A Structural Approach to Reversible Computation

Samson Abramsky
Oxford University Computing Laboratory

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 [15]:

> **Reversible Computation:** R. Landauer [27] 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. [28, 13, 14]) 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* [24], 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 [15] again:

> Currently, almost no algorithms and other programs are designed ac-

> cording 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 [18, 19, 20, 21], 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.

The paper [16] contains some discussion of a reversible abstract machine for the λ-calculus. However, the approach taken and the objectives are very 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 [29], written under the present author's supervision, to ideas originating in [2, 3].

2 The 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 [17, 26, 11]. 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

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

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 $\stackrel{\mathcal{A}}{\longrightarrow}$ on configurations: $(q,t) \stackrel{\mathcal{A}}{\longrightarrow} (q',t')$ 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 [30] and Haskell [33], 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

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

with

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

is *orthogonal* if the following conditions hold:

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

Left-linearity For each i, $1 \le i \le N$, 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 [11, 17, 26], taking advantage of the fact that we are dealing with the simple case of pattern-matching.

[1] On the standard assumption made in the algorithmics of unification [11, 17] that the immediate sub-terms of a given term can be accessed in constant time.

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 the structure

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

where

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

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.

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

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

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

We briefly review some basic material. For further details, see [24].

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 $\mathbf{S}$ and $\mathbf{K}$, 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 [24]):

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

The $\mathbf{B}$ 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

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

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 $\phi(n) \simeq m$, then

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

and if $\phi(n)\uparrow$, 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 [24]):

Theorem 3.1 *The partial functions numeral-wise representable in* **CL** *are exactly the partial recursive functions.*

4 The combinatory algebra $\mathcal{R}$

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

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

We shall discuss minimal requirements on the signature at the end of the section.

4.1 Some constructions on automata

4.1.1 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

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

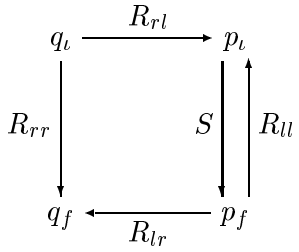
Note that the condition on x is necessary to ensure the linearity of $!R$. The biorthogonality of $!\mathcal{A}$ is easily verified.

4.1.2 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 idea is that terms of the form $r(t)$ correspond to interactions between the functional process represented by $\mathcal{A}$ and its environment, while terms of the form $l(t)$ correspond to interactions with its argument, namely the functional process represented by $\mathcal{B}$. 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.

The "flow of control" in linear application is indicated by the following diagram (which omits steps involving internal states):



4.1.3 Application

We can now define application:

$$\mathcal{A} \cdot \mathcal{B} = \mathsf{LApp}(\mathcal{A}, !\mathcal{B}).$$

4.1.4 The K automaton

We define a particular automaton $\mathcal{A}_\mathbf{K}$.

$$\mathcal{A}_\mathbf{K} = (\{q_\iota, q_f\}, q_\iota, q_f, R_\mathbf{K})$$

where $R_\mathbf{K}$ is the following set of transition rules:

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

Here we use the notation $u \leftrightarrow v$ as a shorthand for the pair of transition rules

$$(q_\iota, u) \to (v, q_f)$$

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

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

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

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

Figure 1: Linear Application

and

$$(q_\iota, v) \to (u, q_f).$$

These rules connect the output of the **K** combinator, after it has been applied to two arguments, to its first argument, thus implementing the defining equation for **K**.

The feature of this automaton that it has no internal states, and all its rules are of the above special form, is typical of the automata corresponding to *normal forms* in our interpretation of functional computation.

4.1.5 The S automaton

We define

$$\mathcal{A}_{\mathbf{S}} = (\{q_\iota, q_f\}, q_\iota, q_f, R_{\mathbf{S}})$$

where $R_{\mathbf{S}}$ is the following set of transition rules:

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

The first rule connects the output of the combinator **S**, after it has been applied to three arguments ($\mathbf{S} \cdot a \cdot b \cdot c$), to the output of its first argument a, after a has been applied to two arguments. The second rule connects the second argument of a to the result of b, after b has been applied to one argument. The third rule connects a first copy of the third argument, c, to the first argument of a. Finally the fourth rule connects a second copy of c to the first argument of b. Collectively these rules "implement" the defining equation for **S**:

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

4.2 Automata and partial symmetries

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. We write $\mathcal{I}$ for the set of all partial injective functions on T_Σ.
We now define some operations on $\mathcal{I}$.

4.2.1 Replication

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

4.2.2 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).

4.2.3 Application

$$f \cdot g = \mathsf{LApp}(f, !g).$$

It can be verified that $f \cdot g$ is a partial injective function. In fact, we have the following result.

Theorem 4.1 *$(\mathcal{I}, \cdot, f_{\mathcal{A}_{\mathbf{S}}}, f_{\mathcal{A}_{\mathbf{K}}})$ is a combinatory algebra.*

This theorem is a minor variation on the results established in [5, 6, 9, 7, 10]; see in particular [10], and the combinatory algebra of partial involutions studied in [7]. The ideas on which this construction is based stem from Linear Logic [18, 21] and Geometry of Interaction [19, 20], in the form developed by the present author and a number of colleagues [2, 3, 5, 6, 9, 7, 10].

We now want to define a subalgebra of $\mathcal{I}$ consisting of those partial injective maps "realized" or "implemented" by a biorthogonal automaton. The key result we need is the following.

Proposition 4.1 *(i)* $!f_{\mathcal{A}} = f_{!\mathcal{A}}$.
(ii) $\mathsf{LApp}(f_{\mathcal{A}}, f_{\mathcal{B}}) = f_{\mathsf{LApp}(\mathcal{A},\mathcal{B})}$.

Proof See Appendix A.
As an immediate consequence of this Proposition, we have:

Proposition 4.2 $f_{\mathcal{A}} \cdot f_{\mathcal{B}} = f_{\mathcal{A}\cdot\mathcal{B}}$.

Combining this Proposition with Theorem 4.1, we obtain the following result.

Theorem 4.2 *$\mathcal{R}$ is a combinatory sub-algebra of $\mathcal{I}$, where the carrier of $\mathcal{R}$ is the set of all $f_{\mathcal{A}}$ for biorthogonal automata $\mathcal{A}$.*

4.2.4 Partial Involutions

We say that $f \in \mathcal{I}$ is a *partial involution* if $f^{\mathsf{op}} = f$. Clearly both $f_{\mathcal{A}_{\mathbf{S}}}$ and $f_{\mathcal{A}_{\mathbf{K}}}$ are partial involutions, and partial involutions are closed under application [7]. 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].

4.3 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 [31].

This restricted situation is still of interest. It suffices to interpret **BCK**-algebras, and hence the *affine* λ-calculus [25]. 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):

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

$R_{\mathbf{B}}$:

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

$R_{\mathbf{C}}$:

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

5 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 [22]; and control structures including recursion, higher-order functions, and continuations [34]. A representation of database query languages in the pure λ-calculus is developed in [23]. The λ-calculus can be compiled into combinators, and in fact this has been extensively studied as an implementation technique [32]. 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. 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 4.2, $f_{\mathcal{A}} = f_{\mathbf{K}}$ 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 $l(p(u, l(p(\varepsilon, v))))$. If we now get a response of the form $r(l(u))$, then the result is the successor of 0, i.e. 1 (!!). Otherwise $\ldots$

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.

5.1 Pure vs. Applied λ-calculus

Our discussion has been based on using the pure λ-calculus or CL, with no constants and δ-rules [24, 12]. 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 prac-

[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.

tice 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 [19, 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, 29]. 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.

6 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 6.1 *Termination in biorthogonal automata is undecidable; in fact, it is Σ_1^0-complete.*

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*. Submitted for publication, 2000.

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

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

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

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

[15] H. Buhrman, J. Tromp and P. Vitányi (2001), Time and Space Bounds for Reversible Simulation. To appear in the Proceedings of ICALP 2001.

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

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

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

[19] (1989) 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.

[20] (1990) 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.

[21] (1995) J.-Y. Girard, Y. Lafont, L. Regnier, eds. *Advances in Linear Logic*, London Math. Soc. Series 222, Camb. Univ. Press.

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

[23] 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.

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

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

[26] 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.

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

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

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

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

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

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

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

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

Appendix A: Proof of Proposition 4.1

(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. □