A Structural Approach to Reversible Computation

Samson Abramsky
Oxford University Computing Laboratory

Abstract

Reversibility is a key issue in the interface between computation and physics, and of growing importance as miniaturization progresses towards its physical limits. Most foundational work on reversible computing to date has focussed on simulations of low-level machine models. By contrast, we develop a more structural approach. We show how high-level functional programs can be mapped *compositionally* (*i.e.* in a syntax-directed fashion) into a simple kind of automata which are immediately seen to be reversible. The size of the automaton is linear in the size of the functional term. In mathematical terms, we are building a concrete *model* of functional computation. This construction stems directly from ideas arising in Geometry of Interaction and Linear Logic—but can be understood without any knowledge of these topics. In fact, it serves as an excellent introduction to them. At the same time, an interesting logical delineation between reversible and irreversible forms of computation emerges from our analysis.

1 Introduction

The importance of reversibility in computation, for both foundational and, in the medium term, for practical reasons, is by now well established. We quote from the excellent summary in the introduction to the recent paper by Buhrman, Tromp and Vitányi [18]:

> **Reversible Computation:** R. Landauer [39] has demonstrated that it is only the "logically irreversible" operations in a physical computer that necessarily dissipate energy by generating a corresponding amount of entropy for every bit of information that gets irreversibly erased; the logically reversible operations can in principle be performed dissipation-free. Currently, computations are commonly irreversible, even though the physical devices that execute them are fundamentally reversible. At the basic level, however, matter is governed by classical mechanics and quantum mechanics, which are reversible. This contrast is only possible at the cost of efficiency loss by generating thermal entropy into the environment. With computational device technology rapidly approaching the elementary particle level it has been argued many times that this effect gains in significance to the extent that efficient operation (or operation at all) of future computers requires them to be reversible . . . The mismatch of computing organization and reality will express itself in friction: computers will dissipate a lot of heat unless their mode of operation becomes reversible, possibly quantum mechanical.

The previous approaches of which we are aware (e.g. [41, 16, 17]) proceed by showing that some standard, low-level, irreversible computational model such as Turing machines can be simulated by a reversible version of the same model. Our approach is more "structural". We firstly define a simple model of computation which is directly reversible in a very strong sense—every automaton $\mathcal{A}$ in our model has a "dual" automaton $\mathcal{A}^{\mathsf{op}}$, defined quite trivially from $\mathcal{A}$, whose computations are exactly the time-reversals of the computations of $\mathcal{A}$. We then establish a connection to models of *functional computation*. We will show that our model gives rise to a *combinatory algebra* [31], and derive universality as an easy consequence. This method of establishing universality has potential significance for the important issue of how to *program* reversible computations. To quote from [18] again:

> Currently, almost no algorithms and other programs are designed according to reversible principles ... To write reversible programs by hand is unnatural and difficult. The natural way is to compile irreversible programs to reversible ones.

Our approach can be seen as providing a simple, *compositional* (i.e. "syntax-directed") compilation from high-level functional programs into a reversible model of computation. This offers a new perspective on reversible computing.

Our approach also has conceptual interest in that our constructions, while quite concrete, are based directly on ideas stemming from Linear Logic and Geometry of Interaction [24, 25, 26, 27, 43, 20], and developed in previous work by the present author and a number of colleagues [2, 3, 5, 6, 7, 9, 10]. Our work here can be seen as a concrete manifestation of these more abstract and foundational developments. However, no knowledge of Linear Logic or Geometry of Interaction is required to read the present paper. In fact, it might serve as an introduction to these topics, from a very concrete point of view. At the same time, an interesting logical delineation between reversible and irreversible forms of computation emerges from our analysis.

The paper [21] contains some discussion of a reversible abstract machine for the λ-calculus. However, the approach taken and the objectives are rather different to ours, and there are no clear analogues to our results. In fact, our concrete view of Geometry of Interaction can be traced back via Ian Mackie's thesis [42], written under the present author's supervision, to ideas originating in [2, 3].

2 The Computational Model

We formulate our computational model as a kind of automaton with some simple term-rewriting capabilities. We assume familiarity with the very basic notions of term rewriting, such as may be gleaned from the opening pages of any of the standard introductory accounts [22, 38, 14]. In particular, we shall assume familiarity with the notions of *signature* $\Sigma = (\Sigma_n \mid n \in \omega)$, and of the term algebras T_Σ and $T_\Sigma(X)$, of ground terms, and terms in a set of variables X, respectively. We will work exclusively with *finite* signatures Σ. We also assume familiarity with the notion of *most general unifier*; given terms $t, u \in T_\Sigma(X)$, we write $\mathcal{U}(t,u)\downarrow\sigma$ if $\sigma : X \longrightarrow T_\Sigma(X)$ is the most general unifying substitution of t and u, and $\mathcal{U}(t,u)\uparrow$ if t and u cannot be unified.

We define a *pattern-matching automaton* to be a structure

$$\mathcal{A} \quad = \quad (Q, q_\iota, q_f, R)$$

where Q is a finite set of states, q_ι and q_f are distinguished initial and final states, and $R \subseteq Q \times T_\Sigma(X) \times T_\Sigma(X) \times Q$ is a finite set of *transition rules*, written

$$\begin{array}{rcl} (q_1, r_1) & \to & (s_1, q_1') \\ & \vdots & \\ (q_N, r_N) & \to & (s_N, q_N') \end{array}$$

where $q_i, q_i' \in Q$, $r_i, s_i \in T_\Sigma(X)$, and the variables occurring in s_i are a subset of those occurring in r_i, $1 \leq i \leq N$. We also stipulate that there are no incoming transitions to the initial state, and no outgoing transitions from the final state: $q_\iota \neq q_i'$ and $q_f \neq q_i$, $1 \leq i \leq N$.
A *configuration* of $\mathcal{A}$ is a pair $(q, t) \in Q \times T_\Sigma$ of a state and a ground term. $\mathcal{A}$ induces a relation $\xrightarrow{\mathcal{A}}$ on configurations: $(q, t) \xrightarrow{\mathcal{A}} (q', t')$ iff

$$\exists i \, (q_i = q \ \wedge \ q_i' = q' \ \wedge \ \mathcal{U}(t, r_i){\downarrow}\sigma \ \wedge \ t' = \sigma(s_i)).$$

Note that the "pattern" r_i has to match the whole of the term t. This is akin to the use of pattern-matching in functional programming languages such as SML [44] and Haskell [47], and is the reason for our choice of terminology.

Note that the cost of computing the transition relation $(q, t) \xrightarrow{\mathcal{A}} (q', t')$ is *independent* of the size of the "input" term t.[1] If we are working with a fixed pattern-matching automaton $\mathcal{A}$, this means that the basic computation steps can be performed in constant time and space, indicating that our computational model is at a reasonable level of granularity.

A *computation* over $\mathcal{A}$ starting with an initial ground term $t_0 \in T_\Sigma$ (the *input*) is a sequence

$$(q_\iota, t_0) \xrightarrow{\mathcal{A}} (q_1, t_1) \xrightarrow{\mathcal{A}} \cdots.$$

The computation is *successful* if it terminates in a configuration (q_f, t_k), in which case t_k is the *output*. Thus we can see a pattern-matching automaton as a device for computing relations on ground terms.
We say that a pattern-matching automaton

$$\mathcal{A} \quad = \quad (Q, q_\iota, q_f, R)$$

with

$$R = \{(q_i, r_i) \to (s_i, q_i') \mid 1 \leq i \leq N\}$$

is *orthogonal* if the following conditions hold:

[1] Under the assumption of *left-linearity* (see below) which we shall shortly make, and on the standard assumption made in the algorithmics of unification [14, 22] that the immediate sub-terms of a given term can be accessed in constant time.

Non-ambiguity For each $1 \leq i < j \leq N$, if $q_i = q_j$, then $\mathcal{U}(r_i, r_j)\uparrow$.

Left-linearity For each i, $1 \leq i \leq N$, no variable occurs more than once in r_i.

Note that non-ambiguity is stated in a simpler form than the standard version for term-rewriting systems [14, 22, 38], taking advantage of the fact that we are dealing with the simple case of pattern-matching.

Clearly the effect of non-ambiguity is that computation is *deterministic*: given a configuration (q, t), at most one transition rule is applicable, so that the relation $\xrightarrow{\mathcal{A}}$ is a partial function. Given a pattern matching automaton $\mathcal{A}$ as above, we define $\mathcal{A}^{\mathsf{op}}$ to be

$$(Q, q_f, q_\iota, R^{\mathsf{op}})$$

where

$$R^{\mathsf{op}} = \{(q_i', s_i) \to (r_i, q_i) \mid 1 \leq i \leq N\}$$

We define $\mathcal{A}$ to be *biorthogonal* if both $\mathcal{A}$ and $\mathcal{A}^{\mathsf{op}}$ are orthogonal pattern-matching automata. Note that if $\mathcal{A}$ is a biorthogonal automaton, so is $\mathcal{A}^{\mathsf{op}}$, and $\mathcal{A}^{\mathsf{op}\,\mathsf{op}} = \mathcal{A}$.

It should be clear that computation in biorthogonal automata is reversible in a deterministic, step-by-step fashion. Thus if we have the computation

$$(q_\iota, t_0) \xrightarrow{\mathcal{A}} \cdots \xrightarrow{\mathcal{A}} (q_f, t_n)$$

in the biorthogonal automaton $\mathcal{A}$, then we have the computation

$$(q_f, t_n) \xrightarrow{\mathcal{A}^{\mathsf{op}}} \cdots \xrightarrow{\mathcal{A}^{\mathsf{op}}} (q_\iota, t_0)$$

in the biorthogonal automaton $\mathcal{A}^{\mathsf{op}}$. Note also that biorthogonal automata are *linear* in the sense that, for each rule $(q, r) \to (s, q')$, the same variables occur in r and in s, and moreover each variable which occurs does so exactly once in r and exactly once in s. Thus there is no "duplicating" or "discarding" of sub-terms matched to variables in applying a rule, whether in $\mathcal{A}$ or in $\mathcal{A}^{\mathsf{op}}$.

Orthogonality is a very standard and important condition in term-rewriting systems. However, biorthogonality is a much stronger constraint, and very few of the term-rewriting systems usually considered satisfy this condition. (In fact, the only familiar examples of biorthogonal rewriting systems seem to be associative/commutative rewriting and similar, and these are usually considered as notions for "rewriting modulo" rather than as computational rewriting systems in their own right).

Our model of computation will be the class of biorthogonal pattern-matching automata; from now on, these will be the only automata we shall consider, and we will refer to them simply as "automata". The reader will surely agree that this computational model is quite simple, and seen to be reversible in a very direct and immediate fashion. We will now turn to the task of establishing its universality.

Remark It would have been possible to represent our computational model more or less entirely in terms of standard notions of term rewriting systems. We briefly sketch how this might be done. Given an automaton

$$\mathcal{A} \quad = \quad (Q, q_\iota, q_f, R)$$

we expand the (one-sorted) signature Σ to a signature over three sorts: V (for values), S (for states) and C (for configurations). The operation symbols in Σ have all their arguments and results of sort V; for each state $q \in Q$, there is a corresponding constant of sort S; and there is a binary operation

$$\langle \cdot, \cdot \rangle : S \times V \longrightarrow C.$$

Now the transition rules R turn into a rewriting system in the standard sense; and orthogonality has its standard meaning. We would still need to focus on initial terms of the form $\langle q_\iota, t \rangle$ and normal forms of the form $\langle q_f, t \rangle$, t ground.

Our main reason for using the automaton formulation is that it does expose some salient structure, which will be helpful in defining and understanding the significance of the constructions to follow.

3 Background on Combinatory Logic

In this section, we briefly review some basic material. For further details, see [31].

We recall that combinatory logic is the algebraic theory **CL** given by the signature with one binary operation (application) written as an infix $\_ \cdot \_$, and two constants **S** and **K**, subject to the equations

$$\begin{array}{lcl} \mathbf{K} \cdot x \cdot y & = & x \\ \mathbf{S} \cdot x \cdot y \cdot z & = & x \cdot z \cdot (y \cdot z) \end{array}$$

(application associates to the left, so $x \cdot y \cdot z = (x \cdot y) \cdot z$). Note that we can define $\mathbf{I} \equiv \mathbf{S} \cdot \mathbf{K} \cdot \mathbf{K}$, and verify that $\mathbf{I} \cdot x = x$.

The key fact about the combinators is that they are *functionally complete*, i.e. they can simulate the effect of λ-abstraction. Specifically, we can define bracket abstraction on terms in $T_{\mathbf{CL}}(X)$:

$$\begin{array}{lcll} \lambda^* x.\, M & = & \mathbf{K} \cdot M & (x \notin \mathsf{FV}(M)) \\ \lambda^* x.\, x & = & \mathbf{I} & \\ \lambda^* x.\, M \cdot N & = & \mathbf{S} \cdot (\lambda^* x.\, M) \cdot (\lambda^* x.\, N) & \end{array}$$

Moreover (Theorem 2.15 in [31]):

$$\mathbf{CL} \vdash (\lambda^* x.\, M) \cdot N = M[N/x].$$

The **B** combinator can be defined by bracket abstraction from its defining equation:

$$\mathbf{B} \cdot x \cdot y \cdot z = x \cdot (y \cdot z).$$

The combinatory *Church numerals* are then defined by

$$\bar{n} \equiv (\mathbf{S} \cdot \mathbf{B})^n \cdot (\mathbf{K} \cdot \mathbf{I})$$

where we define
$$a^n \cdot b = a \cdot (a \cdots (a \cdot b) \cdots).$$
A partial function $\phi : \mathbb{N} \rightharpoonup \mathbb{N}$ is *numeralwise represented* by a combinatory term $M \in T_{\mathbf{CL}}$ if for all $n \in \mathbb{N}$, if $\phi(n)$ is defined and equal to m, then
$$\mathbf{CL} \vdash M \cdot \bar{n} = \bar{m}$$
and if $\phi(n)$ is undefined, then $M \cdot \bar{n}$ has no normal form.
The basic result on computational universality of **CL** is then the following (Theorem 4.18 in [31]):

Theorem 3.1 *The partial functions numeralwise representable in* **CL** *are exactly the partial recursive functions.*

4 Linear Combinatory Logic

We shall now present another system of combinatory logic: *Linear Combinatory Logic* [6, 10, 7]. This can be seen as a finer-grained system into which standard combinatory logic, as presented in the previous section, can be interpreted. By exposing some finer structure, Linear Combinatory Logic offers a more accessible and insightful path towards our goal of mapping functional computation into our simple model of reversible computation.

Linear Combinatory Logic can be seen as the combinatory analogue of Linear Logic [24]; the interpretation of standard Combinatory Logic into Linear Combinatory Logic corresponds to the interpretation of Intuitionistic Logic into Linear Logic. Note, however, that the combinatory systems we are considering are type-free and "logic-free" (*i.e.* purely equational).

Definition 4.1 *A* Linear Combinatory Algebra $(A, \cdot, !)$ *consists of the following data:*

- *An applicative structure* $(A, \cdot)$
- *A unary operator* $! : A \to A$
- *Distinguished elements* $\mathbf{B}$, $\mathbf{C}$, $\mathbf{I}$, $\mathbf{K}$, $\mathbf{D}$, δ, $\mathbf{F}$, $\mathbf{W}$ *of* A

satisfying the following identities (we associate $\cdot$ to the left and write $x \cdot !y$ for $x \cdot (\,!(y))$, etc.) for all variables x, y, z ranging over A.

1.	$\mathbf{B} \cdot x \cdot y \cdot z$	$=$	$x \cdot (y \cdot z)$	*Composition/Cut*
2.	$\mathbf{C} \cdot x \cdot y \cdot z$	$=$	$(x \cdot z) \cdot y$	*Exchange*
3.	$\mathbf{I} \cdot x$	$=$	x	*Identity*
4.	$\mathbf{K} \cdot x \cdot !y$	$=$	x	*Weakening*
5.	$\mathbf{D} \cdot !x$	$=$	x	*Dereliction*
6.	$\delta \cdot !x$	$=$	$!!x$	*Comultiplication*
7.	$\mathbf{F} \cdot !x \cdot !y$	$=$	$!(x \cdot y)$	*Monoidal Functoriality*
8.	$\mathbf{W} \cdot x \cdot !y$	$=$	$x \cdot !y \cdot !y$	*Contraction*

The notion of LCA corresponds to a Hilbert style axiomatization of the $\{!, \multimap\}$ fragment of linear logic [6, 13, 49]. The *principal types* of the combinators correspond to the axiom schemes which they name. They can be computed by a Hindley-Milner style algorithm [32] from the above equations:

$$
\begin{array}{rlcl}
1. & \mathbf{B} & : & (\beta \multimap \gamma) \multimap (\alpha \multimap \beta) \multimap \alpha \multimap \gamma \\
2. & \mathbf{C} & : & (\alpha \multimap \beta \multimap \gamma) \multimap (\beta \multimap \alpha \multimap \gamma) \\
3. & \mathbf{I} & : & \alpha \multimap \alpha \\
4. & \mathbf{K} & : & \alpha \multimap !\beta \multimap \alpha \\
5. & \mathbf{D} & : & !\alpha \multimap \alpha \\
6. & \delta & : & !\alpha \multimap !!\alpha \\
7. & \mathbf{F} & : & !(\alpha \multimap \beta) \multimap !\alpha \multimap !\beta \\
8. & \mathbf{W} & : & (!\alpha \multimap !\alpha \multimap \beta) \multimap !\alpha \multimap \beta
\end{array}
$$

Here $\multimap$ is a *linear function type* (linearity means that the argument is used exactly once), and $!\alpha$ allows arbitrary copying of an object of type α.

A *Standard Combinatory Algebra* consists of a pair $(A, \cdot_s)$ where A is a nonempty set and $\cdot_s$ is a binary operation on A, together with distinguished elements $\mathbf{B}_s, \mathbf{C}_s, \mathbf{I}_s, \mathbf{K}_s$, and $\mathbf{W}_s$ of A, satisfying the following identities for all x, y, z ranging over A:

$$
\begin{array}{rlcl}
1. & \mathbf{B}_s \cdot_s x \cdot_s y \cdot_s z & = & x \cdot_s (y \cdot_s z) \\
2. & \mathbf{C}_s \cdot_s x \cdot_s y \cdot_s z & = & (x \cdot_s z) \cdot_s y \\
3. & \mathbf{I}_s \cdot_s x & = & x \\
4. & \mathbf{K}_s \cdot_s x \cdot_s y & = & x \\
5. & \mathbf{W}_s \cdot_s x \cdot_s y & = & x \cdot_s y \cdot_s y
\end{array}
$$

Note that this is equivalent to the more familiar definition of **SK**-combinatory algebra as given in the previous section. In particular, $\mathbf{S}_s$ can be defined from $\mathbf{B}_s$, $\mathbf{C}_s$, $\mathbf{I}_s$ and $\mathbf{W}_s$ [15, 32]. Let $(A, \cdot, !)$ be a linear combinatory algebra. We define a binary operation $\cdot_s$ on A as follows: for $a, b \in A$, $a \cdot_s b \equiv a \cdot !b$. We define $\mathbf{D}' \equiv \mathbf{C} \cdot (\mathbf{B} \cdot \mathbf{B} \cdot \mathbf{I}) \cdot (\mathbf{B} \cdot \mathbf{D} \cdot \mathbf{I})$. Note that

$$\mathbf{D}' \cdot x \cdot !y = x \cdot y.$$

Now consider the following elements of A.

$$
\begin{array}{rlcl}
1. & \mathbf{B}_s & \equiv & \mathbf{C} \cdot (\mathbf{B} \cdot (\mathbf{B} \cdot \mathbf{B} \cdot \mathbf{B}) \cdot (\mathbf{D}' \cdot \mathbf{I})) \cdot (\mathbf{C} \cdot ((\mathbf{B} \cdot \mathbf{B}) \cdot \mathbf{F}) \cdot \delta) \\
2. & \mathbf{C}_s & \equiv & \mathbf{D}' \cdot \mathbf{C} \\
3. & \mathbf{I}_s & \equiv & \mathbf{D}' \cdot \mathbf{I} \\
4. & \mathbf{K}_s & \equiv & \mathbf{D}' \cdot \mathbf{K} \\
5. & \mathbf{W}_s & \equiv & \mathbf{D}' \cdot \mathbf{W}
\end{array}
$$

Theorem 4.1 *Let $(A, \cdot, !)$ be a linear combinatory algebra. Then $(A, \cdot_s)$ with $\cdot_s$ and the elements $\mathbf{B}_s, \mathbf{C}_s, \mathbf{I}_s, \mathbf{K}_s, \mathbf{W}_s$ as defined above is a standard combinatory algebra.*

Finally, we mention a special case which will arise in our reversible model. An *Affine Combinatory Algebra* is a Linear Combinatory Algebra such that the **K** combinator satisfies the stronger equation

$$\mathbf{K} \cdot x \cdot y = x.$$

Note that in this case we can *define* the identity combinator: $\mathbf{I} \equiv \mathbf{C} \cdot \mathbf{K} \cdot \mathbf{K}$.

5 The Affine Combinatory Algebras $\mathcal{I}$ and $\mathcal{P}$

We fix the following signature Σ for the remainder of this paper.

$$\begin{array}{lcl}\Sigma_0 & = & \{\varepsilon\} \\ \Sigma_1 & = & \{l, r\} \\ \Sigma_2 & = & \{p\} \\ \Sigma_n & = & \varnothing, \quad n > 2.\end{array}$$

We shall discuss minimal requirements on the signature in Section 6.4.
We write $\mathcal{I}$ for the set of all partial injective functions on T_Σ.

5.1 Operations on $\mathcal{I}$

5.1.1 Replication

$$!f = \{(p(t, u), p(t, v)) \mid t \in T_\Sigma \ \wedge \ (u, v) \in f\}$$

5.1.2 Linear Application

$$\mathsf{LApp}(f, g) = f_{rr} \cup f_{rl}; g; (f_{ll}; g)^*; f_{lr}$$

where

$$f_{ij} = \{(u, v) \mid (i(u), j(v)) \in f\} \quad (i, j \in \{l, r\})$$

and we use the operations of relational algebra (union, composition, and reflexive, transitive closure).

The idea is that terms of the form $r(t)$ correspond to interactions between the functional process represented by f and its environment, while terms of the form $l(t)$ correspond to interactions with its argument, namely the functional process represented by g. This is *linear* application because the function interacts with one copy of its argument, whose state changes as the function interacts with it; "fresh" copies of the argument are not necessarily available as the computation proceeds. The purpose of the replication operation described previously is precisely to make the argument copyable, using the first argument of the constructor p to "tag" different copies.

The "flow of control" in linear application is indicated by the following diagram:

$$\begin{tikzcd}
\mathsf{in} \arrow[r] & \bullet \arrow[r, "f_{rr}"] \arrow[d, "f_{rl}"'] & \bullet \arrow[r] & \mathsf{out} \\
& \bullet \arrow[r, shift left, "g"] & \bullet \arrow[u, "f_{lr}"'] \arrow[l, shift left, "f_{ll}"]
\end{tikzcd}$$

Thus the function f will either respond immediately to a request from the environment without consulting its argument (f_{rr}), or it will send a "message" to its argument (f_{rl}), which initiates a dialogue between f and g (f_{ll} and g), which ends with f despatching a response to the environment (f_{lr}). This protocol is mediated by the top-level constructors l and r, which are used (and consumed) by the operation of Linear Application.

5.2 Partial Involutions

Note that $f \in \mathcal{I} \Rightarrow f^{\mathsf{op}} \in \mathcal{I}$, where f^{op} is the relational converse of f. We say that $f \in \mathcal{I}$ is a *partial involution* if $f^{\mathsf{op}} = f$. We write $\mathcal{P}$ for the set of partial involutions.

Proposition 5.1 *Partial involutions are closed under replication and linear application.*

PROOF It is immediate that partial involutions are closed under replication. Suppose that f and g are partial involutions, and that $\mathsf{LApp}(f,g)(u) = v$. We must show that $\mathsf{LApp}(f,g)(v) = u$. There are two cases.
Case 1: $f(r(u)) = r(v)$, in which case $f(r(v)) = r(u)$, and $\mathsf{LApp}(f,g)(v) = u$ as required.
Case 2: for some $w_1, \ldots, w_k$, $k \geq 0$,

$$f(r(u)) = l(w_1), g(w_1) = w_2, f(l(w_2)) = l(w_3), g(w_3) = w_4, \ldots, f(l(w_k)) = l(w_{k+1}),$$
$$g(w_{k+1}) = w_{k+2}, f(l(w_{k+2}) = r(v).$$

Since f and g are involutions, this implies

$$f(r(v)) = l(w_{k+2}), g(w_{k+2}) = w_{k+1}, f(l(w_{k+1})) = l(w_k), \ldots, g(w_4) = w_3, f(l(w_3)) = l(w_2),$$
$$g(w_2) = w_1, f(l(w_1) = r(u),$$

and hence $\mathsf{LApp}(f,g)(v) = u$ as required. $\square$

5.3 Realizing the linear combinators by partial involutions

A partial involution $f \in \mathcal{P}$ is *finitely describable* if there is a finite relation $R \subseteq T_\Sigma(X) \times T_\Sigma(X)$ such that the graph of f is the symmetric closure of

$$\{(\sigma(t), \sigma(u)) \mid \sigma : X \longrightarrow T_\Sigma, \ (t,u) \in R\}.$$

Here $\sigma : X \longrightarrow T_\Sigma$ ranges over *ground substitutions*.

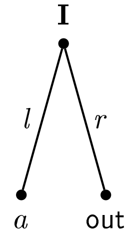
We write $t \leftrightarrow u$ when (t,u) is in the finite description of a partial involution, and refer to such expressions as *rules*.
For each linear combinator κ, we shall give a finite description specifying a partial involution f_κ.

5.3.1 The identity combinator I

As a first, very simple case, consider the identity combinator $\mathbf{I}$, with the defining equation

$$\mathbf{I} \cdot a = a.$$

We can picture the $\mathbf{I}$ combinator, which should evidently be applied to one argument a to achieve its intended effect, thus:



Here the tree represents the way the applicative structure is encoded into the constructors l, r, as reflected in the definition of LApp. Thus when $\mathbf{I}$ is applied to an argument a, the l-branch will be connected to a, while the r-branch will be connected to the output. The equation $\mathbf{I} \cdot a = a$ means that we should have *the same information* at the leaves a and out of the tree. This can be achieved by the rule

$$l(x) \leftrightarrow r(x)$$

and this yields the definition of the automaton for $\mathbf{I}$.
Now we can show that for any partial involution g, we indeed have

$$\mathsf{LApp}(f_{\mathbf{I}}, g) = g.$$

Indeed, for any input t

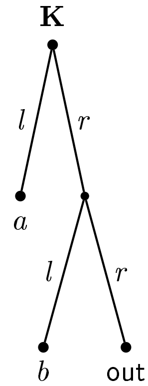
$$\frac{r(t) \overset{f_{\mathbf{I}}}{\longmapsto} l(t) \quad t \overset{g}{\longmapsto} u \quad r(u) \overset{f_{\mathbf{I}}}{\longmapsto} l(u)}{t \overset{\mathsf{LApp}(f_{\mathbf{I}},g)}{\longmapsto} u}$$

5.3.2 The constant combinator K

Next we consider the combinator $\mathbf{K}$, with the defining equation

$$\mathbf{K} \cdot a \cdot b = a.$$

We have the tree diagram



Note that the r branch from the root represents the site of interaction with the environment after the combinator has been applied to one argument. The branch $r(l(\cdots))$ represents interaction with a second argument; while $r(r(\cdots))$ represents the "result" of the application $\mathbf{K} \cdot a \cdot b$.

The defining equation means that we need to make the information at out equal to that at a. This can be accomplished by the rule

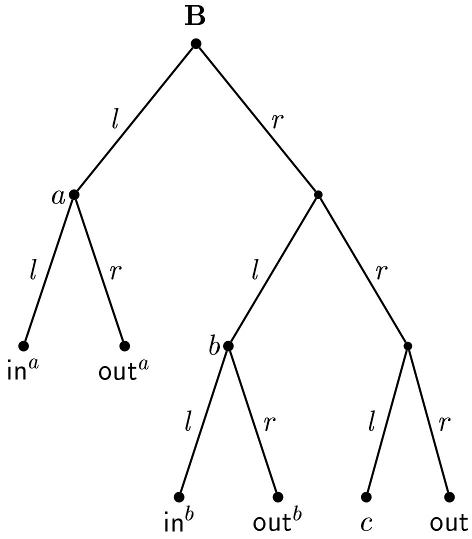
$$l(x) \leftrightarrow r(r(x)).$$

Note that the second argument (b) does not get accessed by this rule, corresponding to the fact that $\mathbf{K}$ is the combinatory equivalent of Weakening (*i.e.* discarding an argument).

5.3.3 The bracketing combinator B

We now turn to a more complex example, the 'bracketing' combinator $\mathbf{B}$, with the defining equation

$$\mathbf{B} \cdot a \cdot b \cdot c = a \cdot (b \cdot c).$$



Here, the arguments a and b themselves have some applicative structure used in the defining equation: a is applied to the result of applying b to c. This means that the automaton realizing $\mathbf{B}$ must access the argument and result positions of a and b, as shown in the tree diagram.

The requirement that the output out of $\mathbf{B}$ should be connected to the output out^a of a translates into the following rule:

$$r(r(r(x))) \leftrightarrow l(r(x)).$$

Similarly, the output out^b of b must be connected to in^a, leading to the rule:

$$l(l(x)) \leftrightarrow r(l(r(x))).$$

Finally, c must be connected to in^b, leading to the rule:

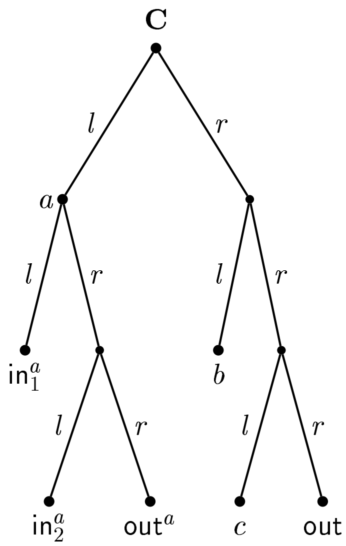
$$r(l(l(x)) \leftrightarrow r(r(l(x))).$$

5.3.4 The commutation combinator C

The $\mathbf{C}$ combinator can be analyzed in a similar fashion. The defining equation is

$$\mathbf{C} \cdot a \cdot b \cdot c = a \cdot c \cdot b.$$

We have the tree diagram



We need to connect b to in_2^a, c to in_1^a, (this inversion of the left-to-right ordering corresponds to the commutative character of this combinator), and out to out^a. We obtain the following set of rules:

$$\begin{array}{lcl} l(l(x)) & \leftrightarrow & r(r(l(x))) \\ l(r(l(x))) & \leftrightarrow & r(l(x))) \\ l(r(r(x))) & \leftrightarrow & r(r(r(x))) \end{array}$$

Note at this point that linear combinatory completeness already yields something rather striking in these terms; that all patterns of accessing arguments and results, with arbitrarily nested (linear) applicative stucture, can be generated by just the above combinators under linear application.

Note that at the multiplicative level, we only need unary operators in the term algebra. To deal with the exponential !, a binary constructor is needed. Note that, as a consequence of the way the replication operator is defined, in an expression of the form $a \cdot !b$, terms at the argument position of a will have the form $p(x, y)$.

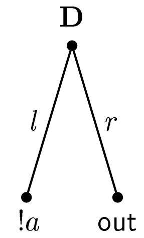
5.3.5 The dereliction combinator D

We start with the dereliction combinator $\mathbf{D}$, with defining equation

$$\mathbf{D} \cdot !a = a.$$

Notice that the combinator expects an argument of a certain form, namely $!a$ (and the equational rule will only "fire" if it has that form).

We have the tree



We need to connect the output to *one copy* of the input. We use the constant ϵ to pick out this copy, and obtain the rule:

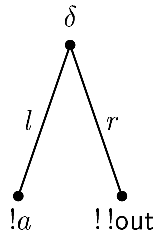
$$l(p(\epsilon, x)) \leftrightarrow r(x).$$

5.3.6 The comultiplication combinator δ

For the comultiplication operator, we have the equation

$$\delta \cdot !a = !\,!a$$

and the tree



Note that a typical pattern at the output will have the form

$$r(p(x, p(y, z)))$$

while a typical pattern at the input has the form

$$l(p(x', y')).$$

The combinator cannot control the shape of the sub-term at y', so we cannot simply unify the two patterns. However, because of the nature of the replication operator, we can impose whatever structure we like on the "copy tag" x', in the knowledge that this will not be changed by the argument $!a$ to which the combinator will be applied. Hence we can match these two patterns up, using the fact that the term algebra T_Σ allows arbitrary nesting of constructors, so that we can write a pattern for the input as

$$l(p(p(x, y), z)).$$

Thus we obtain the rule

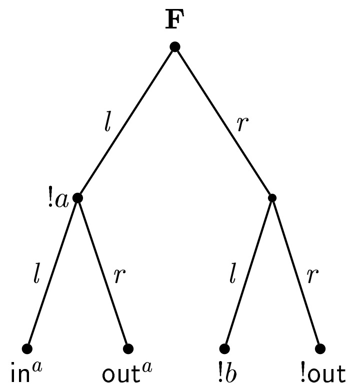
$$l(p(p(x,y),z)) \leftrightarrow r(p(x,p(y,z))).$$

Note that this rule embodies an "associativity isomorphism for pairing", although of course in the free term algebra T_Σ the constructor p is certainly not associative. In the same vein, we can see the rule for the dereliction combinator $\mathbf{D}$ as expressing a "unit isomorphism" for p, with ϵ as the unit element.

5.3.7 The functional distribution combinator F

The combinator $\mathbf{F}$ with equation

$$\mathbf{F} \cdot !a \cdot !b = !(a \cdot b).$$



$\mathbf{F}$ expresses 'closed functoriality' of ! with respect to the linear hom $\multimap$. Concretely, we must move the application of a to b inside the !, which is achieved by commuting the constructors l, r and p. Thus we connect out^a to $!\mathsf{out}$:

$$l(p(x,r(y))) \leftrightarrow r(r(p(x,y)))$$

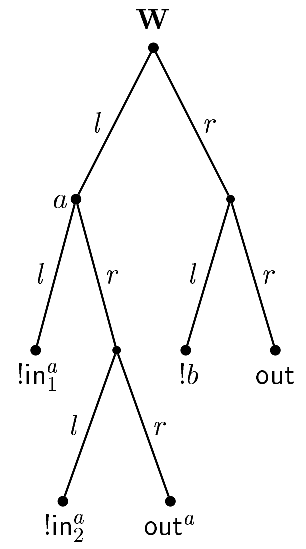
and in^a to $!b$:

$$l(p(x,l(y))) \leftrightarrow r(l(p(x,y))).$$

5.3.8 The duplication combinator W

Finally, we consider the duplication combinator $\mathbf{W}$:

$$\mathbf{W} \cdot a \cdot !b = a \cdot !b \cdot !b.$$



We must connect out and out^a:

$$r(r(x)) \leftrightarrow l(r(r(x))).$$

We also need to connect $!b$ *both* to in_1^a *and* to in_2^a. We do this by using the copy-tag field of $!b$ to split its' address space into two, using the constructors l and r. This tag tells us whether a given copy of $!b$ should be connected to the first (l) or second (r) input of a. Thus we obtain the rules:

$$\begin{array}{rcl} l(l(p(x,y))) & \leftrightarrow & r(l(p(l(x),y))) \\ l(r(l(p(x,y)))) & \leftrightarrow & r(l(p(r(x),y))) \end{array}$$

5.4 The affine combinatory algebras $\mathcal{I}$ and $\mathcal{P}$

Theorem 5.1 $(\mathcal{I}, \cdot, !, f_{\mathbf{B}}, f_{\mathbf{C}}, f_{\mathbf{K}}, f_{\mathbf{D}}, f_{\delta}, f_{\mathbf{F}}, f_{\mathbf{W}})$ *is an affine combinatory algebra, with subalgebra* $\mathcal{P}$.

This theorem is a variation on the results established in [5, 6, 9, 7, 10]; see in particular [10, Propositions 4.2, 5.2], and the combinatory algebra of partial involutions studied in [7]. The ideas on which this construction is based stem from Linear Logic [24, 27] and Geometry of Interaction [25, 26], in the form developed by the present author and a number of colleagues [2, 3, 5, 6, 9, 7, 10].

Once again, combinatory completeness tells us that from this limited stock of combinators, *all* definable patterns of application can be expressed; moreover, we have a universal model of computation.

6 Automatic Combinators

As we have already seen, a pattern-matching automaton $\mathcal{A}$ can be seen as a device for computing a relation on ground terms. The relation $R_{\mathcal{A}} \subseteq T_\Sigma \times T_\Sigma$ is the set of all pairs (t, t') such that there is a computation

$$(q_\iota, t) \xrightarrow{\mathcal{A}}{}^{*} (q_f, t').$$

In the case of a biorthogonal automaton $\mathcal{A}$, the relation $R_{\mathcal{A}}$ is in fact a *partial injective function*, which we write $f_{\mathcal{A}}$. Note that $f_{\mathcal{A}^{\mathsf{op}}} = f_{\mathcal{A}}^{\mathsf{op}}$, the converse of $f_{\mathcal{A}}$, which is also a partial injective function. In the previous section, we defined a linear combinatory algebra $\mathcal{P}$ based on the set of partial involutions on T_Σ. We now want to define a subalgebra of $\mathcal{P}$ consisting of those partial involutions "realized" or "implemented" by a biorthogonal automaton. We refer to such combinators as "Automatic", by analogy with Automatic groups [23], structures [36] and sequences [11].

6.1 Operations on Automata

6.1.1 Replication

Given an automaton $\mathcal{A} = (Q, q_\iota, q_f, R)$, let x be a variable not appearing in any rule in R. We define

$$!\mathcal{A} = (Q, q_\iota, q_f, \,!R)$$

where $!R$ is defined as

$$\{(q, p(x, r)) \to (p(x, s), q') \mid (q, r) \to (s, q') \in R\}.$$

Note that the condition on x is necessary to ensure the linearity of $!R$. The biorthogonality of $!\mathcal{A}$ is easily verified.

6.1.2 Linear Application

See Figure 1. Here $Q \uplus P$ is the disjoint union of Q and P (we simply assume that Q and P have been relabelled if necessary to be disjoint).
The key result we need is the following.

Proposition 6.1 *(i)* $!f_{\mathcal{A}} = f_{!\mathcal{A}}$.
(ii) $\mathsf{LApp}(f_{\mathcal{A}}, f_{\mathcal{B}}) = f_{\mathsf{LApp}(\mathcal{A},\mathcal{B})}$.

Proof (i) $!f_{\mathcal{A}}(p(t,u)) = p(t,v)$ iff $f_{\mathcal{A}}(u) = v$ iff $u \xrightarrow{\mathcal{A}}{}^{*} v$ iff $p(t,u) \xrightarrow{!\mathcal{A}}{}^{*} p(t,v)$.
(ii) Let $\mathcal{C} = \mathsf{LApp}(\mathcal{A},\mathcal{B})$. Suppose $\mathsf{LApp}(f_{\mathcal{A}}, f_{\mathcal{B}})(t) = u$. Then either $f_{rr}(t) = u$, or $f_{rl}(t) = v$, $g(v) = w_1$, $f_{ll}(w_1) = w_2$, $g(w_2) = w_3$, $\ldots$, $f_{ll}(w_k) = w_{k+1}$, $g(w_{k+1}) = w_{k+2}$, $f_{lr}(w_{k+2}) = u$. In the first case, $(q_\iota, r(t)) \xrightarrow{\mathcal{A}}{}^{*} (q_f, r(u))$, and hence $(q_\iota, t) \xrightarrow{\mathcal{C}}{}^{*} (q_f, u)$. In the latter case, $(q_\iota, r(t)) \xrightarrow{\mathcal{A}}{}^{*} (q_f, l(v))$, $(p_\iota, v) \xrightarrow{\mathcal{B}}{}^{*} (p_f, w_1)$, $(q_\iota, l(w_1)) \xrightarrow{\mathcal{A}}{}^{*} (q_f, l(w_2))$, $(p_\iota, w_2) \xrightarrow{\mathcal{B}}{}^{*} (p_f, w_3)$, $\ldots$, $(q_\iota, l(w_k)) \xrightarrow{\mathcal{A}}{}^{*} (q_f, l(w_{k+1}))$, $(p_\iota, w_{k+1}) \xrightarrow{\mathcal{B}}{}^{*} (p_f, w_{k+2})$, $(q_\iota, l(w_{k+2})) \xrightarrow{\mathcal{A}}{}^{*} (q_f, r(u))$, and hence again $(q_\iota, t) \xrightarrow{\mathcal{C}}{}^{*} (q_f, u)$. Thus $\mathsf{LApp}(f_{\mathcal{A}}, f_{\mathcal{B}}) \subseteq f_{\mathsf{LApp}(\mathcal{A},\mathcal{B})}$. The converse inclusion is proved similarly. $\square$

$$\mathcal{A} = (Q, q_\iota, q_f, R) \qquad\qquad \mathcal{B} = (P, p_\iota, p_f, S)$$

$$\mathsf{LApp}(\mathcal{A}, \mathcal{B}) = (Q \uplus P, q_\iota, q_f, T)$$

$$T = \bigcup_{j,k \in \{l,r,i\}} R_{jk} \ \cup \ S$$

$$\begin{array}{lcl}
Q^{\mathsf{int}} & = & Q \setminus \{q_\iota, q_f\} \\
R_{rr} & = & \{(q_\iota, u) \to (v, q_f) \mid (q_\iota, r(u)) \to (r(v), q_f) \in R\} \\
R_{rl} & = & \{(q_\iota, u) \to (v, p_\iota) \mid (q_\iota, r(u)) \to (l(v), q_f) \in R\} \\
R_{ll} & = & \{(p_f, u) \to (v, p_\iota) \mid (q_\iota, l(u)) \to (l(v), q_f) \in R\} \\
R_{lr} & = & \{(p_f, u) \to (v, q_f) \mid (q_\iota, l(u)) \to (r(v), q_f) \in R\} \\
R_{ii} & = & \{(q, u) \to (v, q') \in R \mid q, q' \in Q^{\mathsf{int}}\} \\
R_{ri} & = & \{(q_\iota, u) \to (v, q) \mid (q_\iota, r(u)) \to (v, q) \in R, q \in Q^{\mathsf{int}}\} \\
R_{li} & = & \{(p_f, u) \to (v, q) \mid (q_\iota, l(u)) \to (v, q) \in R, q \in Q^{\mathsf{int}}\} \\
R_{il} & = & \{(q, u) \to (v, p_\iota) \mid (q, u) \to (l(v), q_f) \in R, q \in Q^{\mathsf{int}}\} \\
R_{ir} & = & \{(q, u) \to (v, q_f) \mid (q, u) \to (r(v), q_f) \in R, q \in Q^{\mathsf{int}}\}
\end{array}$$

Figure 1: Linear Application

6.2 Finitely describable partial involutions are Automatic

Now suppose we are given a finite description S of a partial involution f. We define a corresponding automaton $\mathcal{A}$:

$$\mathcal{A} = (\{q_\iota, q_f\}, q_\iota, q_f, R)$$

where

$$R = \bigcup_{(t,u) \in S} \{(q_\iota, t) \to (u, q_f), \ \ (q_\iota, u) \to (t, q_f)\}.$$

It is immediate that $f_{\mathcal{A}} = f$.

Note that $\mathcal{A}$ has no internal states, and all its rules are of the above special form. These features are typical of the automata corresponding to *normal forms* in our interpretation of functional computation.

6.3 The Automatic Universe

The results of the previous two sections yield the following Theorem as an immediate consequence.

Theorem 6.1 *$\mathcal{R}$ is an affine combinatory sub-algebra of $\mathcal{I}$, where the carrier of $\mathcal{R}$ is the set of all $f_{\mathcal{A}}$ for biorthogonal automata $\mathcal{A}$. Moreover, $\mathcal{S} = \mathcal{P} \cap \mathcal{R}$ is an affine combinatory sub-algebra of $\mathcal{R}$.*

Thus we obtain a subalgebra $\mathcal{S}$ of $\mathcal{R}$, of partial involutions realized by biorthgonal automata; and even these very simple behaviours are computationally universal. Partial involutions can be seen as "copy-cat strategies" [3].

6.4 Minimal requirements on Σ

We now pause briefly to consider our choice of the particular signature Σ. We could in fact eliminate the unary operators l and r in favour of two constants, say a and b, and use the representation

$$\begin{array}{lcl} l(t) & \equiv & p(a,t) \\ r(t) & \equiv & p(b,t) \\ p(t,u) & \equiv & p(\varepsilon, p(t,u)). \end{array}$$

We can in turn eliminate a and b, e.g. by the definitions

$$a \equiv p(\varepsilon,\varepsilon) \qquad b \equiv p(p(\varepsilon,\varepsilon),\varepsilon).$$

So one binary operation and one constant—i.e. the pure theory of binary trees—would suffice.

On the other hand, if our signature only contains unary operators and constants, then pattern-matching automata can be simulated by ordinary automata with one stack, and hence are not computationally universal [45].

This restricted situation is still of interest. It suffices to interpret **BCK**-algebras, and hence the *affine* λ-calculus [32]. Recall that the **B** and **C** combinators have the defining equations

$$\begin{array}{l} \mathbf{B} \cdot x \cdot y \cdot z = x \cdot (y \cdot z) \\ \mathbf{C} \cdot x \cdot y \cdot z = x \cdot z \cdot y \end{array}$$

and that **BCK**-algebras admit bracket abstraction for the affine λ-calculus, which is subject to the constraint that applications $M \cdot N$ can only be formed if no variable occurs free in both M and N. The affine λ-calculus is strongly normalizing in a number of steps linear in the size of the initial term, since β-reduction strictly decreases the size of the term.

We build a **BCK**-algebra over automata by using Linear instead of standard application, and defining automata for the combinators **B**, **C** and **K** without using the binary operation symbol p. For reference, we give the set of transition rules for each of these automata:
$R_{\mathbf{K}}$ (linear version):

$$r(r(x)) \leftrightarrow l(x)$$

$R_{\mathbf{B}}$:

$$\begin{array}{lcl} l(r(x)) & \leftrightarrow & r(r(r(x))) \\ l(l(x)) & \leftrightarrow & r(l(r(x))) \\ r(l(l(x))) & \leftrightarrow & r(r(l(x))) \end{array}$$

$R_{\mathbf{C}}$:

$$\begin{array}{lcl} l(l(x)) & \leftrightarrow & r(r(l(x))) \\ l(r(l(x))) & \leftrightarrow & r(l(x))) \\ l(r(r(x))) & \leftrightarrow & r(r(r(x))) \end{array}$$

Note that, since only unary operators appear in the signature, these automata can be seen as performing *prefix string rewriting* [37].

7 Compiling functional programs into reversible computations

Recall that the pure λ-calculus is rich enough to represent data-types such as integers, booleans, pairs, lists, trees, and general inductive types [28]; and control structures including recursion, higher-order functions, and continuations [48]. A representation of database query languages in the pure λ-calculus is developed in [30]. The λ-calculus can be compiled into combinators, and in fact this has been extensively studied as an implementation technique [46]. Although combinatory weak reduction does not capture all of β-reduction, it suffices to capture computation over "concrete" data types such as integers, lists etc., as shown e.g. by Theorem 3.1. Also, combinator algebras form the basic ingredient for *realizability constructions*, which are a powerful tool for building models of very expressive type theories (for textbook presentations see e.g. [12, 19]). By our results in the previous section, a combinator program M can be compiled in a syntax-directed fashion into a biorthogonal automaton $\mathcal{A}$. Moreover, note that the size of $\mathcal{A}$ is *linear* in that of M.

It remains to specify how we can use $\mathcal{A}$ to "read out" the result of the computation of M. What should be borne in mind is that the automaton $\mathcal{A}$ is giving a description of the *behaviour* of the functional process corresponding to the program it has been compiled from. It is *not* the case that the terms in T_Σ input to and output from the computations of $\mathcal{A}$ correspond directly to the inputs and outputs of the functional computation. Rather, the input also has to be compiled as part of the functional term to be evaluated—this is standard in functional programming generally.[2] The automaton resulting from compiling the program *together with its input* can then be used to deduce the value of the output, provided that the output is a concrete value.

We will focus on *boolean-valued* computations, in which the result of the computation is either true or false, which we represent by the combinatory expressions $\mathbf{K}$ and $\mathbf{K}\cdot\mathbf{I}$ respectively. By virtue of the standard results on combinatory computability such as Theorem 3.1, for any (total) recursive predicate P, there is a closed combinator expression M such that, for all n, $P(n)$ holds if and only if

$$\mathbf{CL} \vdash M \cdot \bar{n} = \mathbf{K},$$

and otherwise $\mathbf{CL} \vdash M \cdot \bar{n} = \mathbf{K}\cdot\mathbf{I}$. Let the automaton obtained from the term $M \cdot \bar{n}$ be $\mathcal{A}$. Then by Theorem 6.1, $f_{\mathcal{A}} = f_{\mathbf{K}}$ or $f_{\mathcal{A}} = f_{\mathbf{K}\cdot\mathbf{I}}$. Thus to test whether $P(n)$ holds, we run $\mathcal{A}$ on the input term $r(r(\varepsilon))$. If we obtain a result of the form $l(u)$, then $P(n)$ holds, while if we obtain a result of the form $r(v)$, it does not. Moreover, this generalizes immediately to predicates on tuples, lists, trees etc., as already explained.

More generally, for computations in which e.g. an integer is returned, we can run a sequence of computations on the automaton $\mathcal{A}$, to determine which value it represents. Concretely, for Church numerals, the sequence would look like this. Firstly, we run the automaton on the input $r(r(\varepsilon))$. If the output has the form $r(l(u))$ (so that the term is '$\lambda f.\, \lambda x.\, x$') then the result is 0. Otherwise, it must have the form $l(p(u, r(v)))$ (so it is of the form $\lambda f.\, \lambda x.\, f \ldots$, i.e. it is the successor of $\ldots$), and then we run the automaton again on the input term $l(p(u, l(p(\varepsilon, v))))$.

[2] However, note that, by compositionality, the program can be compiled once and for all into an automaton, and then each input value can be compiled and "linked in" as required.

If we now get a response of the form $r(l(u))$, then the result is the successor of 0, i.e. 1 (!!). Otherwise ...

In effect, we are performing a meta-computation (which *prima facie* is irreversible), each "step" of which is a reversible computation, to read out the output. It could be argued that something analogous to this always happens in an implementation of a functional programming language, where at the last step the result of the computation has to be converted into human-readable output, and the side-effect of placing it on an output device has to be achieved.

This aspect of recovering the output deserves further attention, and we hope to study it in more detail in the future.

Pure vs. Applied λ-calculus

Our discussion has been based on using the pure λ-calculus or CL, with no constants and δ-rules [31, 15]. Thus integers, booleans etc. are all to be represented as λ-terms. The fact that λ-calculus and Combinatory Logic can be used to represent data as well as control is an important facet of their universality; but in the usual practice of functional programming, this facility is not used, and applied λ-calculi are used instead. It is important to note that this option is *not* open to us if we wish to retain reversibility. Thus if we extend the λ-calculus with e.g. constants for the boolean values and conditional, and the usual δ-rules, then although we could continue to interpret terms by orthogonal pattern-matching automata, *biorthogonality—i.e. reversibility—would be lost.* This can be stated more fundamentally in terms of Linear Logic: while the multiplicative-exponential fragment of Linear Logic (within which the λ-calculus lives) can be interpreted in a perfectly reversible fashion (possibly with the loss of soundness of some conversion rules [25, 10]), this fails for the additives. This is reflected formally in the fact that in the passage from modelling the pure λ-calculus, or Multiplicative-Exponential Linear Logic, to modelling PCF, the property of partial injectivity of the functions $f_{\mathcal{A}}$ (the "history-free strategies" in [3, 8]) is lost, and non-injective partial functions must be used [3, 8, 42]. It appears that this gives a rather fundamental delineation of the boundary between reversible and irreversible computation in logical terms. This is also reflected in the denotational semantics of the λ-calculus: for the pure calculus, complete lattices arise naturally as the canonical models (formally, the property of being a lattice is preserved by constructions such as function space, lifting, and inverse limit), while when constants are added, to be modelled by sums, inconsistency arises and the natural models are cpo's [1]. This suggests that the pure λ-calculus itself provides the ultimate reversible simulation of the irreversible phenomena of computation.

8 Universality

A minor variation of the ideas of the previous section suffices to establish universality of our computational model. Let W be a recursively enumerable set. There is a closed combinatory term M such that, for all $n \in \mathbb{N}$,

$$n \in W \iff \mathbf{CL} \vdash M \cdot \bar{n} = \bar{0}$$

and if $n \notin W$ then $M \cdot \bar{n}$ does not have a normal form. Let $\mathcal{A}$ be the automaton compiled from $M \cdot \bar{n}$. Then we have a reduction of membership in W to the question of whether $\mathcal{A}$ produces an output in response to the input $r(r(\varepsilon))$. As an immediate consequence, we have the following result.

Theorem 8.1 *Termination in biorthogonal automata is undecidable; in fact, it is* Σ_1^0*-complete.*

As a simple corollary, we derive the following result.

Proposition 8.1 *Finitely describable partial involutions are not closed under linear application.*

PROOF The linear combinators are all interpreted by finitely describable partial involutions, and it is clear that replication preserves finite describability. Hence if linear application also preserved finite describability, all combinator terms would denote finitely describable partial involutions. However, this would contradict the previous Theorem, since termination for a finitely describable partial involution reduces to a finite number of instances of pattern-matching, and hence is decidable. □

This leads to the following:

> **Open Question**: Characterize those partial involutions in $\mathcal{S}$, or alternatively, those which arise as denotations of combinator terms.

References

[1] S. Abramsky, The lazy λ-calculus, in *Research Topics in Functional Programming*, ed. D. A. Turner, pp. 65–116. Addison Wesley, 1990.

[2] S. Abramsky, R. Jagadeesan, New Foundations for the Geometry of Interaction, *Information and Computation,* 111(1):53–119, 1994. Conference version appeared in LiCS '92.

[3] S. Abramsky and R. Jagadeesan. Games and Full Completeness for Multiplicative Linear Logic, *Journal of Symbolic Logic*, vol. 59, no. 2, 543–574, 1994.

[4] S. Abramsky, R. Jagadeesan and P. Malacaria, Full Abstraction for PCF (Extended Abstract). In *Proceedings of TACS '94*, M. Hagiya and J. C. Mitchell, eds. Springer LNCS Vol. 789, pp. 1–15, 1994.

[5] S. Abramsky, Retracing Some Paths in Process Algebra. In *Proceedings of CONCUR '96*, Springer Lecture Notes in Computer Science Vol. 1119, pp. 1-17. Springer-Verlag, 1996.

[6] S. Abramsky, *Interaction, Combinators and Complexity*. Lecture Notes, Siena, Italy, 1997.

[7] S. Abramsky and M. Lenisa, A Fully Complete PER Model for ML Polymorphic Types, *Proceedings of CSL 2000*, Springer LNCS Volume 1862, pp. 140–155, 2000.

[8] S. Abramsky, R. Jagadeesan and P. Malacaria, Full Abstraction for PCF. *Information and Computation* 163:409–470, 2000. Extended abstract appeared as [4].

[9] S. Abramsky and J. Longley. *Realizability models based on history-free strategies*, manuscript, 2000.

[10] S. Abramsky and E. Haghverdi and P. J. Scott. *Geometry of Interaction and Linear Combinatory Algebras. Mathematical Structures in Computer Science* 12:625–665, 2002.

[11] J.-P. Allouche and J. Shallit, *Automatic Sequences: Theory, Applications, Generalizations.* Cambridge University Press 2003.

[12] A. Asperti and G. Longo. *Categories, Types and Structures.* MIT Press 1991.

[13] A. Avron, The semantics and proof theory of linear logic. *Theoretical Computer Science* 57:161–184, 1988.

[14] F. Baader and T. Nipkow, *Term Rewriting and All That.* Cambridge University Press, 1999.

[15] H. P. Barendregt *The Lambda Calculus*, Studies in Logic, Vol. 103, North-Holland, 1984.

[16] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Development*, 17:525–532, 1973.

[17] C. H. Bennett. The thermodynamics of computation—a review. *Int. J. Theoret. Phys.*, 21:905–940, 1982.

[18] H. Buhrman, J. Tromp and P. Vitányi (2001), Time and Space Bounds for Reversible Simulation. *Proceedings of ICALP 2001*, Springer Lecture Notes in Computer Science Vol. 2076:1017–1027, 2001.

[19] R. Crole. *Categories for Types.* Cambridge University Press, 1993.

[20] V. Danos and L. Regnier. Local and asynchronous beta-reduction. In *Proceedings of the Eighth International Symposium on Logic in Computer Science*, IEEE Press, 296–306, 1993.

[21] V. Danos and L. Regnier, Reversible, Irreversible and Optimal λ-machines, in *Electronic Notes in Theoretical Computer Science*, 1996.

[22] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science Volume B*, 243–320. Elsevier, 1990.

[23] D. Epstein, J. Cannon, D. Holt, S. Levy, M. Paterson and W. Thurston, *Word Processing in Groups*, Jones and Bartlett, 1992.

[24] J.-Y. Girard, Linear Logic. *Theoretical Computer Science* 50(1):1-102, 1987.

[25] J.-Y. Girard, Geometry of Interaction I: Interpretation of System F, in: *Logic Colloquium '88*, ed. R. Ferro, et al. North-Holland, pp. 221-260, 1989.

[26] J.-Y. Girard, Geometry of Interaction II: Deadlock- free Algorithms. In *Proceedings of COLOG-88* (P. Martin-Lof, G. Mints, eds.) Springer LNCS Vol. 417, pp. 76-93, 1990.

[27] J.-Y. Girard, Y. Lafont, L. Regnier, eds. *Advances in Linear Logic*, London Math. Soc. Series 222, Camb. Univ. Press, 1995.

[28] J.-Y. Girard, Y. Lafont and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[29] E. Haghverdi. *A Categorical Approach to Linear Logic, Geometry of Proofs and Full Completeness*. Ph.D. thesis, University of Ottawa, 2000.

[30] G. G. Hillebrand, P. C. Kanellakis and H. Mairson, Database Query Languages Embedded in the Typed Lambda Calculus, in *Proceedings of LiCS '93*, pp. 332–343, IEEE Computer Society Press, 1993.

[31] J. R. Hindley and J. P Seldin. *Introduction to Combinators and the λ-calculus*. Cambridge University Press, 1986.

[32] R. Hindley (1997) *Basic Simple Type Theory*, Cambridge Tracts in Theoretical Computer Science, no. 42, Cambridge Univ. Press.

[33] P. M. Hines, *The algebra of self-similarity and its applications*, Ph.D. thesis, University of Wales, Bangor, 1998.

[34] P. M. Hines, The categorical theory of self-similarity, *Theory and Applications of Categories*, 6:33–46, 1999.

[35] A. Joyal, R. Street and D. Verity, Traced monoidal categories. *Math. Proc. Camb. Phil. Soc.* 1996.

[36] B. Khoussainov and A. Nerode, Automatic presentations of structures, in *Springer Lecture Notes in Computer Science* 960:367–392, 1995.

[37] N. Kuhn and K. Madlener. A method for enumerating cosets of a group presented by a canonical system. In *Proceedings ISSAC '89*, 338–350, 1989.

[38] J. W. Klop. Term rewriting systems. In *Handbook of Theoretical Computer Science Volume 2*, S. Abramsky, D. Gabbay and T. S. E. Maibaum, eds., pp. 1–116. Oxford University Press, 1992.

[39] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Development*, 5:183–191, 1961.

[40] M. V. Lawson. *Inverse Semigroups: the theory of partial symmetries*. World Scientific, 1998.

[41] Y. Lecerf. Machines de Turing Réversible. *Compte Rendus*, 257:2597–2600, 1963.

[42] Ian Mackie, *The Geometry of Implementation*, Ph.D. thesis, Imperial College, University of London, 1994.

[43] P. Malacaria and L. Regnier. Some results on the interpretation of λ-calculus in Operator Algebras. In *Proceedings of the Sixth International Symposium on Logic in Computer Science*, IEEE Press, 63–72, 1991.

[44] R. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[45] M. Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.

[46] S. L. Peyton Jones. *The implementation of functional programming languages*. Prentice Hall 1987.

[47] S. Peyton Jones (ed). *Haskell 98: a Non-strict, Purely Functional Language*. Available at `http://www.haskell.org/onlinereport/`, 1999.

[48] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[49] A. S. Troelstra, *Lectures on Linear Logic*. Center for the Study of Language and Information Lecture Notes No. 29, 1992.