Replication, Sharing, Deletion, Lists, and Numerals:
Progress on Universally Programmable Intelligent Matter

UPIM Report 3

Technical Report UT-CS-02-493

Bruce J. MacLennan*

Department of Computer Science
University of Tennessee, Knoxville
www.cs.utk.edu/~mclennan

November 22, 2002

Abstract

This report addresses and resolves several issues in the use of combinators for molecular computation. The issues include assumptions about binding sites and linking groups, "capping" of unused sites, replication and sharing of structures in a molecular context, creation of cyclic structures, disassembly of unneeded structures, representation of Boolean values and conditionals, representation of LISP-style lists, and representation of numerals.

1 Introduction

This report is not intended to provide an introductory or systematic presentation of combinatory logic; necessary background information is in a previous report, the "Molecular Combinator Reference Manual" [Mac02a], which (1) defines general terminology and notation; (2) defines the combinators and states important properties of them; and (3) defines related notations (mostly involving subscripts and superscripts) with their properties. Sections and equations from that report will be cited, for example, "Sec. 2 [Mac02a]" or "Eq. 50 [Mac02a]."

*This research is supported by Nanoscale Exploratory Research grant CCR-0210094 from the National Science Foundation. It has been facilitated by a grant from the University of Tennessee, Knoxville, Center for Information Technology Research.

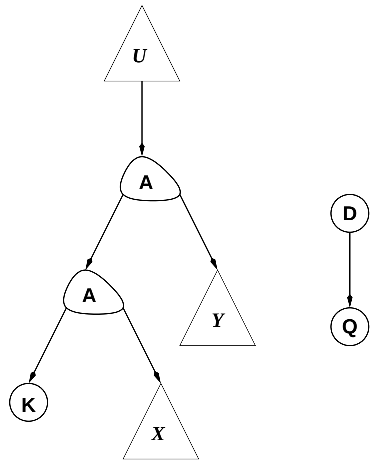

Figure 1: Diagram of reactants for K substitution. Arrows represent linking groups; small rounded triangular shape is A (application) complex; circular shapes are primitive combinators; large triangular shapes (labeled U, X, and Y) represent arbitrary combinator networks.

2 Links

A molecular combinator network comprises various *nodes* (primitive molecular complexes) connected by *links* (linking groups). In order for computation to proceed correctly, the links must be *directed* or *oriented*, and so we usually show them with arrowheads in diagrams such as Fig. 1 (which shows the reactants for a K-substitution, see Sec. 9 [Mac02a]). In accord with computer science convention, the arrow points from the parent node to its offspring, that is, downward in trees. (As is common in expression trees, the dataflow is upward, and therefore against the arrows.) In molecular terms, the link is a molecular group with distinct binding sites at its ends, which we may call the *head* and *tail*.

Nodes may be classified as *leaves* or *interior nodes*. Most leaves are molecular groups with a single binding site, to which the head of a link can bind. In computational terms, they deliver a result but have no inputs. The most common leaves are primitive combinators such as S and K. A few leaf types (D, P), which will be discussed later, bind the tail of a link.

So far, all interior nodes have three binding sites. The most common is the *application* or A primitive, which represents the application of a function to its parameter. Therefore, an A node has two "inputs," representing the function and its parameter, to which the tails of links can bind, and it has one "output," representing the result of applying the function to its parameter, to which the head of a link can bind (see Fig. 1 for examples). Some other primitives (e.g., R, V, discussed later) have one "input" and two "outputs."

Because of their interpretation in expression trees, we define a *result* site to be a site to which a link head can bind, and an *argument* site to be one to which a link tail binds. Therefore we can say: S and K each have one result site; D and P each have one argument site; A has two argument sites and one result site; R and V have two result sites and one argument.

3 Result and Argument Caps

In doing molecular combinatory programming, we deal with complexes only when they are in well-defined, stable states; in particular, we deal with them only when all the binding sites are filled, not during transient stages when binding sites may be unfilled or shared between two groups. Therefore, when complexes have unused binding sites (e.g., when they are available as reactants or generated as reaction waste products), they must be filled by some place-holders. For this purpose we have postulated two otherwise inert groups, P and Q. The *result cap* P can fill or "cap" a result site on any complex; likewise the *argument cap* Q can fill an argument site. When both are required as reactants, they may come bound as a pair PQ (e.g., Figs. 6, 10–13, pp. 6–12).

As a consequence of the foregoing rules, molecular combinatory reactions permute the sites to which the affected links are bound, but do not create or destroy any links or other molecular groups.

4 Replication and Sharing

4.1 The Problem

Combinatory logic is a *term-rewriting system* [HO82, Ros73] or *abstract calculus* [Mac90]. Therefore, a rule such as S-substitution,

$$\mathsf{S}XYZ \Longrightarrow XZ(YZ),$$

can be thought of as an operation on parenthesized linear expressions,

$$(((\mathsf{S}X)Y)Z) \Longrightarrow ((XZ)(YZ)), \tag{1}$$

or as an operation on trees, as shown in Fig. 2. The latter interpretation is, of course, what suggests combinatory logic as a basis for universal molecular computation. However, as discussed in a previous report [Mac02b], there are differences between term-rewriting systems and molecular processes. In the context of term-rewriting systems, the copying of a term, such as Z in Fig. 2 or Eq. 1, is assumed to be an atomic (constant-time) operation. This is certainly a poor assumption for molecular computation, in which the replication of a large structure could take considerable time.

Constant-time copying is also a poor assumption in conventional computation, and so implementations of term-rewriting systems typically share a single copy of a structure rather than making multiple copies; this is shown in Fig. 3. This strategy works because the term-rewriting systems of greatest interest (including combinatory logic) satisfy the

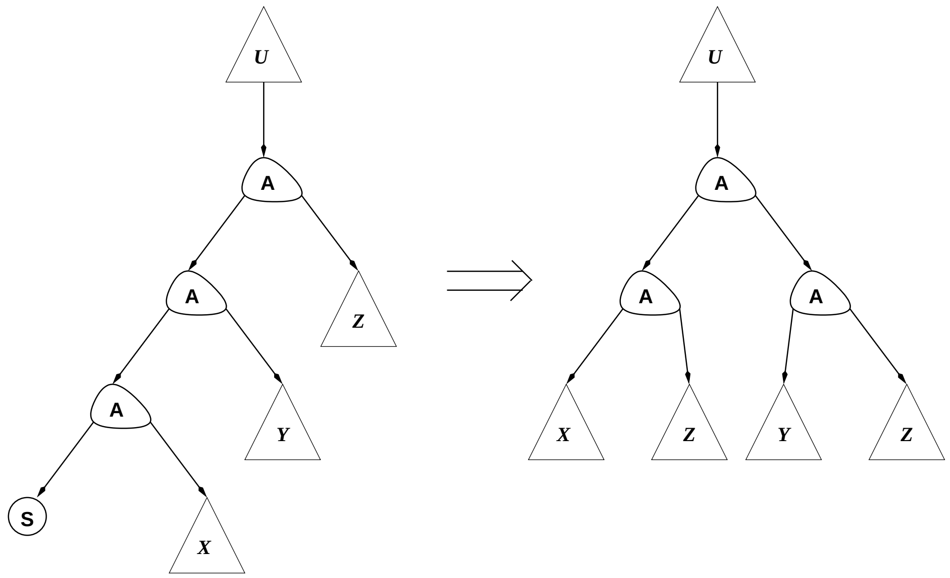

Figure 2: S-substitution with copying. U, X, Y, and Z are any combinator trees. In this implementation of the S operation, the tree Z is copied.

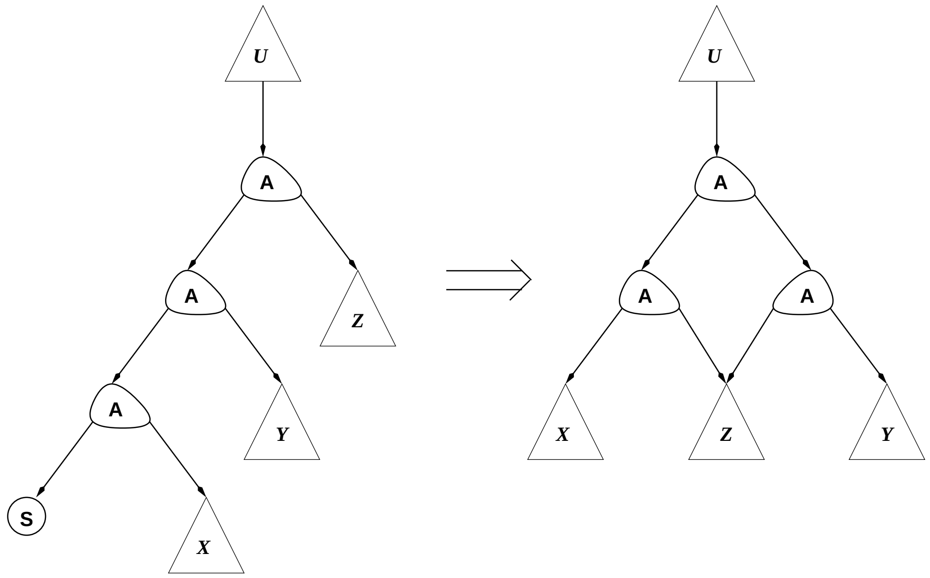

Figure 3: S-substitution with sharing. U, X, Y, and Z are any combinator trees. In this implementation of the S operation, an additional pointer is created to the tree Z.

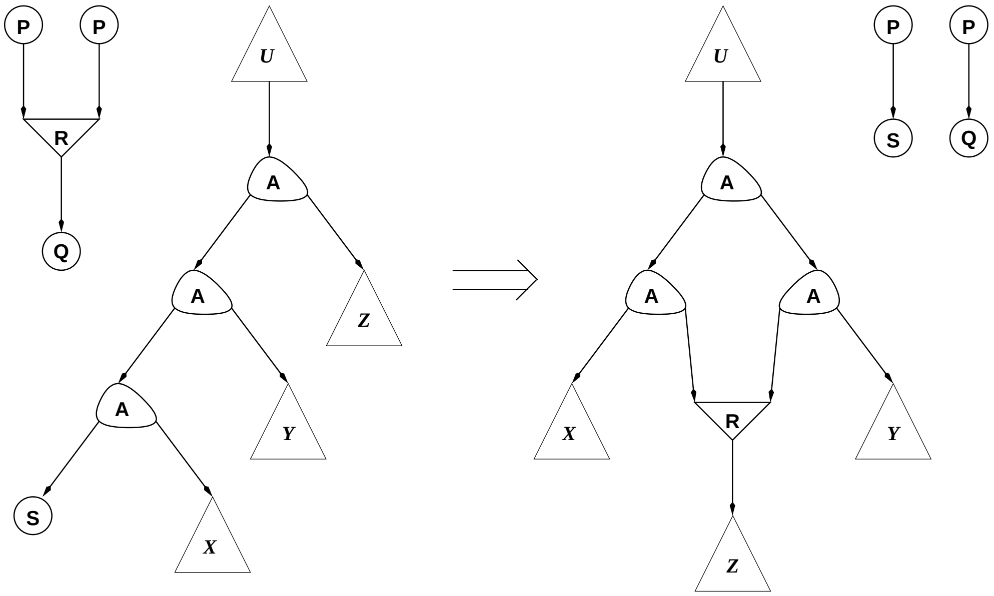

Figure 4: S-substitution reaction with replication. The reaction introduces an R (replicator) complex, which begins the replication of Z, which can proceed in parallel with other substitutions. This diagram shows both the reactants needed for the substitution as well as the reaction products. Notice that the two A complexes on the right-hand side are oriented in opposite directions.

Church-Rosser Property [CR36, Ros73], which implies that such sharing will not affect the results of computation (interpreted as linear parenthesized expressions; it may produce different graph structures). Unfortunately, this does not seem to be a good approach for molecular computation, since it may result in an unlimited number of pointers to a structure. In molecular terms, this would correspond to an unlimited number of links to a binding site, which is impossible.

Various ways around this problem, such as having binary "fan-in" nodes to the shared structure, do not seem feasible, since these intermediate nodes would block the application of the computational reactions. Therefore we have opted for a different solution, described in the following section.

4.2 Replication

Our approach is something of a hybrid between the copying and sharing implementations; it might be called "lazy replication." The two uses begin by linking to a single copy of a tree, which is gradually split into two replicates (see Fig. 4). Thanks to the Church-Rosser Property, as soon as the roots of the replicates are separated, they can begin to be used separately, although some processes might have to wait until the replication has sufficiently progressed.

The most important reaction is illustrated in Fig. 5: when replication encounters an A

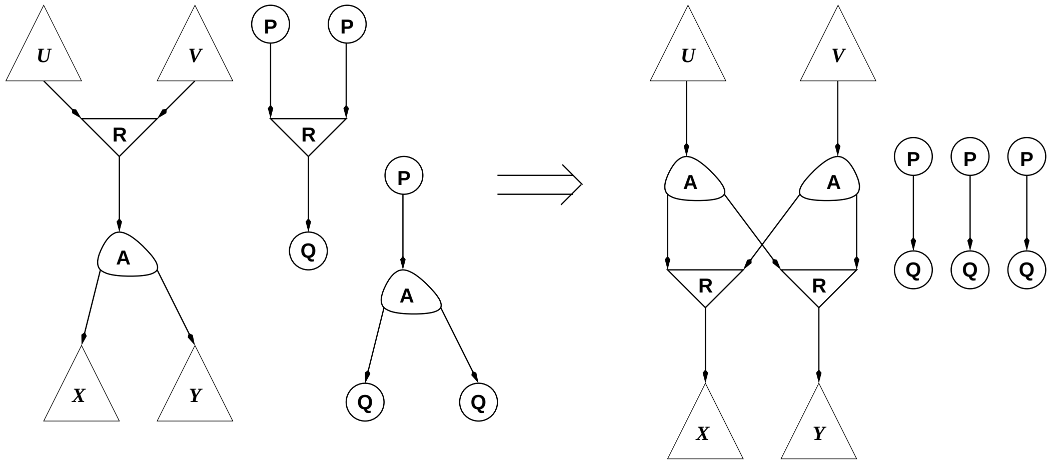

Figure 5: Replication of an application (A primitive), which triggers replication of its two daughter nodes, which can proceed in parallel. Notice that the two A primitives on the right-hand side are oriented in opposite directions.

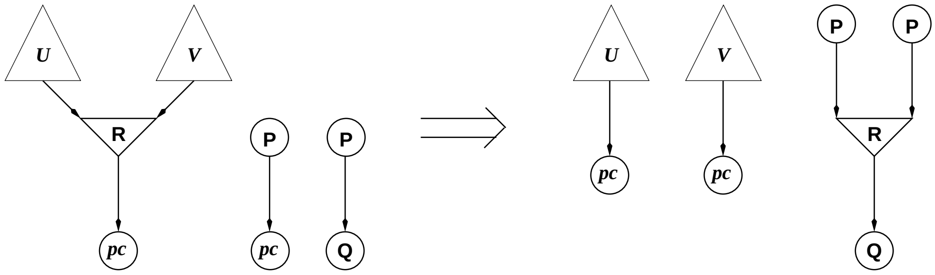

Figure 6: Replication of a primitive combinator complex pc (such as S, K, or Y̌).

(application) primitive linked to subtrees X and Y, a new A is allocated and the two As are linked to corresponding replicates of X and Y, thus triggering recursive replication of the subtrees. The molecular reaction is described:

$$UV\mathsf{RA}XY + \mathsf{PAQ}_2 + \mathsf{P}_2\mathsf{RQ} \longrightarrow UV\mathsf{A}_2\mathsf{R}_2XY + 3\mathsf{PQ}.$$

The reactants include "capped" A and R groups; the reaction releases three PQ pairs as waste.

Eventually replication will reach a leaf of the tree, that is, a primitive combinator (e.g., S, K, or Y̌); replication terminates with the allocation of a new instance of the primitive (Fig. 6). The reaction is simply:

$$UV\mathsf{R}p + \mathsf{P}p + \mathsf{PQ} \longrightarrow Up + Vp + \mathsf{P}_2\mathsf{RQ},$$

where p is any primitive combinator. The R complex, appropriately "capped," is released as a reaction waste product.

Complete reaction specifications for replication can be found in Sec. 13 [Mac02a]. Replication also interacts with deletion, which will be discussed in Sec. 6 of this report.

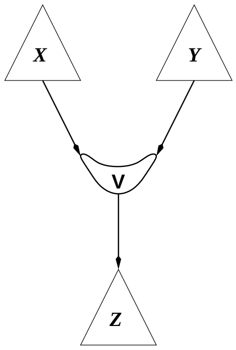

Figure 7: The V primitive allows sharing of a network. Networks X and Y both connect to network Z via V (sharing) primitive.

5 Sharing

5.1 The V Primitive

As remarked, the copying and sharing implementations (Figs. 2 and 3) are equivalent for term-rewriting systems satisfying the Church-Rosser Property, such as combinatory logic, but they result in different network structures. Since one goal of universally programmable intelligent matter is the assembly of specific nanostructures, we must be able to control the networks that are constructed. Therefore, although lazy replication is a good solution for implementing combinatory reductions, there will be circumstances in which we will want to create specific shared structures; one obvious example is the creation of cyclic structures (Sec. 5.3).

To accomplish sharing we postulate a *sharing primitive*, denoted V (to suggest its shape), which allows two links to point at one binding site; therefore V has two result sites and one argument site (Fig. 7). Situations in which more than two links are intended to point to the same destination are accomplished by using multiple V groups. Thus, a V primitive occurs in the same configuration as a R primitive, but it does not trigger replication. Conceptually, and perhaps physically, it acts like an inert replication operator.

Notice that the V primitive introduces an extra level of indirection between the shared structure and the references to it (Fig. 7). This is a fundamental difference between *symbolic linking*, such as we have on conventional computers, in which any number of cells may hold the address of the shared structure, and *physical linking*, such as we have in molecular computation. As a consequence, shared structures cannot be used computationally with full generality, since the V groups will often disrupt the patterns that trigger the computational reactions. There are various ways of working around these limitations, but they seem unduly complicated. For now it seems better to restrict the use of V to the construction of noncomputational nanostructures.

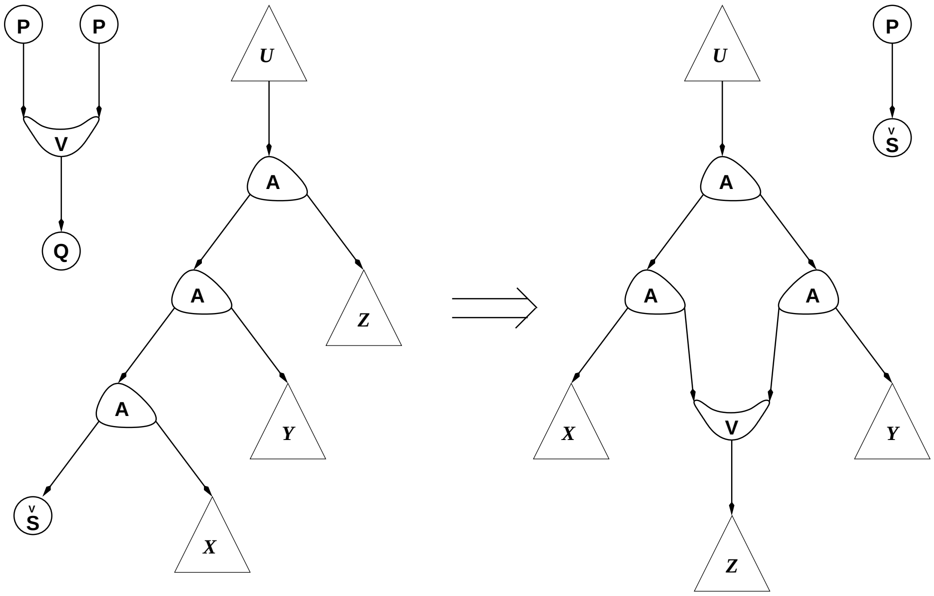

Figure 8: Š-substitution, which introduces a V (sharing) primitive. The V complex will block most other substitutions; therefore its primary purpose is the creation of static shared structures. Notice that the two A complexes on the right-hand side are oriented in opposite directions.

5.2 The Š Combinator

It is not enough to have a sharing primitive; we must also have some general means of introducing it into molecular structures. The simplest approach is suggested by the parallel between replication and sharing: modify a R-producing operator to produce a V instead. Two of the simplest combinators that replicate their arguments are S and W:

$$\mathsf{S}XYZ \implies XZ(YZ), \tag{2}$$

$$\mathsf{W}XY \implies XYY. \tag{3}$$

A sharing version of S, which we denote Š, is shown in Fig. 8; its reaction is:

$$U\mathsf{A}_3\check{\mathsf{S}}XYZ + \mathsf{P}_2\mathsf{VQ} \longrightarrow U\mathsf{A}_3XY\mathsf{V}Z + \mathsf{P}\check{\mathsf{S}} + \mathsf{PQ}. \tag{4}$$

To notate the fact that a structure is shared, we often use parenthesized superscripts, and so we may write:

$$\check{\mathsf{S}}XYZ \Longrightarrow XZ^{(1)}(YZ^{(0)}), \tag{5}$$

or, less precisely, we use primes, $\check{\mathsf{S}}XYZ \implies XZ'(YZ)$. (See also Sec. 17 [Mac02a] on the notation for sharing.)

Since W is simpler than S, a corresponding sharing operation W̌ might seem a better choice. It would be defined

$$\check{\mathsf{W}}XY \Longrightarrow XY^{(1)}Y^{(0)}, \tag{6}$$

with a corresponding reaction:

$$U\mathsf{A}_2\check{\mathsf{W}}XY + \mathsf{P}_2\mathsf{VQ} \longrightarrow U\mathsf{A}_2X\mathsf{V}Y + \mathsf{P}\check{\mathsf{W}} + \mathsf{PQ}. \tag{7}$$

Nevertheless, we have decided tentatively to take $\check{\mathsf{S}}$ as the primitive. There are several reasons:

1. The reaction for $\check{\mathsf{S}}$ (Fig. 8) is very similar to that for S (Fig. 4); the former has V where the latter has R. Therefore one reaction might be modified to yield the other.
2. $\check{\mathsf{S}}$ can be defined in terms of $\check{\mathsf{W}}$ and vice versa. However, the definition of $\check{\mathsf{W}}$ in terms of $\check{\mathsf{S}}$ (Eq. 45 [Mac02a]), $\check{\mathsf{W}} = \mathsf{C}\check{\mathsf{S}}\mathsf{I}$, is much simpler than that of $\check{\mathsf{S}}$ in terms of $\check{\mathsf{W}}$ (Eq. 34 [Mac02a]), $\check{\mathsf{S}} = \mathsf{B}(\mathsf{B}(\mathsf{B}\check{\mathsf{W}})\mathsf{C})(\mathsf{BB})$.

Nevertheless we will often find that $\check{\mathsf{W}}$ is more convenient in programming; in particular we can exponentiate it (Sec. 28 [Mac02a]) to create *sharing chains*,

$$\check{\mathsf{W}}^n XY \Longrightarrow XY^{(n)}Y^{(n-1)}\cdots Y^{(1)}Y^{(0)}, \tag{8}$$

which can be used to link together large structures.

5.3 The $\check{\mathsf{Y}}$ Combinator

The so-called *paradoxical* or *fixed-point* combinator Y is defined so that

$$\mathsf{Y}F \Longrightarrow F(\mathsf{Y}F). \tag{9}$$

It's easy to see that this leads to a nonterminating process:

$$\mathsf{Y}F \Longrightarrow F(\mathsf{Y}F) \Longrightarrow F(F(\mathsf{Y}F)) \Longrightarrow F(F(F(\mathsf{Y}F))) \Longrightarrow \cdots \tag{10}$$

Nevertheless this operation is useful in conventional functional programming for defining recursive functions [Bur75, Mac90]. Whether it will be similarly useful in molecular combinatory programming is less obvious, but if it is needed, it can be defined in terms of S and K (Sec. 21 [Mac02a]), so it does not need to be supported by a primitive reaction.

However, as we have seen (Sec. 4.1), sharing and copying are closely related, and cyclic structures are abstractly equivalent to infinite structures. Similarly, the infinite expansion of $\mathsf{Y}F$ can be interpreted as a cyclic structure, $\mathsf{Y}F \Longrightarrow y$, where $y = Fy$. This suggests that an appropriate (sharing) version of Y might be used to construct cyclic structures. (Indeed, in many functional programming language implementations on conventional computers, Y creates a self-referential structure; hence its use to implement recursion.)

Figure 9 shows a reaction implementing $\check{\mathsf{Y}}$, a sharing version of the fixed-point combinator. The reaction is described:

$$U\mathsf{A}\check{\mathsf{Y}}F + \mathsf{P}_2\mathsf{VQ} \longrightarrow U\mathsf{VA}F + \mathsf{P}\check{\mathsf{Y}} + \mathsf{PQ}. \tag{11}$$

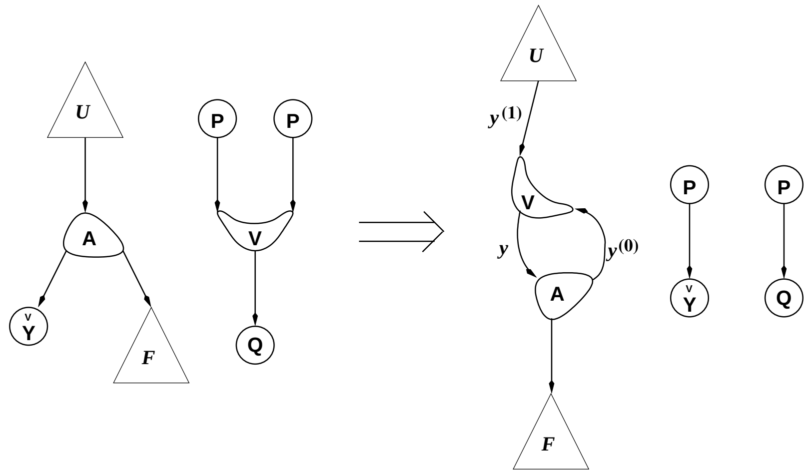

Figure 9: $\check{\mathsf{Y}}$-substitution, which introduces a cycle by means of a V (sharing) primitive. The indicated sharing structure may be described $\check{\mathsf{Y}}F \Longrightarrow y^{(1)}$ where $y \equiv Fy^{(0)}$. Normally F is a combinatory program complex, which will lead to further substitutions that will expand the cycle into a more useful structure.

Using our convention (Sec. 5.2) for notating sharing, the creation of the cyclic structure can be written as a substitution rule:

$$\check{\mathsf{Y}}F \Longrightarrow y^{(1)} \quad \text{where } y \equiv Fy^{(0)}. \tag{12}$$

Here y is defined as a name for the result of the A node in Fig. 9, and $y^{(1)}$ and $y^{(0)}$ are the two links to the V primitive.

The creation of the very tight cycle between the A and V nodes, shown in Fig. 9, might not seem very useful, but it is, as can be seen when we realize that F can be any combinatory complex, and therefore $Fy^{(0)}$ can result in very complex computations involving the link $y^{(0)}$. Examples will be presented in later reports.

6 Deletion

Combinator computation proceeds by permuting, replicating, and deleting network structures [CFC58, Sec. 5H]. In a molecular context, this means that the computational process will generate many waste structures. These could, of course, be abandoned, but it seems better to arrange for their disassembly, so that their component groups can be recycled as reaction resources. Indeed, without such recycling the reaction space could become cluttered with waste products, and residual but useless computation in discarded complexes could consume valuable reaction resources. Therefore, at this time at least, it appear preferable to arrange for the disassembly and recycling of deleted structures. To accomplish this, we postulate a primitive D (deletion) operator, which may be linked to a network to disassemble it recursively.

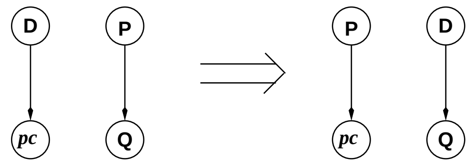

Figure 10: Deletion of a primitive combinator complex pc (such as S, K, or Y̌).

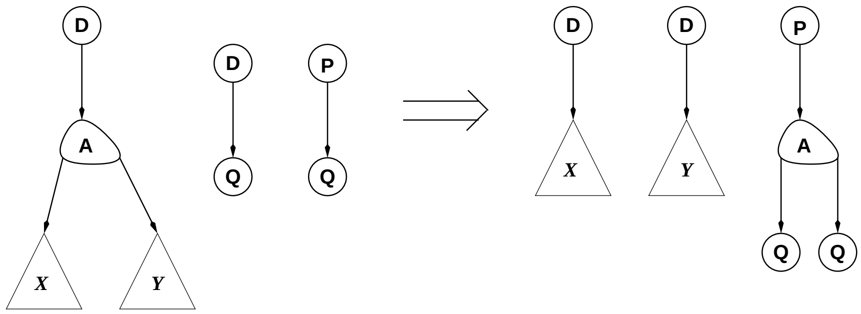

Figure 11: Deletion of an application (A) primitive. This triggers deletion of its two daughters, which may proceed in parallel.

Figure 10 shows the base of the recursive process; deletion of a primitive combinator causes it to be "capped" and released for reuse. The reaction is:

$$\mathsf{D}p + \mathsf{PQ} \longrightarrow \mathsf{P}p + \mathsf{DQ}, \tag{13}$$

where p represents any primitive combinator (e.g., S or K). Deletion of an application (A) primitive triggers the deletion of its daughters (Fig. 11); the reaction is:

$$\mathsf{DA}XY + \mathsf{DQ} + \mathsf{PQ} \longrightarrow \mathsf{D}X + \mathsf{D}Y + \mathsf{PAQ}_2. \tag{14}$$

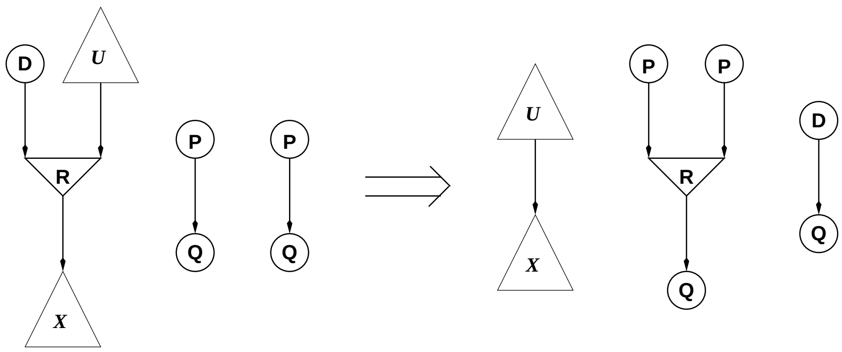

Figure 12: Deletion of a replication in progress. If a deletion catches up with a replication (R primitive), then both the deletion and replication are terminated.

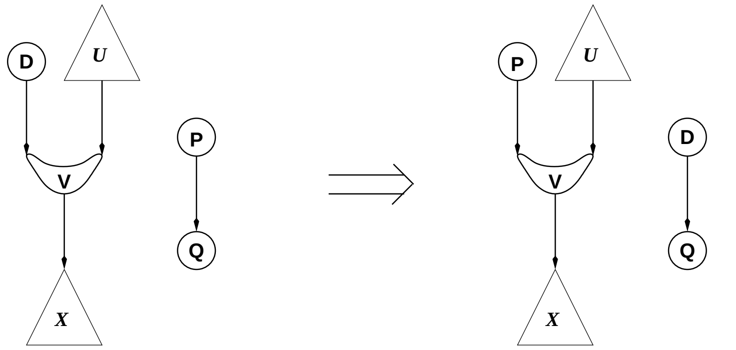

Figure 13: Deletion of one path to a sharing node (V primitive). The deleted path is "capped" with a P primitive (result cap), but the other path is left intact.

The foregoing reactions are sufficient, but there are advantages to considering the interaction of deletion with R and V primitives. If a deletion catches up with a replication in progress, then it should surely cancel the replication (Fig. 12):

$$\mathsf{D}U\mathsf{R}X + 2\mathsf{PQ} \longrightarrow UX + \mathsf{P}_2\mathsf{RQ} + \mathsf{DQ}. \tag{15}$$

It would surely be wasteful to wait for the duplication to complete, and then have to begin the process of disassembling the new copy! Another argument in favor of this reaction comes from observations of functional programs translated into SK combinator trees. The standard translation algorithms generate many instances of the I combinator, which is defined $\mathsf{I} = \mathsf{SKK}$. This means that an identity operation is implemented by replicating and then deleting a copy of the argument:

$$\mathsf{I}X \Longrightarrow \mathsf{SKK}X \Longrightarrow \mathsf{K}X(\mathsf{K}X) \Longrightarrow X.$$

Although some of these pointless replications can be avoided by cleverer translation algorithms, a certain amount of it is an avoidable characteristic of combinatory computation. Therefore we will be better off if these useless replications are terminated before they complete.

Finally, we must consider the effect of deleting a link to a sharing (V) primitive. We could leave it unspecified, in which case it would be reasonable to suppose that the deletion stops when it reaches the V primitive. Alternately, and more neatly, we could specify the D results in the deleted path being "capped" (Fig. 13):

$$\mathsf{D}U\mathsf{V}X + \mathsf{PQ} \longrightarrow \mathsf{P}U\mathsf{V}X + \mathsf{DQ}. \tag{16}$$

(Indeed, the two approaches may not be so different, for D might be usable in place of the result cap P.) On the other hand, when both links to a V complex have been destroyed, it seems reasonable to trigger the deletion of the (previously) shared structure:

$$\mathsf{DPV}X + \mathsf{PQ} \longrightarrow \mathsf{D}X + \mathsf{P}_2\mathsf{VQ}. \tag{17}$$

(This reaction is not included in the specification of deletion in Sec. 6 [Mac02a].)

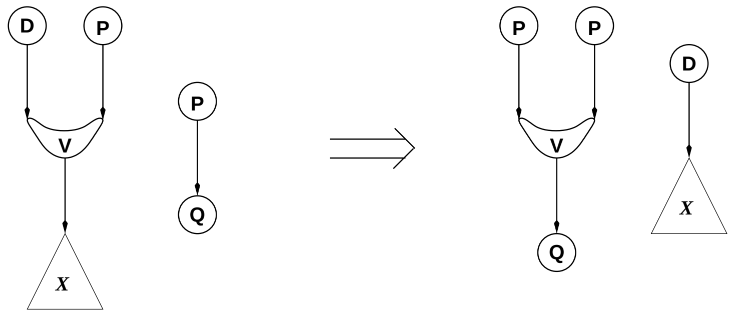

Figure 14: Deletion of only remaining path to a sharing node (V primitive). The shared complex is recursively deleted, and the "capped" V primitive is released as a waste product.

7 Boolean Values, Conditionals, etc.

The fundamental meaning of true and false is the ability to select between two alternatives. Since in combinatory logic, we take data to be (potentially active) functions, we can define true to select the first of two alternatives, and false to select the second:

$$\mathsf{true} XY \Longrightarrow X, \tag{18}$$

$$\mathsf{false} XY \Longrightarrow Y. \tag{19}$$

These are the usual definitions in the lambda calculus and combinatory logic [Bar84, Bur75, Mac84, Mac90, e.g.] They are defined in combinatory logic as follows:

$$\mathsf{true} = \mathsf{K}, \tag{20}$$

$$\mathsf{false} = \mathsf{SK}. \tag{21}$$

The logical operations are easy to define. Since $\mathsf{not} X = X\ \mathsf{false}\ \mathsf{true}$, we can define $\mathsf{not} = \mathsf{C}_{[2]}\mathsf{I}\ \mathsf{false}\ \mathsf{true}$ (see Secs. 4, 30 [Mac02a]). Thus we have (see also Sec. 31 [Mac02a]):

$$\mathsf{not} = \mathsf{C}_{[2]}\mathsf{I}\ \mathsf{false}\ \mathsf{true}, \tag{22}$$

$$\mathsf{and} = \mathsf{C}^{[2]}\mathsf{I}\ \mathsf{false}, \tag{23}$$

$$\mathsf{or} = \mathsf{CI}\ \mathsf{true}. \tag{24}$$

If C is a Boolean-valued expression (one reducing to true or false), then the conditional "if C then X else Y" can be represented by the expression CXY. For greater clarity, we can write it using ALGOL68 notation, $(C \to X \mid Y)$.

$$(C \to X \mid Y) \Longrightarrow CXY. \tag{25}$$

This is not quite the same as a conditional in a programming language, since reduction could proceed in X and Y in parallel, even though one of them will be discarded. This could be a problem if one of the computations in nonterminating, since that computation could consume all the reaction resources, even though it won't be selected. This is especially a problem with recursive function definitions, since typically one branch of the conditional will lead to recursive expansion of the function.

One common method of avoiding this problem is to "delay" execution of the arms of the conditional until one of them is chosen, at which point its execution is "released" [Bur75, Mac90]. Execution of an expression is delayed by "abstracting" [Abd76, CFC58] a dummy formal parameter from it. Then it cannot be reduced until a corresponding dummy actual is provided. These operations are defined:

$$\langle\!\langle E \rangle\!\rangle \implies \lambda x(E), \tag{26}$$

$$\mathsf{force} F \implies FA, \tag{27}$$

$$\mathsf{if}\ C\ \mathsf{then}\ X\ \mathsf{else}\ Y \implies \mathsf{force}(C \to \langle\!\langle X \rangle\!\rangle \mid \langle\!\langle Y \rangle\!\rangle). \tag{28}$$

$\langle\!\langle E \rangle\!\rangle$ means $\lambda x(E)$, where x is any variable that does not occur in E; this delays execution of E by converting it to a single-parameter function. Execution is allowed by providing an actual parameter A, which represents any combinator (the inert N would be a good choice). Thus, $\mathsf{force}\langle\!\langle E \rangle\!\rangle \implies E$. The 'if then else ' is then defined to delay execution of its arms until one of them is chosen.

8 LISP-style Lists

8.1 Representation Based on Triples

There are a number of ways to define LISP-style lists in combinatory logic. One way [Mac84] is to define a "cell" with three fields, cell NFR. If N is true, then F and R are the "first" (car, head) and "rest" (cdr, tail) of the list. If N is false, then cell NFR represents a null list, and the values F and R are irrelevant. This can be accomplished by defining cell so that cell NFR returns a function that, when applied to a "selector" s, returns the selected component of the cell. Therefore, to begin we define three selectors with the properties:

$$\mathsf{1of3}xyz \implies x, \tag{29}$$

$$\mathsf{2of3}xyz \implies y, \tag{30}$$

$$\mathsf{3of3}xyz \implies z. \tag{31}$$

These are easily defined in combinatory logic (see Secs. 24, 30 [Mac02a]):

$$\mathsf{1of3} = \mathsf{C}_{[0]}\mathsf{K}^2 = \mathsf{K}^2, \tag{32}$$

$$\mathsf{2of3} = \mathsf{C}_{[1]}\mathsf{K}^2 = \mathsf{CK}^2, \tag{33}$$

$$\mathsf{3of3} = \mathsf{C}_{[2]}\mathsf{K}^2. \tag{34}$$

cell NFR then is defined to be a function that applies a provided selector s to NFR:

$$\mathsf{cell}\, NFR \implies (\lambda s.sNFR). \tag{35}$$

Thus, cell should satisfy $\mathsf{cell}\,NFRS \Longrightarrow SNFR$, and a suitable definition is (Sec. 30 [Mac02a]):

$$\mathsf{cell} = \mathsf{C}_{[3]}\mathsf{I}. \tag{36}$$

A definition of list-processing operations is then straight-forward:

$$\begin{aligned}
\mathsf{nil}_{\mathrm{M}} &= \mathsf{cell}\,\mathsf{false}\bot\bot, & (37)\\
\mathsf{cons}_{\mathrm{M}} &= \mathsf{cell}\,\mathsf{true}, & (38)\\
\mathsf{nonnull}_{\mathrm{M}} &= \lambda x.(x\mathsf{1of3}) = \mathsf{CI}\ \mathsf{1of3}, & (39)\\
\mathsf{first}_{\mathrm{M}} &= \lambda x.(x\mathsf{2of3}) = \mathsf{CI}\ \mathsf{2of3}, & (40)\\
\mathsf{rest}_{\mathrm{M}} &= \lambda x.(x\mathsf{3of3}) = \mathsf{CI}\ \mathsf{3of3}, & (41)\\
\mathsf{null}_{\mathrm{M}} &= \mathsf{not} \circ \mathsf{nonnull}_{\mathrm{M}}. & (42)
\end{aligned}$$

(The subscripts are to distinguish these definitions from the alternatives considered in Sec. 8.2. $\bot$ is any combinator; N would be a good choice.)

Let's explore the actual combinatory representation of such a list. The "cons cell" resulting from $\mathsf{cons}_{\mathrm{M}}FR$ looks like this:

$$\mathsf{cons}_{\mathrm{M}}FR \Longrightarrow \mathsf{cell}\,\mathsf{true}FR \Longrightarrow \mathsf{cell}\,\mathsf{K}FR \Longrightarrow \mathsf{C}_{[3]}\mathsf{IK}FR. \tag{43}$$

Therefore, let

$$L_{\mathrm{M}} = \mathsf{C}_{[3]}\mathsf{IK}. \tag{44}$$

The representation of an n-element list is:

$$\begin{aligned}
\langle X_1, X_2, \ldots, X_n\rangle &\Longrightarrow \mathsf{cons}_{\mathrm{M}}X_1(\mathsf{cons}_{\mathrm{M}}X_2(\cdots(\mathsf{cons}_{\mathrm{M}}X_n\mathsf{nil}_{\mathrm{M}})\cdots)), & (45)\\
&\Longrightarrow L_{\mathrm{M}}X_1(L_{\mathrm{M}}X_2(\cdots(L_{\mathrm{M}}X_n\mathsf{nil}_{\mathrm{M}})\cdots)), & (46)
\end{aligned}$$

where $\mathsf{nil}_{\mathrm{M}} = \mathsf{C}_{[3]}\mathsf{I}(\mathsf{SK})\mathsf{NN}$. Thus, a list structure is a network as in Fig. 15. We can see that each list element requires two A nodes and an instance of L_{M}, which has size:

$$|L_{\mathrm{M}}| = 24\mathsf{S} + 23\mathsf{K} + 46\mathsf{A} = 93 \text{ total primitives.} \tag{47}$$

(On the measurement of network sizes, see **Introduction** in [Mac02a].) In addition, the list is terminated by a representation of nil, but we can use any complex such that

$$\mathsf{nonnull}\ \mathsf{nil} \Longrightarrow \mathsf{nil}\ \mathsf{1of3} \Longrightarrow \mathsf{false} \Longrightarrow \mathsf{SK}.$$

Thus we can use

$$\begin{aligned}
\mathsf{nil} &= \mathsf{K}(\mathsf{SK}), & (48)\\
|\mathsf{nil}| &= 1\mathsf{S} + 2\mathsf{K} + 2\mathsf{A} = 5 \text{ total.} & (49)
\end{aligned}$$

Hence the size of this list representation is

$$|\langle X_1, \ldots, X_n\rangle| = (24n+1)\mathsf{S} + (23n+2)\mathsf{K} + (48n+2)\mathsf{A} = 95n + 5 \text{ total,} \tag{50}$$

exclusive of the sizes of the X_k.

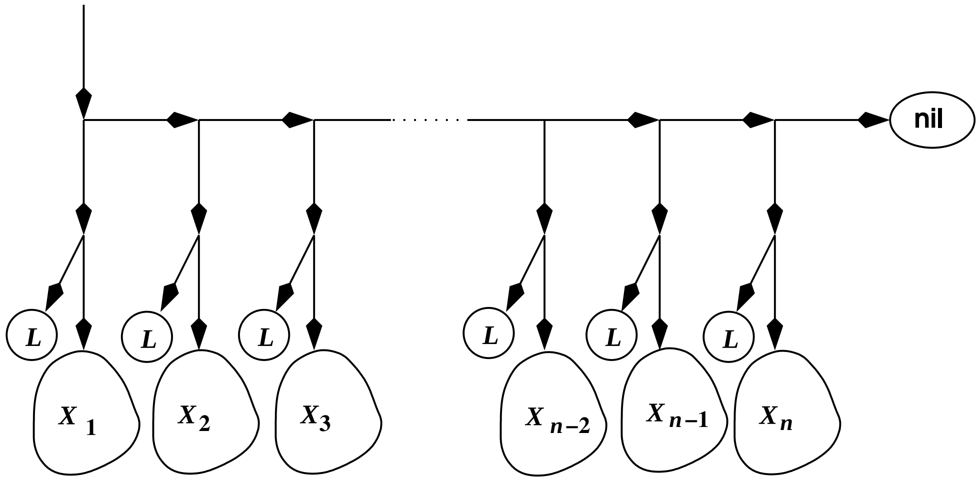

Figure 15: LISP-style list as a combinator complex. The diagram shows the molecular representation of the list $\langle X_1, \ldots, X_n \rangle$. Interior nodes are application (A) primitives.

8.2 Representations Based on Pairs

A second approach defines a "cons cell" as an ordered pair [Bar84, LV97]. Here are typical definitions:

$$\begin{aligned} \mathsf{cons}_{\mathrm{T}} &= \lambda xys.sxy = \mathsf{C}_{[2]}\mathsf{I}, & (51)\\ \mathsf{nil}_{\mathrm{T}} &= \mathsf{K\ true}, & (52)\\ ? &= \mathsf{K(K\ false)}, & (53)\\ \mathsf{null}_{\mathrm{T}} &\Longrightarrow \mathsf{CI?}, & (54)\\ \mathsf{first}_{\mathrm{T}} &\Longrightarrow \mathsf{CI\ true}, & (55)\\ \mathsf{rest}_{\mathrm{T}} &\Longrightarrow \mathsf{CI\ false}. & (56) \end{aligned}$$

The latter two definitions use true and false to select the components of a pair. It's easy to show that

$$\begin{aligned} \mathsf{first}_{\mathrm{T}}(\mathsf{cons}_{\mathrm{T}}XY) &\Longrightarrow X,\\ \mathsf{null}_{\mathrm{T}}(\mathsf{cons}_{\mathrm{T}}XY) &\Longrightarrow (\mathsf{cons}_{\mathrm{T}}XY)? \Longrightarrow \mathsf{false},\\ \mathsf{null}_{\mathrm{T}}\mathsf{nil} &\Longrightarrow \mathsf{true}, \end{aligned}$$

etc. The representation of a list looks the same as in Fig. 15, except that

$$\begin{aligned} L_{\mathrm{T}} &= \mathsf{C}_{[2]}\mathsf{I}, & (57)\\ |L_{\mathrm{T}}| &= 15\mathsf{S} + 14\mathsf{K} + 28\mathsf{A} = 57 \text{ total}, & (58)\\ \mathsf{nil}_{\mathrm{T}} &= \mathsf{K\ true} = \mathsf{KK}, & (59)\\ |\mathsf{nil}_{\mathrm{T}}| &= 0\mathsf{S} + 2\mathsf{K} + 1\mathsf{A} = 3 \text{ total}. & (60) \end{aligned}$$

Hence,

$$|\langle X_1, \ldots, X_n \rangle| = 15n\mathsf{S} + (14n+2)\mathsf{K} + (30n+1)\mathsf{A} = 59n + 3 \text{ total} \tag{61}$$

(exclusive of the $|X_k|$), which is smaller than the representation in triples. However, it is difficult to say whether it matters in molecular terms.

Curry and Feys [CFC58] used a variant of this approach to define pairs:

$$\begin{aligned} \mathsf{cons}_\mathrm{C} &= \lambda xys.[s(\mathsf{K}y)x] = \mathsf{C}(\mathsf{B}(\mathsf{C}_{[2]}\mathsf{I})\mathsf{K}), && (62) \\ \mathsf{nil}_\mathrm{C} &= \mathsf{K}\ \mathsf{true} = \mathsf{KK}, && (63) \\ \mathsf{first}_\mathrm{C} &= \mathsf{CIZ}_0, && (64) \\ \mathsf{rest}_\mathrm{C} &= \mathsf{CIZ}_{n+1}, \text{ for any } n \geq 0. && (65) \end{aligned}$$

(See Sec. 9.3 below on the "Church numerals" Z_k.) With this representation it is easy to show that a list has size

$$|\langle X_1, \ldots, X_n \rangle| = (15n+1)\mathsf{S} + (15n+2)\mathsf{K} + (31n+1)\mathsf{A} = 61n + 3 \text{ total}, \tag{66}$$

exclusive of the $|X_k|$.

8.3 Sequences (n-tuples)

Rather than building up lists by pairs, it is possible to represent them directly; that is, instead of using triples or couples, we use n-tuples [Bar84, p. 134]. Define an n-tuple to be a function that takes a selector and returns the selected element:

$$\langle X_1, \ldots, X_n \rangle = \lambda s(sX_1 \cdots X_n). \tag{67}$$

This is satisfied by the following combinatorial expression:

$$\langle X_1, \ldots, X_n \rangle \Longrightarrow \mathsf{C}_{[n]}\mathsf{I}X_1 \cdots X_n. \tag{68}$$

The ith element of a list is selected by applying it to a selector function sel_i^n, which selects the i element of an n-element sequence. By Eq. 26 [Mac02a],

$$\mathsf{K}^{i-1}FX_1 \cdots X_{i-1} \Longrightarrow F. \tag{69}$$

Also, by Eq. 26 [Mac02a],

$$\mathsf{K}^{n-i}X_i \cdots X_n \Longrightarrow X_i. \tag{70}$$

Therefore, substituting K^{n-i} for F in Eq. 69, we have

$$\mathsf{K}^{i-1}\mathsf{K}^{n-i}X_1 \cdots X_{i-1}X_i \cdots X_n \Longrightarrow \mathsf{K}^{n-i}X_i \cdots X_n \Longrightarrow X_i.$$

Therefore,

$$\mathsf{sel}_i^n = \mathsf{K}^{i-1}\mathsf{K}^{n-i}. \tag{71}$$

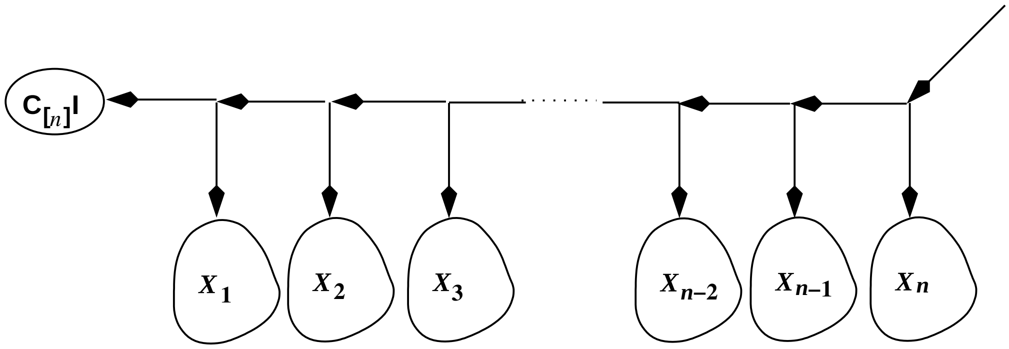

Figure 16: Sequence $\langle X_1, \ldots, X_n \rangle$ represented as combinator structure. Interior nodes represent application (A) primitives.

In contrast to the preceding definitions, the overhead of sequences is very small (Fig. 16):

$$|\langle X_1, \ldots, X_n \rangle| = |\mathsf{C}_{[n]}\mathsf{I}| + n\mathsf{A}, \tag{72}$$

exclusive of the $|X_k|$. However, $|\mathsf{C}^{[n]}| \in \mathcal{O}(n)$, since from Sec. 30 [Mac02a] we can calculate

$$|\mathsf{C}_{[n]}| = (9n - 4)\mathsf{S} + (8n - 4)\mathsf{K} + (17n - 9)\mathsf{A} = 34n - 17 \text{ total.}$$

Hence,

$$|\langle X_1, \ldots, X_n \rangle| = (9n - 3)\mathsf{S} + (8n - 2)\mathsf{K} + (18n - 6)\mathsf{A} = 35n - 11 \text{ total,} \tag{73}$$

exclusive of the $|X_k|$.

8.4 Comparison

The most efficient representation in terms of space is the sequence (n-tuple) representation, but it has the disadvantage that one must know the length n of the sequence. This precludes LISP-style list processing (that is, having a function recur until the end of the list is reached). Also, the elements are indexed with the selector functions sel_i^n, although it would be possible to define them as functions of i, thus allowing computation of subscripts. Therefore, although sequences may be useful for particular purposes, it seems that the common list representation should be based on pairs (cons cells).

9 Numerals

9.1 Unary Numeral System

As in ordinary computation, in programmable intelligent matter we often want to represent nonnegative integers. For example, we might define a complex G such that GMN produces an $M \times N$ grid, given suitable representations of the integers M, N. In electronic computers, integers are represented in binary notation, and of course it is possible to define combinatory programs that use binary numerals, but this does not seem to be the best approach. Rather, the simplest approach is to represent numbers as *unary numerals*; that is, the number N is represented by some structure (e.g., a chain) of size N. The savings in size by using binary numerals (size $\lceil \log_2 N \rceil$) instead of unary numerals is not so great, in molecular terms, and the decoding complexity is significantly greater.

9.2 List Representation

The simplest unary representation of n is as a list of length n; this is in effect what Peano used in his axiomatization of natural numbers. With such a representation it is simple to compute the successor of a number (cons a new element onto it), compute its predecessor (take the rest of the list), or ask if it's 0 (test if it's null). One problem with this approach is its relative inefficiency. Based on our preceding analysis of list representations, the number n will require a network of at least $59n + 3$ primitives.

9.3 Church Numerals

Of course, n may be represented by any structure of size n, and so it may be better to pick a structure that does something useful. This fits better with the blurring of the distinction between program and data in combinatory logic, in which data is often active.

To move toward a more function-oriented view, start by thinking of the number n as a list of length n, $\langle X_1, X_2, \ldots, X_n \rangle$. From this, we are led to consider an argument list of length n, such as $(F X_1 X_2 \cdots X_n)$. This is turn suggests an n-fold functional composition as a representation of n, such as $F_1(F_2(\cdots(F_n X)\cdots))$, or, more simply, $F^n X$ (see Sec. 28 [Mac02a] for this notation). This immediate leads us to the *iterators* or *Church numerals* Z_n (Sec. 23 [Mac02a]), which are defined so that

$$\mathsf{Z}_n F \Longrightarrow F^n.$$

According to Sec. 23 [Mac02a], Z_n is $10n + 7$ in size, which is considerably better than the list representation (see also Fig. 17); more importantly, the Church numerals are immediately useful, without the need of a program to interpret them. Sec. 23 [Mac02a] also shows that it is possible to add, multiply, and exponentiate Church numerals. Predecessor, subtraction, and zero test are not needed often, but when they are, they can be accomplished by converting to the list representation, doing the operation, and converting back to Church numerals, as shown by Barendregt [Bar84, pp. 140–1].

9.4 Representation of Large Numbers

The reader may be worried about the use of unary numerals in the cases where we need to represent large numbers, since $|\mathsf{Z}_n| \in \mathcal{O}(n)$. In these cases we can simple compute the numbers we need by exponentiation. By Eq. 67 [Mac02a], $\mathsf{Z}_{n^m} = \mathsf{Z}_m \mathsf{Z}_n$, so we can represent n^m by $\mathsf{Z}_m \mathsf{Z}_n$, which has size $\mathcal{O}(m + n)$, specifically $10(m + n) + 15$.

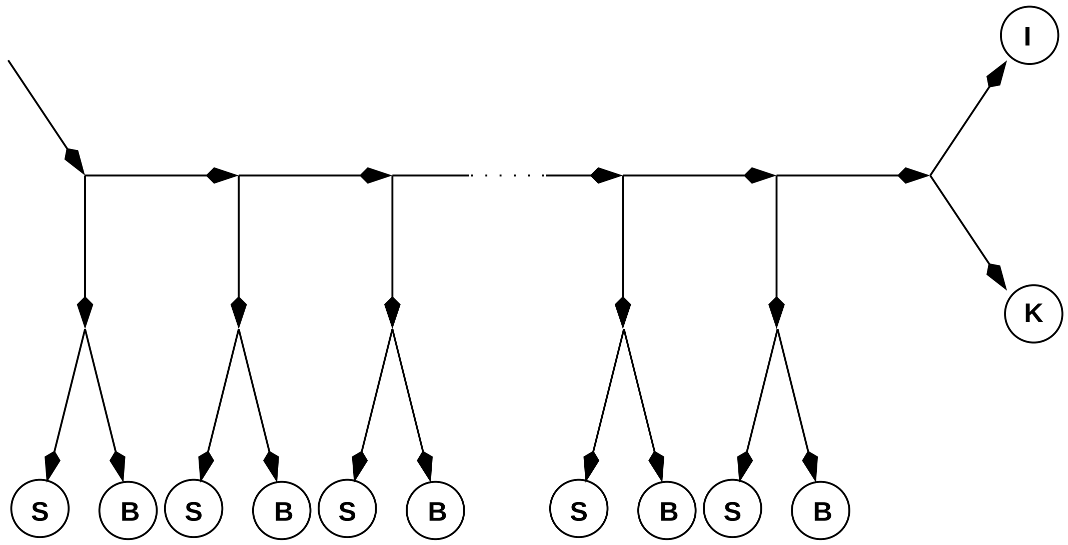

Figure 17: Iterator or Church Numeral represented as combinator structure. The iterator Z_n is represented by a chain of n SB complexes with a terminal KI complex. Interior nodes represent application (A) primitives.

References

[Abd76] S. K. Abdali. An abstraction algorithm for combinatory logic. *Journal of Symbolic Logic*, 41(1):222–224, March 1976.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.

[Bur75] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, Reading, 1975.

[CFC58] H. B. Curry, R. Feys, and W. Craig. *Combinatory Logic, Volume I*. North-Holland, Amsterdam, 1958.

[CR36] A. Church and J. Barkley Rosser. Some properties of conversion. *Trans. American Math. Soc.*, 39:472–482, 1936.

[HO82] C. Hoffman and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[LV97] M. Li and P. M. B. Vitanyi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer-Verlag, New York, second edition, 1997.

[Mac84] Bruce J. MacLennan. Computable real analysis. Technical Report NPS52-84-024, Dept. of Computer Science, Naval Postgraduate School, 1984.

[Mac90] Bruce J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, Reading, 1990.

[Mac02a] Bruce J. MacLennan. Molecular combinator reference manual — UPIM report 2. Technical Report CS-02-489, Dept. of Computer Science, University of Tennessee, Knoxville, 2002. Available at http://www.cs.utk.edu/ ~library/TechReports/ 2002/ut-cs-02-489.ps.

[Mac02b] Bruce J. MacLennan. Universally programmable intelligent matter (exploratory research proposal) — UPIM report 1. Technical Report CS-02-486, Dept. of Computer Science, University of Tennessee, Knoxville, 2002. Available at http://www.cs.utk.edu/ ~library/TechReports/ 2002/ut-cs-02-486.ps.

[Ros73] Barry K. Rosen. Tree manipulation systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, January 1973.