1\cdots x_n))) \\
& & \qquad\qquad\quad \phi_{n-1}) \\
& & \qquad\qquad \phi_n) \\
& = & \lambda\phi_1\cdots\phi_n.\underbrace{(\mathbf{S}_{\text{AG}}\ c_n\ (\mathbf{S}_{\text{AG}}\ c_n\ (\cdots(\mathbf{S}_{\text{AG}}\ c_n}_{n\text{ times}}\ (\sigma_{\text{AG}}\ c_k\ c_n)\ \underbrace{\phi_1)\cdots)\ \phi_{n-1})\ \phi_n)}_{n\text{ times}}
\end{array}
$$

We generate such a repeated application by repeatedly applying the function f, defined so that $\langle M_r, c_r\rangle \overset{f}{\Longrightarrow} \langle M_{r+1}, c_{r+1}\rangle$. Assuming the variable n, which stands for the Church numeral c_n in the previous expression, and which occurs free in f, we define f as follows:

$$
\begin{array}{lcl}
f & = & \lambda p.\langle(\mathbf{B}_{\text{AG}}\ (\pi_2^2\ p)\ (\mathbf{S}_{\text{AG}}\ n)\ (\pi_1^2\ p)), \\
& & \quad (S^+\ (\pi_2^2\ p))\rangle
\end{array}
$$

We can now use f to define M_k^n:

$$M_k^n \ = \ (\pi_1^2\ (c_n\ f\ \langle(\sigma_{\text{AG}}\ c_k\ c_n), c_0\rangle))$$

We now define M_{AG} by abstracting c_k, c_n over the parameterized expression, to get:

$$
\begin{array}{lcl}
M_{\text{AG}} & \equiv & \lambda kn.(\pi_1^2\ (n\ f\ \langle(\sigma_{\text{AG}}\ k\ n), c_0\rangle)) \\
& \equiv & \lambda kn.(\pi_1^2\ (n\ (\lambda p.\langle(\mathbf{B}_{\text{AG}}\ (\pi_2^2\ p)\ (\mathbf{S}_{\text{AG}}\ n)\ (\pi_1^2\ p)), \\
& & \qquad\qquad\qquad (S^+\ (\pi_2^2\ p))\rangle) \\
& & \qquad\quad \langle(\sigma_{\text{AG}}\ k\ n), c_0\rangle))
\end{array}
$$

This definition satisfies the requirement that $(M_{\text{AG}}\ c_k\ c_n) = M_k^n$. Combined with Proposition 5.1, it follows that for $n \geq 1$ and for each $k = 1, \ldots, n$, we have: $(\Phi_{\text{AG}}\ c_k\ c_n\ (M_{\text{AG}}\ c_k\ c_n)) = (\Psi_{\text{AG}}\ c_k\ c_n)$. The stronger $\twoheadrightarrow$ property does not hold when working with encodings, which are by definition, β-equivalent. Finally, just as M was used to construct a chain of infinitely-many different fixed-point combinators, so can M_{AG} be used to construct a chain of infinitely-many arity-generic fixed-point combinators: If $\Phi_1^n, \ldots, \Phi_n^n$ are n multiple fixed-point combinators, then so are

$$
\begin{array}{l}
(M_{\text{AG}}\ c_1\ c_n\ \Phi_1^n\cdots\Phi_n^n), \\
\cdots \\
(M_{\text{AG}}\ c_n\ c_n\ \Phi_1^n\cdots\Phi_n^n)
\end{array}
$$

and so are

$$
\begin{array}{l}
(M_{\text{AG}}\ c_1\ c_n\ (M_{\text{AG}}\ c_1\ c_n\ \Phi_1^n \cdots \Phi_n^n), \\
\qquad\qquad\quad \cdots \\
\qquad\qquad\quad (M_{\text{AG}}\ c_n\ c_n\ \Phi_1^n \cdots \Phi_n^n)), \\
\cdots \\
(M_{\text{AG}}\ c_n\ c_n\ (M_{\text{AG}}\ c_1\ c_n\ \Phi_1^n \cdots \Phi_n^n), \\
\qquad\qquad\quad \cdots \\
\qquad\qquad\quad (M_{\text{AG}}\ c_n\ c_n\ \Phi_1^n \cdots \Phi_n^n))
\end{array}
$$

etc.

5.9.4. *Summary and conclusion.* We defined n-ary (Φ_k^n, Ψ_k^n) and arity-generic $(\Phi_{\text{AG}}, \Psi_{\text{AG}})$ generalizations of Curry's and Turing's fixed-point combinators, and showed that these generalizations maintain the n-ary and arity-generic generalizations of the relationship originally discovered by Böhm. The significance of arity-generic fixed-point combinators is that they are *single terms* that parameterize over the number of fixed-point equations and the index of a fixed point, so they can be used to find any fixed point of any number of fixed-point equations: They can be used interchangeably to define mutually-recursive procedures, mutually-recursive data structures, etc.

For example, if E, O are the *even* and *odd* generating functions given by:

$$
\begin{array}{rcl}
E & \equiv & \lambda eon.(\mathbf{Zero}?\ n\ \mathbf{True}\ (o\ (P^-\ n))) \\
O & \equiv & \lambda eon.(\mathbf{Zero}?\ n\ \mathbf{False}\ (e\ (P^-\ n)))
\end{array}
$$

Then we can use Curry's arity-generic fixed-point combinator to define the λ-terms that compute the *even* and *odd* functions on Church numerals as follows:

$$
\begin{array}{rcl}
\text{IsEven?} & \equiv & (\Phi_{\text{AG}}\ c_1\ c_2\ E\ O) \\
\text{IsOdd?} & \equiv & (\Phi_{\text{AG}}\ c_2\ c_2\ E\ O)
\end{array}
$$

Alternatively, we can use Turing's arity-generic fixed-point combinator to do the same:

$$
\begin{array}{rcl}
\text{IsEven?}' & \equiv & (\Psi_{\text{AG}}\ c_1\ c_2\ E\ O) \\
\text{IsOdd?}' & \equiv & (\Psi_{\text{AG}}\ c_2\ c_2\ E\ O)
\end{array}
$$

It might seem intuitive that in order to generate n multiple fixed points, we would need n generating expressions, and this intuition is responsible for the $O(n^2)$ size of the n-ary extensions of Curry's and Turing's fixed-point combinators. A more compact approach, however, is to pass along a single aggregation of the n fixed points, which can be done using a single generator function that is applied to itself. This approach was taken by Kiselyov [16] in his construction of a variadic, multiple fixed-point combinator in Scheme:

```
(define Y*
  (lambda s
    ((lambda (u) (u u))
     (lambda (p)
       (map (lambda (si)
              (lambda x
                (apply (apply si (p p)) x)))
            s)))))
```

A corresponding arity-generic version can be encoded in the λ-calculus in two ways. First, to emphasize the brevity of this construction, we can write:

$$\begin{aligned} Y^* \quad = \quad & \lambda n v_s.((\lambda u.(u\ u)) \\ & \qquad (\lambda p.(\mathrm{Map}_{\mathrm{AG}}\ n\ (\lambda s_i v_x.(p\ p\ s_i\ v_x))\ v_s))) \end{aligned}$$

Note that since the Apply combinator reverses its two arguments, we can avoid it altogether by reversing its two arguments *in situ*, essentially *inlining* the Apply combinator. Then for any $n \in \mathbb{N}$, let $f_1, \ldots f_n \in \Lambda$ be some λ-expressions, and let $\Phi_1^n, \ldots, \Phi_n^n$ be a set of n multiple fixed-point combinators, Y^* satisfies:

$$(Y^*\ c_n\ \langle f_1, \ldots f_n\rangle) \quad = \quad \langle(\Phi_1^n\ f_1 \cdots f_n), \ldots, (\Phi_n^n\ f_1 \cdots f_n)\rangle$$

But to be consistent with how we defined and used other arity-generic terms, we should rather define a Curried variant Y^*_{Curried}:

$$\begin{aligned} Y^*_{\mathrm{Curried}} \quad = \quad & \lambda n.(\mathbf{B}_{\mathrm{AG}}\ n\ (\lambda u.(u\ u)) \\ & \qquad (\mathbf{B}_{\mathrm{AG}}\ n\ (\mathbf{C}\ (\mathbf{B}\ (\mathrm{Map}_{\mathrm{AG}}\ n) \\ & \qquad\qquad (\lambda p s_i v_x.(p\ p\ s_i\ v_x)))) \\ & \qquad\qquad (\langle \textvisiblespace, \ldots, \textvisiblespace \rangle_{\mathrm{AG}}\ n))) \end{aligned}$$

This variant takes a Church numeral, followed by n λ-expressions, and returns the n-tuple of their multiple fixed points:

$$(Y^*_{\mathrm{Curried}}\ c_n\ f_1 \cdots f_n) \quad = \quad \langle(\Phi_1^n\ f_1 \cdots f_n), \ldots, (\Phi_n^n\ f_1 \cdots f_n)\rangle$$

So it seems that the shortest known multiple fixed-point combinator in Scheme translates to a very short multiple fixed-point combinator in the λ-calculus, perhaps the shortest known as well.

5.10. **Derivation of the Arity-Generic One-Point Basis Maker.** In a previous work [12], we have shown that for any n λ-terms $E_1, \ldots, E_n$, which need not even be combinators, it is possible to define a single term X that generates $E_1, \ldots, E_n$. Such a term is known as a *one-point basis* [4, Section 8.1].

It is straightforward to construct a *dispatcher* λ-term D, such that $(D\ c_k) = E_k$, for all $k = 1, \ldots, n$. Let $X = \langle M, c_0\rangle$, where $M = \lambda mba.(\mathbf{Zero?}\ b\ \langle m, (S^+\ a)\rangle\ (D\ b))$. Then, for any $k = 1, \ldots, n$, we have:

$$X(\underbrace{X \cdots X}_{k+1}) \quad = \quad \langle M, c_0\rangle\,(\underbrace{\langle M, c_0\rangle \cdots \langle M, c_0\rangle}_{k+1})$$

$$\begin{aligned} &= \langle M, c_0\rangle\, \langle M, c_k\rangle \\ &= (D\ c_k) \\ &= E_k \end{aligned}$$

Notice that a different dispatcher is needed for each n, and for each $E_1, \ldots, E_n$.

Using our arity-generic basis, we can abstract a Church numeral over our construction, and obtain an arity-generic one-point basis *maker*. We define M so as to use an arity-generic selector to dispatch over n expressions:

$$M \quad \equiv \quad \lambda mba.(\mathbf{Zero}?\ b\ (\lambda x.x\ m\ (S^+\ a))\ (\sigma_{\text{AG}}\ b\ c_n\ x_1 \cdots x_n))$$

We use M to define the Arity-Generic basis maker MakeX_{AG}:

$$\begin{aligned}
\text{MakeX}_{\text{AG}} &= \lambda n x_1 \cdots x_n.\,\langle M, c_0\rangle \\
&= \lambda n x_1 \cdots x_n z.(z\ M\ c_0) \\
&= \lambda n x_1 \cdots x_n z.(\mathbf{I}\ z\ M\ c_0) \\
&= \lambda n x_1 \cdots x_n z.(\mathbf{C}\ \mathbf{I}\ M\ z\ c_0) \\
&= \lambda n x_1 \cdots x_n z.(\mathbf{C}\ (\mathbf{C}\ \mathbf{I}\ M)\ c_0\ z) \\
&=_\eta \lambda n x_1 \cdots x_n.(\mathbf{C}\ (\mathbf{C}\ \mathbf{I}\ M)\ c_0) \\
&= \lambda n x_1 \cdots x_n.(\mathbf{C}\ \mathbf{C}\ c_0\ (\mathbf{C}\ \mathbf{I}\ M)) \\
&= \lambda n x_1 \cdots x_n.(\underbrace{\mathbf{B}\ (\mathbf{C}\ \mathbf{C}\ c_0)\ (\mathbf{C}\ \mathbf{I})}_{\text{aliased by } A_1}\ M) \\
&= \lambda n x_1 \cdots x_n.(A_1\ (\lambda mba.(\mathbf{Zero}?\ b\ (\lambda x.(x\ m\ (S^+\ a))) \\
&\qquad\qquad (\sigma_{\text{AG}}\ b\ n\ x_1 \cdots x_n)))) \\
&= \lambda n x_1 \cdots x_n.(A_1\ (\lambda mba.(\underbrace{\mathbf{B}_{\text{AG}}\ n\ (\mathbf{Zero}?\ b\ (\lambda x.(x\ m\ (S^+\ a))))\ (\sigma_{\text{AG}}\ b\ n)}_{\text{aliased by } A_2} \\
&\qquad\qquad x_1 \cdots x_n))) \\
&= \lambda n x_1 \cdots x_n.(A_1\ (\lambda mba.(A_2\ x_1 \cdots x_n))) \\
&= \lambda n x_1 \cdots x_n.(A_1\ (\lambda mba.(\langle x_1, \ldots, x_n\rangle\ \ A_2))) \\
&= \lambda n x_1 \cdots x_n.(A_1\ (\underbrace{(\lambda vmba.(v\ A_2))}_{\text{aliased by } A_3}\ \langle x_1, \ldots, x_n\rangle)) \\
&= \lambda n x_1 \cdots x_n.(A_1\ (A_3\ (\langle _, \ldots, _\rangle_{\text{AG}}\ n\ x_1 \cdots x_n))) \\
&= \lambda n x_1 \cdots x_n.(A_1\ (\underbrace{\mathbf{B}_{\text{AG}}\ n\ A_3\ (\langle _, \ldots, _\rangle_{\text{AG}}\ n)}_{\text{aliased by } A_4}\ x_1 \cdots x_n)) \\
&= \lambda n x_1 \cdots x_n.(\mathbf{B}_{\text{AG}}\ n\ A_1\ A_4\ x_1 \cdots x_n) \\
&=_\eta \lambda n.(\mathbf{B}_{\text{AG}}\ n\ A_1\ A_4) \\
&= \lambda n.(\mathbf{B}_{\text{AG}}\ n\ A_1 \\
&\qquad \underbrace{(\mathbf{B}_{\text{AG}}\ n\ A_3\ (\langle _, \ldots, _\rangle_{\text{AG}}\ n)}_{\text{un-aliasing } A_4}))
\end{aligned}$$

$$
\begin{aligned}
&= \lambda n.(\mathbf{B}_{\mathrm{AG}}\ n\ A_1 \\
&\qquad (\mathbf{B}_{\mathrm{AG}}\ n\ (\underbrace{(\lambda vmba.(v\ A_2))}_{\text{un-aliasing } A_3}\ (\langle \_, \ldots, \_ \rangle_{\mathrm{AG}}\ n))) \\
&= \lambda n.(\mathbf{B}_{\mathrm{AG}}\ n\ A_1 \\
&\qquad (\mathbf{B}_{\mathrm{AG}}\ n\ ((\lambda vmba.(v\ \underbrace{(\mathbf{B}_{\mathrm{AG}}\ n\ (\mathbf{Zero}?\ b\ (\lambda x.(x\ m\ (S^+\ a))))\ (\sigma_{\mathrm{AG}}\ b\ n))))}_{\text{un-aliasing } A_2} \\
&\qquad\qquad (\langle \_, \ldots, \_ \rangle_{\mathrm{AG}}\ n)))) \\
&= \lambda n.(\mathbf{B}_{\mathrm{AG}}\ n\ (\underbrace{\mathbf{B}\ (\mathbf{C}\ \mathbf{C}\ c_0)\ (\mathbf{C}\ \mathbf{I})}_{\text{un-aliasing } A_1}) \\
&\qquad (\mathbf{B}_{\mathrm{AG}}\ n\ ((\lambda vmba.(v\ (\mathbf{B}_{\mathrm{AG}}\ n\ (\mathbf{Zero}?\ b\ (\lambda x.(x\ m\ (S^+\ a)))) \\
&\qquad\qquad (\sigma_{\mathrm{AG}}\ b\ n)))) \\
&\qquad\qquad (\langle \_, \ldots, \_ \rangle_{\mathrm{AG}}\ n))))
\end{aligned}
$$

We may now define $\mathrm{MakeX}_{\mathrm{AG}}$ as follows:

$$
\begin{aligned}
\mathrm{MakeX}_{\mathrm{AG}} \equiv\ & \lambda n.(\mathbf{B}_{\mathrm{AG}}\ n\ (\mathbf{B}\ (\mathbf{C}\ \mathbf{C}\ c_0)\ (\mathbf{C}\ \mathbf{I})) \\
&\qquad (\mathbf{B}_{\mathrm{AG}}\ n\ ((\lambda vmba.(v\ (\mathbf{B}_{\mathrm{AG}}\ n\ (\mathbf{Zero}?\ b\ (\lambda x.(x\ m\ (S^+\ a)))) \\
&\qquad\qquad (\sigma_{\mathrm{AG}}\ b\ n)))) \\
&\qquad\qquad (\langle \_, \ldots, \_ \rangle_{\mathrm{AG}}\ n))))
\end{aligned}
$$

We can use $\mathrm{MakeX}_{\mathrm{AG}}$ as follows. For any $n > 1$ and $E_1, \ldots, E_n \in \Lambda$, we can define X as follows:

$$X \equiv (\mathrm{MakeX}_{\mathrm{AG}}\ c_n\ E_1 \cdots E_n)$$

We now have:

$$
\begin{aligned}
(X\ (X\ X)) &= E_1 \\
(X\ (X\ X\ X)) &= E_2 \\
&\cdots \\
(X\ \underbrace{(X \cdots X)}_{n+1}) &= E_n
\end{aligned}
$$

Notice that we have made no assumptions about $E_1 \ldots E_n$, and in particular, have not required that they be combinators. Our one-point basis maker, $\mathrm{MakeX}_{\mathrm{AG}}$, provides an abstract mechanism for packaging λ-terms, in a way that they can later be "unpacked".

5.11. **Summary and Conclusion.** We used our extended basis and bracket-abstraction algorithm to encode useful arity-generic λ-terms of increasing complexity. We took the approach that working with sequences of expressions in an intuitive, modular and systematic way should resemble "list processing" known from LISP/Scheme and other functional programming languages.

In the spirit of list processing, the first part of this section introduces arity-generic λ-terms for picking elements of sequences, constructing ordered n-tuples, applying λ-terms to the elements of a tuple, extending and reversing tuples, and constructing new ordered n-tuples by *mapping* over existing tuples. All these λ-terms correspond to the basic machinery for list processing, e.g., in LISP/Scheme. Once these were defined, we were ready to look at more complex arity-generic λ-terms.

Our detailed examples include arity-generic fixed-point combinators, and an arity-generic generator for one-point bases.

We encoded arity-generic generalizations of two historical fixed-point combinators by Curry and Turing. These fixed-point combinators maintain a relationship discovered by Böhm, so it is natural to wonder whether this relationship is maintained in the arity-generic generalizations of these fixed-point combinators, and we have shown this to be the case up to β-equivalence.

We then encoded an arity-generic generator for one-point bases, so that any number of λ-terms can be "compacted" into a single expression from which they can be generated.

We tested all the arity-generic definitions in this work using a normal-order reducer for the λ-calculus, and have verified that they behave as expected on an array of examples.

6. Related Work

The expressive power of the λ-calculus has fostered the advent of functional languages. For example, the Algorithmic Language Scheme [27] was developed as an interpreter for the λ-calculus, and offered programmatic support for playing with λ-definability, from Church numerals to a call-by-value version of Curry's fixed-point combinator [25]. Since Scheme provides linguistic support for variadic functions, it has become a sport to program call-by-value fixed-point operators for variadic functions. Queinnec presented the Scheme procedure `NfixN2`, that is a variadic, applicative-order multiple fixed-point combinator [23, Pages 457–458]. The author presented one that directly extends Curry's fixed-point combinator [13] and was a motivation for Section 5.9.

The original aim of the Combinatory-Logic program, as pursued by Schönfinkel [24], was the elimination of bound variables [6]. To this end, Schönfinkel introduced five constants, each with a conversion rule that described its behavior. These constants are known today as $\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$. While Schönfinkel did not leave an explicit *abstraction algorithm* for translating terms with bound variables to equivalent terms without bound variables [9, page 8], Cardon and Hindley claim it extremely likely that he knew of such an algorithm [6].

As far as we have been able to verify, the first to have considered the question of how to encode inductive and arity-generic λ-terms was Curry, first in an extended Combinatory Logic framework [8], where Curry first mentions such variables, and refers to them as *apparent variables*, and later, for Combinatory Logic [9, Section 5E]. We have not found this terminology used elsewhere, and since the term *arity-generic* is much more self-explanatory, we have chosen to stick with it.

Abdali, in his article *An Abstraction Algorithm for Combinatory Logic* [1], presented a much simpler algorithm for encoding inductive and arity-generic λ-terms. Abdali introduces the terms:

- $\mathscr{K}$, which is an arity-generic generalization of $\mathbf{K}$, and identical to the $\mathbf{K}_{\text{AG}}$ combinator used throughout this article.
- $\mathscr{I}$, which is an arity-generic selector, and is identical to the σ_{AG} combinator introduced in Section 5.1.
- $\mathscr{B}$, which is a *double* arity-generic generalization of Curry's $\Phi = \lambda xyzu.(x\ (y\ u)\ (z\ u))$ combinator [9], generalized for two independent indices.

These combinators can augment any basis, and provide for a straightforward encoding of arity-generic λ-terms. Abdali does not explain how he came up with the double generalization of Curry's Φ combinator, or how he encoded the definitions for $\mathscr{K}, \mathscr{I}, \mathscr{B}$ in terms of the basis he chose to use. Arity-generic expressions encoded using $\mathscr{K}, \mathscr{I}, \mathscr{B}$, are not as concise as they could be, because the $\mathscr{B}$ combinator introduces variables even in when they are not needed in parts of an application, and in such cases, a subsequent projection is needed to remove them.

Barendregt [4] seems to have considered this question at least for some special cases, as in Exercises 8.5.13 and 8.5.20, the later of which he attributes to David A. Turner.[3]

Schönfinkel's original $\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$ basis, coupled with Turner's bracket-abstraction algorithm for that basis, offers several advantages in terms of brevity of the resulting term, simplicity, intuitiveness and ease of application of the algorithm. In the original bracket-abstraction algorithm for $\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$, the length of the encoded λ-term is less than or equal to the length of the original λ-term, because each application is replaced by a combinator, and abstractions are either represented by a single combinator, or are removed altogether through η-reduction. The additional arity-generic combinators with which we extended the $\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$ basis maintain this conciseness, because a sequence of left-associated applications to a sequence of variables is replaced by a single arity-generic combinator, and a sequence of Curried, nested λ-abstractions is either removed via repeated η-expansions, or is replaced with by a single arity-generic combinator. The extension of the basis and the corresponding bracket-abstraction algorithm to handle arity-generic λ-terms is straightforward and intuitive.

7. Discussion

The ellipsis ('$\cdots$') and its typographical predecessor '&c' (an abbreviation for the Latin phrase *et cetera*, meaning "and the rest") have been used as meta-mathematical notation, to abbreviate mathematical objects (numbers, expressions, formulae, structures, etc.) for hundreds of years, going back to the 17th century and possibly earlier. Such abbreviations permeate the writings of Isaac Newton, John Wallis, Leonhard Euler, Carl Friedrich Gauss, and up to the present. Despite its ubiquity, and perhaps as a paradoxical tribute to this ubiquity, the ellipsis does not appear as an entry in standard texts on the history of mathematical notation, even though the authors of these texts make extensive use of ellipses in their books [5, 20]. Neither is the ellipsis discussed in the Kleene's classical text on meta-mathematics [18], nor does it even appear as an entry in the list of symbols and notation

[3]Barendregt refers to Turner's article *A New Implementation Technique for Applicative Languages* [29], but as this article contains no mention of n-ary expressions and their encoding in the λ-calculus, it is plausible that he had really intended to refer to another article by Turner, also published in 1979: *Another Algorithm for Bracket Abstraction* [28].

at the end of the book, even though Kleene makes extensive use of the ellipses both in the main text as well as in the list of symbols and notation.

Discussions about the ellipsis and its meanings seem to concentrate in computer literature: Roland Backhouse refers to the ellipsis as the *dotdotdot notation* in one of the more mathematical parts of his book *Program Construction: Calculating Implementations From Specifications* [3, Section 11.1], and suggests that they have many disadvantages, the most important being that "... it puts a major burden on the reader, requiring them to interpolate from a few example values to the general term in a bag of values." Some of the examples of ellipses he cites can be rewritten using summations, products, and the like. Others, however involve the meta-language, e.g., functions that take n arguments, where n is a meta-variable. Such examples of ellipses cannot be removed as easily.

The ellipsis also appears in some programming languages. In some languages (C, C++, and Java) it is used to define variadic procedures. In other languages (Ruby, Rust, and GNU extensions to C and C++) it is used to define a range. In Scheme, the ellipsis is part of the syntax for writing macros, which can be thought of as a meta-language for Scheme. A formal treatment of ellipses in the macro language for Scheme was done by Eugene Kohlbecker in his PhD thesis [19].

Arity-generic terms are somewhat reminiscent of *variadic* procedures in programming languages: The term *variadic*, introduced by Strachey [26], refers to the *arity* of a procedure, i.e., the number of arguments to which it can be applied. A *dyadic* procedure can be applied to two arguments. A *triadic* procedure can be applied to three arguments. A *variadic* procedure can be applied to any number of arguments. Programming languages that provide a syntactic facility for defining variadic procedures include C++ and LISP/Scheme. The λ-calculus has no such syntactic facility, and so it is somewhat of a misnomer to speak of *variadic λ-terms*, since the number of arguments is an *explicit* parameter in our definitions, whereas in the application of a variadic procedure to some arguments, the *number* of arguments is *implicit* in an implementation. Nevertheless, within the classical, untyped λ-calculus, arity-generic λ-terms provide an expressivity that comes very close to having variadic λ-terms.

Variadic procedures are not just about the procedure *interface*. When used in combination with `map` and `apply`, they can provide a kind of generality that is typically deferred to the meta-language or macro system [13, 19]. Arity-generic λ-definability achieves similar generality in the classical λ-calculus, with some notable differences: Variadic procedures are applied to arbitrarily-many arguments, and their parameter is bound to the list of the values of these arguments. By contrast, arity-generic expressions take the number of arguments, and return that many Curried λ-abstractions. In this work, we used ordered n-tuples, rather than linked lists, as is common in most functional programming languages, in what is perhaps reminiscent of array programming languages. As a result of the choice to use ordered n-tuples, the *apply* operation became very simple. It would be straightforward to choose to use linked lists instead, at the cost of having to define *apply* as a *left fold* operation.

In this work we show how to define, in the language of the λ-calculus, expressions that contain meta-linguistic ellipses, the size of which is indexed by a meta-variable. For such an indexed λ-term E_n, our goal was to find a term E_{AG} that takes n as an explicit parameter, and assuming it to be a Church numeral denoting the size of the indexed expression, evaluates to E_n: $(E_{\text{AG}}\ c_n) = E_n$. We call E_{AG} an arity-generic generalization of E.

Of course, our choice of using Church numerals in this paper is based on their ubiquity. In fact, any numeral system can be used, and we have also constructed an arity-generic basis around Scott numerals [30].

Our approach has been to extend the basis $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ with the arity-generic generalizations of $\mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}$ combinators and to extend Turner's bracket-abstraction algorithm to handle abstractions of sequences of variables over an expression. We then used this extended basis and this extended bracket-abstraction algorithm to encode arity-generic λ-terms. Our goal has not been to remove all abstractions in arity-generic terms, but only those abstractions that are over sequences of variables. Of course, it is possible to remove all remaining abstractions, but our goal here has been to define indexed expressions in the λ-calculus, without resorting to meta-linguistic ellipses, for which the removal of all abstractions is unnecessary.

In the first part of this work we presented a natural, arity-generic generalization to Schönfinkel's $\{\mathbf{I}, \mathbf{K}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ basis for the set of combinators in the $\lambda\mathbf{K}\beta\eta$-calculus, and extended Turner's bracket-abstraction algorithm to make use of the additional arity-generic combinators in the extended basis. The extended algorithm retains the conciseness and simplicity of Turner's original algorithm.

The second part of this work uses the arity-generic basis and the corresponding bracket-abstraction algorithm to develop tools for arity-generic λ-definability, and incidentally demonstrates how the arity-generic basis can be used: We introduced several arity-generic λ-terms that perform a wide variety of computations on ordered n-tuples. These computations were inspired by, and resemble to some extent, the facilities for *list manipulation* that are native to the LISP/Scheme programming language [2, 11, 21]: Terms that compute mappings, reversal, arity-generic fixed-point combinators, arity-generic one-point bases, etc. Implementing in the λ-calculus a functional subset of the list processing capabilities of LISP/Scheme is a popular exercise.

In his textbook on the λ-calculus, Barendregt states that there are two ways to define ordered n-tuples: Inductively, using nested ordered pairs, and another way, which Barendregt characterizes as being "more direct", as $\langle M_0, \ldots, M_n \rangle = \lambda z.(z\ M_0 \cdots M_n)$ [4, pages 133-134]. Section 5.3 shows how to make this more direct definition inductive.

In a previous work [13], we derived an applicative-order, variadic fixed-point combinator in Scheme. In that work, we relied on Scheme's support for writing variadic procedures, and consequently, on the primitive procedure `apply`, to apply procedures to lists of their arguments. In the present work, we had control over the representation of sequences, so we could encode an arity-generic version of `apply`, as well as arity-generic fixed-point combinators, all within the λ-calculus.

Acknowledgments

The author is grateful to his anonymous reviewers and to his editor, Neil D. Jones. Thanks are also due to John Franco and Albert Meyer for comments and questions about a previous work, and to Olivier Danvy for his encouragement and suggestions.

References

[1] S. Kamal Abdali. An abstraction algorithm for combinatory logic. *The Journal of Symbolic Logic*, 41(1):222–224, March 1976.

[2] Harold Abelson and Gerald Jay. *Sussman with Julie Sussman. Structure and Interpretation of Computer Programs*. MIT Press, 1985.

[3] Roland Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[4] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North-Holland, revised edition, 1984.

[5] Florian Cajori. *A history of mathematical notations*. Dover Publications, 1993.

[6] Felice Cardone and J. Roger Hindley. Lambda-calculus and combinators in the 20th century. In Dov M. Gabbay and John Woods, editors, *Logic from Russell to Church*, volume 5 of *Handbook of the History of Logic*, pages 723–817. North-Holland, 2009.

[7] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[8] Haskell B. Curry. Apparent variables from the standpoint of combinatory logic. *Annals of Mathematics*, 34(3):381–404, July 1933.

[9] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume I. North-Holland Publishing Company, 1958.

[10] Haskell B. Curry, J. Roger Hindley, and Jonathan P. Seldin. *Combinatory Logic*, volume II. North-Holland Publishing Company, 1972.

[11] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer*. Science Research Associates, Inc, 1986.

[12] Mayer Goldberg. A construction of one-point bases in extended lambda calculi. *Information Processing Letters*, 89(6):281 – 286, 2004.

[13] Mayer Goldberg. A variadic extension of Curry's fixed-point combinator. *Higher-Order and Symbolic Computation*, 18(3/4):371–388, 2005.

[14] Mayer Goldberg. The Lambda Calculus: Outline of lectures., 2007-2011. Department of Computer Science, Ben-Gurion University. Document URL: `http://lambda.little-lisper.org/`.

[15] Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[16] Oleg Kiselyov. Simplest poly-variadic fix-point combinators for mutual recursion. `http://okmij.org/ftp/Computation/fixed-point-combinators.html`, 2002.

[17] Stephen C. Kleene. A Theory of Positive Integers in Formal Logic. Part I. *American Journal of Mathematics*, 57(1):153–173, January 1935.

[18] Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland Publishing, 1964.

[19] Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp*. PhD thesis, Indiana University, Computer Science Department, Bloomington, Indiana, 1986.

[20] Joseph Mazur. *Enlightening Symbols: A Short History of Mathematical Notation and Its Hidden Powers*. Princeton University Press, 2014.

[21] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, 1962.

[22] Sandra Pakin. *APL\360 reference manual*. Science Research Associates, Inc., 1972.

[23] Christian Queinnec. *LISP In Small Pieces*. Cambridge University Press, 1996.

[24] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924. Translated by Stefan Bauer-Mengelberg as "On the building blocks of mathematical logic", in Jean van Heijenoort, 1967. *A Source Book in Mathematical Logic*, 1879–1931. Harvard University Press. Pages 355–66.

[25] Guy L. Steele Jr. and Gerald J. Sussman. Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1976.

[26] Christopher Strachey. Fundamental concepts in programming languages. International Summer School in Computer Programming, Copenhagen, Denmark, August 1967. Reprinted in Higher-Order and Symbolic Computation 13(1/2):11–49, 2000.

[27] Gerald J. Sussman and Guy L. Steele Jr. Scheme: An interpreter for extended lambda calculus. AI Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1975. Reprinted in Higher-Order and Symbolic Computation 11(4):405–439, 1998.

[28] David A. Turner. Another algorithm for bracket abstraction. *The Journal of Symbolic Logic*, 44(2):267–270, June 1979.

[29] David A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9(9):31–49, 1979.

[30] Christopher P. Wadsworth. Some unusual λ-calculus numeral systems. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 215–230. Academic Press, London, 1980.

[31] Stephanie Weirich and Chris Casinghino. Arity-generic datatype-generic programming. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 15–26, New York, NY, USA, 2010. ACM.