Towards a Reversible Functional Language

Tetsuo Yokoyama[1], Holger Bock Axelsen[2], and Robert Glück[2]

[1] Department of Software Engineering, Nanzan University
tyokoyama@acm.org
[2] DIKU, Department of Computer Science, University of Copenhagen
funkstar@diku.dk, glueck@acm.org

Abstract. We identify concepts of reversibility for a functional language by means of a set of semantic rules with specific properties. These properties include injectivity along with local backward determinism, an important operational property for an efficient reversible language. We define a concise reversible first-order functional language in which access to the backward semantics is provided to the programmer by inverse function calls. Reversibility guarantees that in this language a backward run (inverse interpretation) is as fast as the corresponding forward run itself. By adopting a *symmetric first-match* policy for case expressions, we can write overlapping patterns in case branches, as is customary in ordinary functional languages, and also in leaf expressions, unlike existing inverse interpreter methods, which enables concise programs. In patterns, the use of a duplication/equality operator also simplifies inverse computation and program inversion. We discuss the advantages of a reversible functional language using example programs, including run-length encoding. Program inversion is seen to be as lightweight as for imperative reversible languages and realized by recursive descent. Finally, we show that the proposed language is *r-Turing complete.*

1 Introduction

Functional languages provide a natural and general mechanism for manipulating structured data, associated with powerful pattern-matching features and abstract data types. They also enable higher-level abstractions than imperative languages and thus more concise programs with the same functionality. For these reasons it is interesting to apply such a well-established language paradigm to reversibility. We aim to develop a reversible functional language for studying the foundations of reversible programming.

From the viewpoint of reversibility, functional programming enforces us to take a more rigorous and restricted approach than in existing work. For example, the imperative reversible language Janus [12,18] allows irreversible arithmetic and logical operators in expressions, and these do not have bidirectionality between input and output. In pure functional languages, it is more natural to describe natural numbers using only declarative fundamental features, *e.g.* based on the Peano axioms. This approach forces us to use only purely reversible language constructions, which is suitable for developing both basic theory and general concepts for reversible computing.

The contribution of this paper is to identify a concept of reversibility in a functional language setting and specify a purely-reversible and purely-functional language. Reversible computation of injective functions can be regarded as a case of inverse interpretation [1]. Because general solutions to inverse interpretation, *e.g.* McCarthy's generate-and-test [13], are often inefficient in practice, our interest lies in finding a solution by constructing a programming language which has efficient inverse computation built in by design.

Injectivity of the function implemented by a program is not sufficient for *efficient* inverse interpretation; backward deterministic control flow is also required. Consider the function *inc*, which takes a natural number n as argument and returns $n+1$ in the form of Peano numbers, where Z denotes zero and $S(n)$ denotes the successor of n. A straightforward implementation is:

$$\begin{aligned} \mathit{inc}\ n \triangleq\ & \textbf{case}\ n\ \textbf{of} \\ & Z \quad\ \ \to \underline{S(Z)} \\ & S(n') \to \textbf{let}\ n'' = \mathit{inc}\ n'\ \textbf{in}\ \underline{S(n'')} \end{aligned} \tag{1}$$

For example, *inc* $(S(S(Z)))$ returns $S(S(S(Z)))$. Since *inc* is injective, the inverse function inc^{-1} exists. But the naïve general inverse interpretation of *inc* has local *nondeterminism*; the underlined expressions can match the same value, $S(Z)$, and inverse interpretation cannot immediately distinguish between the branches.

A reversible functional languages must guarantee both backward determinism and termination (if the input for a given output exists) without using the general approach. Here, we propose a reversible functional language which has these properties. We say that a reversible language has *locally* forward and backward deterministic semantics. Below, we specify what this means in operational semantics.

For exploring the theoretical foundations of reversible computing, we focus on a purely-reversible and side-effect free first-order functional language. There exist several imperative and pseudo-functional reversible languages. As far as we know, Janus [12,20] is the first reversible language, and is imperative. We regard Gries' invertible language [11], R and PISA [8] as also belonging to this category. Baker proposed Ψ-Lisp [4], reversible linear Lisp; due to the use of a state and a hidden history stack, it is neither purely reversible nor purely functional. Mu, Hu, and Takeichi proposed INV [16], a point-free functional language with relational semantics, which essentially includes both forward and backward nondeterminism. Bowman, James, and Sabry have proposed Π [6], another point-free reversible functional language. Because of the nature of point-free languages, these do not have powerful pattern matching, so *e.g.* overlapping branch patterns in loops is prohibited. While the above examples all have reversible language features, a main contribution here is to separate reversibility in functional languages from other features.

Grammar:				Syntax domains:
q	$::=$	d^*	(program)	$q \in$ Programs
d	$::=$	$f\ l \triangleq e$	(definition)	$d \in$ Definitions
l	$::=$	x	(variable)	$f \in$ Functions
	$\mid$	$c(l_1, \ldots, l_n)$	(constructor)	$l \in$ Left-expressions
	$\mid$	$\lfloor l \rfloor$	(duplication/equality)	$e \in$ Expressions
e	$::=$	l	(left-expression)	$x \in$ Variables
	$\mid$	$\textbf{let}\ l_{out} = f\ l_{in}\ \textbf{in}\ e$	(let-expression)	$c \in$ Constructors
	$\mid$	$\textbf{rlet}\ l_{in} = f\ l_{out}\ \textbf{in}\ e$	(rlet-expression)	
	$\mid$	$\textbf{case}\ l\ \textbf{of}\ \{l_i \to e_i\}_{i=1}^{m}$	(case-expression)	

Fig. 1. Abstract syntax of the first-order functional language ($n \geq 0$, $m \geq 1$)

2 The Language

The reversible functional language that we present here is simple, yet powerful enough to be r-Turing complete (see Section 2.6). Both in syntax and semantics, this language differs from conventional first-order functional languages. For example, the language is extended to include inverse function calls, and symmetric matching for case-expressions serves to make its semantics forward and backward deterministic. It may serve as a model for designing other, more sophisticated reversible functional languages.

2.1 Syntax

The first-order functional language (Fig. 1) is tailored to guarantee reversibility and is a modfied version of a language that was originally defined to investigate automatic program inversion [10]. A *program* q is a sequence of function definitions. A *function definition* d consists of a pattern l (left-expression) and a body e (expression). A *left-expression* l can contain variables, constructors and duplication/equality operators ($\lfloor \cdot \rfloor$). An expression e is a left-, let-, rlet- or case-expression. An *rlet-expression* invokes the inverse semantics of a function f. We call $l_i \to e_i$ the i-th *branch* of a case-expression.

We consider only well-formed programs in the following sense: each variable in patterns appears at most once, and each variable is bound before its use and is used *linearly* in each branch. A value v is recursively defined by a constructor c with arguments v_i: $v ::= c(v_1, \ldots, v_n)$ where $n \geq 0$. As is customary, a nullary constructor $c()$ is written as c. A list is constructed by an infix constructor (:) and a nullary constructor [ ] that represents the empty list. The unary and binary tuples, $\langle \cdot \rangle$ and $\langle \cdot, \cdot \rangle$, are a convenient shorthand for two different constructors.

2.2 Reversibility and Most General Matcher

Linearity is essential to a reversible language to avoid discarding values. Every variable defined by a pattern on the left-hand side of a case-expression must be used once in the expression on the right-hand side, otherwise information is lost.

Sometimes, we want to use a value more than once, but this is not allowed by linearity. Instead of duplicating the value implicitly by using a variable twice, we make this operation explicit by requiring that the value is duplicated by the operator $\lfloor\cdot\rfloor$. For example, rather than syntactically using variable h twice to duplicate the head of a list, as in expression $h : h : t$, the value is duplicated by $\lfloor\langle h\rangle\rfloor$ and bound to two fresh variables h_1 and h_2 in a case-expression:

$$\begin{array}{ll} \underline{\text{Invalid}} & \underline{\text{Well-formed}} \\ \mathit{dbl}\ x \triangleq \ \textbf{case}\ x\ \textbf{of} & \mathit{dbl}\ x \triangleq \ \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad h : t \to h : h : t & \qquad\qquad h : t \to \textbf{case}\ \lfloor\langle h\rangle\rfloor\ \textbf{of} \\ & \qquad\qquad\qquad \langle h_1, h_2\rangle \to h_1 : h_2 : t \end{array} \tag{2}$$

Using an explicit duplication operator simplifies the inversion of functional programs because duplication in one computation direction requires an equality test in the other direction, and vice versa.

The above *duplication/equality operator* [9] is defined by

$$\lfloor\langle v\rangle\rfloor \ = \ \langle v, v\rangle \qquad \text{(duplication)} \tag{3}$$

$$\lfloor\langle v, v'\rangle\rfloor \ = \ \begin{cases} \langle v\rangle & \textbf{if}\ v = v' \\ \langle v, v'\rangle & \textbf{otherwise} \end{cases} \qquad \text{(equality test)} \tag{4}$$

The operator is *self-inverse*, which means that it can be used to determine its input from its output (e.g., $\lfloor\langle v\rangle\rfloor = \langle v, v\rangle$ and $\langle v\rangle = \lfloor\langle v, v\rangle\rfloor$).

We take this idea further and allow the operator to occur also in the patterns of case-expressions. This simplifies forward and backward computation, and the inversion of programs. To illustrate this symmetry, consider the following pairwise functionally equivalent case-expressions for duplicating and testing values:

$$\begin{array}{ll} \textbf{case}\ \lfloor\langle l\rangle\rfloor\ \textbf{of} & \textbf{case}\ \langle l\rangle\ \textbf{of} \\ \quad \langle x, y\rangle \to \cdots & \quad \lfloor\langle x, y\rangle\rfloor \to \cdots \end{array} \qquad \text{(duplication)} \tag{5}$$

$$\begin{array}{ll} \textbf{case}\ \lfloor\langle l, l'\rangle\rfloor\ \textbf{of} & \textbf{case}\ \langle l, l'\rangle\ \textbf{of} \\ \quad \langle x\rangle \quad\ \to \cdots & \quad \lfloor\langle x\rangle\rfloor \quad\ \to \cdots \\ \quad \langle x, y\rangle \to \cdots & \quad \lfloor\langle x, y\rangle\rfloor \to \cdots \end{array} \qquad \text{(equality test)} \tag{6}$$

This extension of patterns has the advantage that the same left-expressions can be used everywhere, which simplifies inverse computation and program inversion.

The $\lfloor\cdot\rfloor$-operator can occur anywhere in a left-expression (see Fig. 1). Equality of values in pairs can be tested at the leaf left-expressions in backward computation, which is useful for case selections, as we shall see in the example function

$$\frac{}{v \lhd x \rightsquigarrow \{x \mapsto v\}}\ \textsc{Var}$$

$$\frac{v_1 \lhd l_1 \rightsquigarrow \sigma_1 \quad \cdots \quad v_n \lhd l_n \rightsquigarrow \sigma_n}{c(v_1, \ldots, v_n) \lhd c(l_1, \ldots, l_n) \rightsquigarrow \uplus_{i=1}^{n} \sigma_i}\ \textsc{Con}$$

$$\frac{\lfloor v \rfloor\!\downarrow = v' \quad v' \lhd l \rightsquigarrow \sigma}{v \lhd \lfloor l \rfloor \rightsquigarrow \sigma}\ \textsc{Dup/Eq}$$

Fig. 2. R-Match: Reversible matching operation ($\uplus$ denotes disjoint union)

plus (Fig. 4). Left-expressions define a unique composition and decomposition of values in our language. Pattern matching using left-expressions is formalized as follows. To define a reversible matching semantics, we use a *most general matcher* between left-expressions and values. Given linear left-expression l and value v, *left-exp judgment*

$$v \lhd l \rightsquigarrow \sigma \tag{7}$$

returns σ, the most general matcher of l and v. Figure 2 shows the semantic rules R-Match that define the most general matcher. A *substitution* σ is a mapping of variables x_i to values v_i: $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ where $n \geq 0$. In particular, $\{\,\}$ is the identity substitution, σl is the application of σ to l, and $l\downarrow$ is the application of all $\lfloor\cdot\rfloor$-operators in left-expression l by means of Eq. 3 and 4.

Lemma 1 (The Most General Matcher). *Given value v and left-expression l, if left-exp judgment $v \lhd l \rightsquigarrow \sigma$ is satisfied, then σ is the most general matcher of l and v such that*

$$v \lhd l \rightsquigarrow \sigma \quad \Longrightarrow \quad (\sigma l)\!\downarrow = v \ \wedge \ \forall \sigma'. \ \big[(\sigma' l)\!\downarrow = v \ \Longrightarrow \ \exists \sigma''. \ \sigma' = \sigma \uplus \sigma''\big] \ . \tag{8}$$

Proof. By straightforward induction on the derivation of $v \lhd l \rightsquigarrow \sigma$. □

2.3 Semantics

An *expression judgment* is a relation of a substitution σ, program q, expression e, and value v:

$$\sigma \vdash_q e \hookrightarrow v \,. \tag{9}$$

The operational semantics is defined in Fig. 3. LeftExp rule is resolved by a left-exp judgment. Without loss of generality, to avoid name clashes, we assume that l_f and e_f in the FunExp rule contain fresh variables each time f is applied.

LetExp and RLetExp rules have inverse functionality. While both rules apply function f in the premise, their input and output, as indicated by the subscripts, are exchanged. The reversible semantics allows us to define the inverse functionality by swapping the input and output without invoking program inversion. Only a reversible semantics enables this *rule sharing.* We shall see examples of rlet-expressions later when we define a reversible Turing machine.

The CaseExp rule is more involved. We constrain the semantics of case-expressions to be symmetric regarding the branch selection by requiring that the

$$\frac{v \lhd l \rightsquigarrow \sigma}{\sigma \vdash_q l \hookrightarrow v}\ \textsc{LeftExp}$$

$$\frac{f\ l_f \triangleq e_f \in q \quad \sigma \vdash_q l \hookrightarrow v' \quad v' \lhd l_f \rightsquigarrow \sigma_f \quad \sigma_f \vdash_q e_f \hookrightarrow v}{\sigma \vdash_q f\ l \hookrightarrow v}\ \textsc{FunExp}$$

$$\frac{\sigma_{in} \vdash_q f\ l_{in} \hookrightarrow v_{out} \quad v_{out} \lhd l_{out} \rightsquigarrow \sigma_{out} \quad \sigma_{out} \uplus \sigma_e \vdash_q e \hookrightarrow v}{\sigma_{in} \uplus \sigma_e \vdash_q \textbf{let}\ l_{out} = f\ l_{in}\ \textbf{in}\ e \hookrightarrow v}\ \textsc{LetExp}$$

$$\frac{v_{in} \lhd l_{in} \rightsquigarrow \sigma_{in} \quad \sigma_{out} \vdash_q f\ l_{out} \hookrightarrow v_{in} \quad \sigma_{out} \uplus \sigma_e \vdash_q e \hookrightarrow v}{\sigma_{in} \uplus \sigma_e \vdash_q \textbf{rlet}\ l_{in} = f\ l_{out}\ \textbf{in}\ e \hookrightarrow v}\ \textsc{RLetExp}$$

$$\frac{\begin{array}{c}\sigma_l \vdash_q l \hookrightarrow v' \quad \sigma_{l_j} \uplus \sigma_t \vdash_q e_j \hookrightarrow v \\ j = \min\{i \mid v' \lhd l_i \rightsquigarrow \sigma_{l_i}\} = \min\{i \mid l' \in \mathit{leaves}(e_i)\ \wedge\ v \lhd l' \rightsquigarrow \_\}\end{array}}{\sigma_l \uplus \sigma_t \vdash_q \textbf{case}\ l\ \textbf{of}\ \{l_i \to e_i\}_{i=1}^m \hookrightarrow v}\ \textsc{CaseExp}$$

Fig. 3. Operational semantics for the reversible functional language

first-matching branch is the same in *both* directions. Otherwise, the CaseExp is undefined. Consider the following case-expression as an example.

$$\begin{array}{l}\textbf{case}\ l\ \textbf{of} \\ \quad l_1 \to\ \cdots\ \textbf{in}\ l'_1 \\ \qquad \vdots \\ \quad l_i \to\ \cdots\ \textbf{in}\ l'_i \\ \qquad \vdots \\ \quad l_n \to\ \cdots\ \textbf{in}\ l'_n \end{array} \tag{10}$$

The value v of l is matched against the left-hand side of each branch (l_1, l_2, ...) until the first successful match at some l_i, that is, $v \lhd l_i \rightsquigarrow \sigma_i$. As usual, the right-hand side of the i-th branch is then evaluated in σ_i and a value v' is returned by l'_i. Now, for symmetry, we require that v' does *not* match any of the preceding $l'_1, \ldots, l'_{i-1}$; otherwise, the case-expression is undefined. This *symmetric first-match policy* ensures that case-expressions forward and backward deterministic.

Thus, in backward computation, a given value v' of the whole case-expression is matched against the leaf left-expressions of each branch (l'_1, l'_2, ...) until the first successful match at some l'_i, that is, $v' \lhd l'_i \rightsquigarrow \sigma'_i$. The result of the backward computation of the right hand side of the i-th branch instantiates l_i to v. Then, for symmetry, we require that v does not match any of the preceding $l_1, \ldots, l_{i-1}$; otherwise, evaluating the case-expression is undefined. (The set *leaves*(e) contains all left-expressions at the tips of e.[1]) Therefore, as usual, $l_1, \ldots, l_n$ need not be syntactically orthogonal and the same holds for $l'_1, \ldots, l'_n$.

[1] $\mathit{leaves}(\textbf{let}\ l_1{=}f\ l_2\ \textbf{in}\ e){=}\mathit{leaves}(e)$, $\mathit{leaves}(\textbf{case}\ l\ \textbf{of}\ \{p_i \to e_i\}_{i=1}^m) = \cup_{i=1}^m \mathit{leaves}(e_i)$ $\mathit{leaves}(\textbf{rlet}\ l_1{=}f\ l_2\ \textbf{in}\ e) = \mathit{leaves}(e)$, $\mathit{leaves}(l) = \{l\}$.

$$\begin{aligned} \mathit{fib}\ n \triangleq\ & \textbf{case}\ n\ \textbf{of} \\ & \quad Z \quad\ \ \to \langle S(Z), S(Z)\rangle \\ & \quad S(m) \to \textbf{let}\ \langle x, y\rangle = \mathit{fib}\ m\ \textbf{in} \\ & \quad\qquad\qquad \textbf{let}\ z = \mathit{plus}\ \langle y, x\rangle\ \textbf{in}\ z \end{aligned} \tag{12}$$

$$\begin{aligned} \mathit{plus}\ \langle x, y\rangle \triangleq\ & \textbf{case}\ y\ \textbf{of} \\ & \quad Z \quad\ \to \lfloor\langle x\rangle\rfloor \\ & \quad S(u) \to \textbf{let}\ \langle x', u'\rangle = \mathit{plus}\ \langle x, u\rangle\ \textbf{in}\ \langle x', S(u')\rangle \end{aligned} \tag{13}$$

Fig. 4. Fibonacci-pair function *fib* and addition $\mathit{plus}\langle x, y\rangle = \langle x, x+y\rangle$[2]

Because of the symmetric semantics of case-expressions, we can compute the increment function from above both forward and backward:

$$\{n \mapsto Z\} \vdash_q \mathit{inc}\ n \hookrightarrow S(Z) \tag{11}$$

where q is a program which includes the function definition of *inc* in Eq. 1. Without the symmetric first-match policy, the value $S(Z)$ could be a consequence of two different instances of the CaseExp rule because $S(Z)$ matches both of the underlined left-expressions $S(Z)$ and $S(n'')$, and we would thus have to search deeper in the derivation tree to decide which was the right instance. However, the policy ensures that inverse interpretation is locally deterministic and, in this example, selects the first branch and never the second.

If a function terminates with an output for a given input, inverse computation of the function terminates for that output and returns the original input, and vice versa.

Example program. Given a number n, the *Fibonacci-pair function* [9] computes a tuple containing the $(n+1)$-th and $(n+2)$-th Fibonacci number. The functions *fib* and *plus* are defined for Peano numbers in Fig. 4. Note the use of the $\lfloor\cdot\rfloor$-operator on the right-hand side of the first branch of *plus* to duplicate x in forward computation and to check equality of a pair of values in backward computation. We can relate numbers to the corresponding Fibonacci pairs via an expression judgment. For example, for the second pair we have:

$$\{n \mapsto S(S(Z))\} \vdash_q \mathit{fib}\ n \hookrightarrow \langle S(S(Z)), S(S(S(Z)))\rangle \tag{14}$$

2.4 Reversibility and Semantics

In this section, we show in what sense the functional language defined above is reversible. We first examine the matching operation (left-expression judgments) and then continue with the rules of the operational semantics (expression judgments).

[2] For simplicity, $x + y$ represents the Peano number for the sum of x and y.

We have local determinism in the left-exp judgment in the sense that l determines uniquely which rule applies, and in the premises each left-expression is uniquely determined. Here, let $\_$ be a wildcard value or a wildcard substitution.[3]

Lemma 2 (The Unique Derivation of Left-Exp Judgments $(\cdot \lhd l \leadsto \cdot)$). *Any left-exp judgment $\_ \lhd l \leadsto \_$ is the consequence of at most a single rule in Fig. 2 and in its premises left-expressions are determined uniquely.*

Proof. Given linear left-expression l, left-exp judgment $\_ \lhd l \leadsto \_$ determines which rule to apply because all left-expressions l in the consequences are orthogonal. Also, all and only the immediate proper sub-left-expressions of l (*e.g.* $l_1, \ldots, l_n$ for $l = c(l_1, \ldots, l_n)$) appear exactly once in the premise left-exp judgments, which are thus also determined uniquely. □

This implies efficiency for an implementation of the left-exp judgment, as the structure of l completely decides the structure of the derivation. We further have that for any given left-expression l the left-exp judgment is an injective relation between substitutions and values.

Lemma 3 (The Global Reversibility of Left-Exp Judgments $(\cdot \lhd l \leadsto \cdot)$). *The* R-Match *relation $\cdot \lhd \cdot \leadsto \cdot$ (Fig. 2) obeys the following formulas.*

$$\forall l \forall \sigma \forall v_1 \forall v_2.\ v_1 \lhd l \leadsto \sigma \ \wedge\ v_2 \lhd l \leadsto \sigma \implies v_1 = v_2 \tag{15}$$

$$\forall l \forall v \forall \sigma_1 \forall \sigma_2.\ v \lhd l \leadsto \sigma_1 \ \wedge\ v \lhd l \leadsto \sigma_2 \implies \sigma_1 = \sigma_2 \tag{16}$$

Proof. By Lemma 2 the derivation tree for any left-expression judgment completely and uniquely follows the structure of l, so the two derivations in the antecedents of the implications in formulas (15) and (16) must use the same rules in both consequence and premises. By induction on these derivations the lemma then easily follows. □

Note that Lemma 2 does not imply Lemma 3 for arbitrary rule sets, and that the inverse direction is also not satisfied: it is possible to satisfy global reversibility without unique derivations. Also, the rule set in which the Var rule is replaced with

$$\frac{}{\_ \lhd x \leadsto \{x \mapsto \_\}}\ \textsc{Var'}$$

still satisfies Lemma 2 but not Lemma 3.

In the operational semantics (Fig. 3) for the exp-judgment, we again have that the rule selection is locally and uniquely determined.

Lemma 4 (The Unique Derivation of Exp-Judgments $(\cdot \vdash_q e \hookrightarrow \cdot)$). *Any expression judgment $\_ \vdash_q e \hookrightarrow \_$ is the consequence of at most a single rule in Fig. 3 and in its premises expressions and left-expressions are determined uniquely, except for the* CaseExp *rule where either a substitution σ or value v is needed for uniqueness.*

[3] The instances of a wildcard value are arbitrary; two wildcards do not necessarily have the same value.

Proof. Analogous to Lemma 2, expression e uniquely determines which rule to apply because all expressions e in the consequences are orthogonal. For all rules except CaseExp the syntactic form of the expression in the consequence of the exp-judgment also determines which rules to apply in the premises by simple case analysis.

In the CaseExp rule, the symmetric first-match policy determines the premise for the case chosen, so the value of j depends on the particular substitution σ (or value v), meaning that e_j is not syntactically defined by e alone (as all the other expressions and left-expressions in the premises are). However, for a given substitution σ (or value v) uniqueness of e_j follows by Lemmas 2 and 3. $\square$

This sort of *local determinism* and the reversibility of left-exp judgments leads to the following lemma.

Lemma 5 (The Global Reversibility of Exp-Judgments ($\cdot \vdash_q e \hookrightarrow \cdot$)). *The operational semantics ($\cdot \vdash_q \cdot \hookrightarrow \cdot$) (Fig. 3) obeys the following formulas.*

$$\forall e \forall v \forall \sigma_1 \forall \sigma_2 .\ \sigma_1 \vdash_q e \hookrightarrow v \ \wedge\ \sigma_2 \vdash_q e \hookrightarrow v \implies \sigma_1 = \sigma_2 \tag{17}$$

$$\forall e \forall \sigma \forall v_1 \forall v_2 .\ \sigma \vdash_q e \hookrightarrow v_1 \ \wedge\ \sigma \vdash_q e \hookrightarrow v_2 \implies v_1 = v_2 \tag{18}$$

Proof (Sketch). Similar to Lemma 3, by induction on the derivations of the exp-judgments in the antecedent of each implication. The reversibility of any left-exp judgments in the premises of such derivations is ensured by Lemma 3. $\square$

While property (18) should be recognizable as forward determinism, the reversibility of the language semantics is encapsulated by property (17) and is distinctly non-standard: for any given expression e (including functional calls), we only need the resulting value v, to determine the unique initial environment σ wherein e evaluates to v.

Analogous to the relationship between Lemma 2 and Lemma 3, Lemma 4 does not imply Lemma 5 and the inverse direction is also not satisfied.

Reversibility affects termination analysis as well. Reversibility guarantees that backward interpretation terminates if forward interpretation terminates, and vice versa. On the other hand, reversibility does not by itself guarantee termination. Consider the function

$$\mathit{infinite}\ x \ \triangleq\ \textbf{let}\ y = \mathit{infinite}\ x\ \textbf{in}\ y\,. \tag{19}$$

For example, we can reasonably expect that in an implementation, the function expression *infinite Z* is recursively unfolded arbitrary times by the LetExp and FunExp rules, leading to non-termination. However, the infinite unfolding process is still locally reversible and the expression (vacuously) satisfies the global reversibility properties described in Lemma 5.

2.5 Examples

We shall here show how programs are realized in the proposed language, and how program inversion is lightweight. *Run-length encoding* is a data compression

$$
\begin{array}{l}
\mathit{pack}\ s \triangleq \mathbf{case}\ s\ \mathbf{of} \\
\quad [\,] \to [\,] \\
\quad c_1 : r \to \mathbf{let}\ s = \mathit{pack}\ r\ \mathbf{in} \\
\qquad \mathbf{case}\ s\ \mathbf{of} \\
\qquad\quad [\,] \to \langle c_1, S(Z)\rangle : [\,] \\
\qquad\quad h : t \to \mathbf{case}\ h\ \mathbf{of} \\
\qquad\qquad \langle c_2, n\rangle \to \mathbf{case}\ \lfloor\langle c_1, c_2\rangle\rfloor\ \mathbf{of} \\
\qquad\qquad\quad \langle c_1', c_2'\rangle \to \langle c_1', S(Z)\rangle : (\langle c_2', n\rangle : t) \\
\qquad\qquad\quad \langle c\rangle \to \langle c, S(n)\rangle : t
\end{array}
\tag{20}
$$

Fig. 5. Run-length encoding function *pack*

algorithm in which each contiguous single-character sequence is replaced with a pair of the character and its count. In the proposed reversible language a (recursive) program for run-length encoding *pack* is shown in Fig. 5 (cf. [9]). For example, *pack* $[\mathrm{A},\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{C},\mathrm{C}]$[4] evaluates to $[\langle A,2\rangle,\langle B,1\rangle,\langle C,3\rangle]$,[5] which is realized by an expression judgment

$$\{\,\} \vdash_q \mathit{pack}\ [\mathrm{A},\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{C},\mathrm{C}] \ \hookrightarrow\ [\langle A,2\rangle,\langle B,1\rangle,\langle C,3\rangle]\,. \tag{21}$$

Now, all functions are reversible, and can be inversely applied. Hence, function *unpack* can be defined very simply using *pack*:

$$\mathit{unpack}\ x \ \triangleq\ \mathbf{rlet}\ x = \mathit{pack}\ y\ \mathbf{in}\ y\,. \tag{22}$$

In general, for any well-formed function f and variables x and y, expression

$$\mathbf{rlet}\ x = f\ y\ \mathbf{in}\ y \tag{23}$$

returns the same value as $f^{-1}\ x$ does where f^{-1} is an inverse function of f.

In contrast to irreversible languages, program inversion for the reversible language is always possible and lightweight in the sense that it does not require global program analysis. A recursive descent local program inversion for the language is given in Fig. 6. For the inversion of case expression, unification of left-expression l and each of the patterns p_i is used to generate patterns for the inverse cases. Failure of unification means that the branch is never selected no matter what instances of l are provided in the forward interpretation. Even in such a case, the translation of the branch e_i has to continue, as the symmetric first-match policy enforces us to check the tips of e_i during computation.

For example, program inversion of *fib* in Fig. 4 yields fib^{-1} in Fig. 7. As in an ordinary functional language, the first-match policy in the forward direction

[4] Following the standard convention, list $A\!:\!B\!:\!C\!:\![\,]$ is abbreviated as $[A,B,C]$.

[5] For brevity, Peano numbers are here represented by ordinary decimal numbers.

$$\begin{array}{rcl}
\mathcal{I}_p[\![d^*]\!] & = & \mathcal{I}_d[\![d]\!]^* \\
\mathcal{I}_d[\![f\ l \triangleq e]\!] & = & f^{-1}\ x \triangleq \mathbf{case}\ x\ \mathbf{of}\ \mathcal{I}[\![e, l]\!] \\
\\
\mathcal{I}[\![l, e]\!] & = & \{l \rightarrow e\} \\
\mathcal{I}[\![\mathbf{let}\ l_1 = f\ l_2\ \mathbf{in}\ e', e]\!] & = & \mathcal{I}[\![e', \mathbf{let}\ l_2 = f^{-1}\ l_1\ \mathbf{in}\ e]\!] \\
\mathcal{I}[\![\mathbf{rlet}\ l_1 = f\ l_2\ \mathbf{in}\ e', e]\!] & = & \mathcal{I}[\![e', \mathbf{rlet}\ l_2 = f^{-1}\ l_1\ \mathbf{in}\ e]\!] \\
\mathcal{I}[\![\mathbf{case}\ l\ \mathbf{of}\ \{p_i \rightarrow e_i\}_{i=1}^m, e]\!] & = & \cup_{i=1}^m(\mathbf{if}\ \sigma_i \neq \bot\ \mathbf{then}\ \mathcal{I}[\![e_i, \sigma_i e]\!] \\
 & & \qquad \mathbf{else}\ \mathcal{I}[\![e_i, \mathbf{case}\ p_i\ \mathbf{of}\ l \rightarrow e]\!]) \\
 & & \mathbf{where}\ \sigma_i \text{ is the unification of } l \text{ and } p_i
\end{array}$$

Fig. 6. Program inversion (x is a fresh variable)

$$\begin{array}{lll}
\mathit{fib}^{-1}\ x_1 & \triangleq & \mathbf{case}\ x_1\ \mathbf{of} \\
 & & \quad \langle S(Z), S(Z)\rangle \rightarrow Z \\
 & & \quad x_2 \qquad\qquad\ \rightarrow \mathbf{let}\ \langle y, x\rangle = \mathit{plus}^{-1}\ x_2\ \mathbf{in} \\
 & & \qquad\qquad\qquad\quad\ \ \mathbf{let}\ m = \mathit{fib}^{-1}\ \langle x, y\rangle\ \mathbf{in} \\
 & & \qquad\qquad\qquad\quad\ \ S(m)
\end{array} \tag{24}$$

$$\begin{array}{l}
\mathit{plus}^{-1}\ z \triangleq \mathbf{case}\ z\ \mathbf{of} \\
\qquad \lfloor\langle x\rangle\rfloor \qquad\ \rightarrow \langle x, Z\rangle \\
\qquad \langle x', S(u')\rangle \rightarrow \mathbf{let}\ \langle x, u\rangle = \mathit{plus}^{-1}\ \langle x', u'\rangle\ \mathbf{in}\ \langle x, S(u)\rangle
\end{array} \tag{25}$$

Fig. 7. Inverse functions of *fib* and *plus* (x_1 and x_2 are fresh variables)

ensures x_2 only match with values that are not $\langle S(Z), S(Z)\rangle$. The subtraction, $\mathit{plus}^{-1}\langle x, x + y\rangle = \langle x, y\rangle$, is obtained by program inversion of *plus* from Fig. 4. Here, we see how it is convenient that the $\lfloor\cdot\rfloor$ operator can occur in the pattern of a case-expression, so that, in this example, the inversion is realized by just swapping the left- and right-hand sides of branches.

2.6 *r*-Turing Completeness

We show the proposed language is r-Turing complete [3]; the language can simulate any reversible Turing machine (RTM).

Definition 1 (Turing Machine). *A Turing machine T is a tuple $(Q, \Sigma, b, \delta, q_s, q_f)$ where Q is a finite set of states, Σ is a finite set of tape symbols, $b \in \Sigma$ is the blank symbol,*

$$\delta : Q \times [(\Sigma \times \Sigma) \cup \{\leftarrow, \downarrow, \rightarrow\}] \times Q \tag{26}$$

is a partial relation defining the transition rules, $q_s \in Q$ is the starting state, and $q_f \in Q$ is the final state. Symbols $\leftarrow$, $\downarrow$, $\rightarrow$ represent the three shift directions (left, stay, right).

The form of a triple $\in \delta$ is either $\langle q_1, \langle s_1, s_2\rangle, q_2\rangle$ or $\langle q_1, d, q_2\rangle$ where $q_1, q_2 \in Q$, $s_1, s_2 \in \Sigma$, and $d \in \{\leftarrow, \downarrow, \rightarrow\}$. The former *symbol rule* says that in state q_1 with the tape head reading symbol s_1, write s_2 and change into state q_2. The latter *shift rule* says that in state q_1, move the tape head in direction d and change the state to q_2.

Definition 2 (Reversible Turing Machine [5,3]). *A Turing machine T is* forward deterministic *iff for any distinct pair of triples $\langle q_1, a, q_2\rangle \in \delta$ and $\langle q_1', a', q_2'\rangle \in \delta$, if $q_1 = q_1'$ then $a = \langle s_1, s_2\rangle \wedge a' = \langle s_1', s_2'\rangle \wedge s_1 \neq s_2$. A Turing machine is* backward deterministic *iff for any distinct pair of triples $\langle q_1, a, q_2\rangle \in \delta$ and $\langle q_1', a', q_2'\rangle \in \delta$, if $q_2 = q_2'$ then $a = \langle s_1, s_2\rangle \wedge a' = \langle s_1', s_2'\rangle \wedge s_2 \neq s_2'$. A Turing machine is* reversible *iff it is forward and backward deterministic.*

If the number of non-blank symbols on the tape is finite, the infinite length tape can be uniquely represented by a triple $\langle l, s, r\rangle$ where l and r hold the left and right non-blank portions of the tape (in the form of lists) with respect to the tape head, and s contains the symbol under the tape head. To realize an infinite tape in finite space in the reversible setting, we require that only non-blank symbols can be on the bottom of either tape stack, cf. [18].

Let *step* be an implementation of the transition relation, such that *step* takes a configuration (a pair of the current state and tape) and returns the next configuration. Given a sequence of state transition rules d^*, we obtain

$$\begin{aligned} \mathit{step}\ \langle q, t\rangle \ &\triangleq\ \textbf{case}\ \langle q, t\rangle\ \textbf{of} \\ &\qquad \mathcal{T}[\![d]\!]^* \end{aligned} \tag{27}$$

where $\mathcal{T}$ is a translator from a state transition rule to the corresponding case branch in the proposed language, defined as:

$$\begin{aligned} \mathcal{T}[\![\langle q_1, \langle s_1, s_2\rangle, q_2\rangle]\!] &= \langle \overline{q_1}, \langle l, \overline{s_1}, r\rangle\rangle \rightarrow \langle \overline{q_2}, \langle l, \overline{s_2}, r\rangle\rangle \\ \mathcal{T}[\![\langle q_1, \leftarrow, q_2\rangle]\!] &= \langle \overline{q_1}, t'\rangle \rightarrow \textbf{let}\ t' = \mathit{move}_l\ t''\ \textbf{in}\ \langle \overline{q_2}, t''\rangle \\ \mathcal{T}[\![\langle q_1, \rightarrow, q_2\rangle]\!] &= \langle \overline{q_1}, t'\rangle \rightarrow \textbf{rlet}\ t'' = \mathit{move}_l\ t'\ \textbf{in}\ \langle \overline{q_2}, t''\rangle \\ \mathcal{T}[\![\langle q_1, \downarrow, q_2\rangle]\!] &= \langle \overline{q_1}, t'\rangle \rightarrow \langle \overline{q_2}, t'\rangle \end{aligned} \tag{28}$$

An overlined state or symbol $\bar{\cdot}$ represents a corresponding constructor in the language. Because of the reversibility of the source RTM, leaf left-expressions as well as the patterns appearing in the branches of *step* are disjoint, leading to the reversibility of *step*.

As an example, consider the incrementation of a non-negative binary number yielding its successor in binary representation (with the least significant digit first), cf. [18]. An RTM computing this function is shown in Fig. 8. It works as follows: Initially, the tape head is to the left of the first bit. The tape head then moves to the right, flipping bits until it flips a 0 to a 1, and then returns to the original position. It is easily verified that the machine in Fig. 8 is reversible. The rules given in Fig. 8 are translated into function *step* in Fig. 9.

$$\begin{array}{cccc}
\langle q_s, \langle b, b\rangle, q_1\rangle & \langle q_2, \langle 0, 1\rangle, q_3\rangle & \langle q_2, \langle b, b\rangle, q_3\rangle & \langle q_4, \langle 0, 0\rangle, q_3\rangle \\
\langle q_1, \rightarrow, q_2\rangle & \langle q_2, \langle 1, 0\rangle, q_1\rangle & \langle q_3, \leftarrow, q_4\rangle & \langle q_4, \langle b, b\rangle, q_f\rangle
\end{array}$$

Fig. 8. Transition rules for binary number incrementation

$$\begin{array}{lll}
step\ \langle q, t\rangle & \triangleq & \mathbf{case}\ \langle q, t\rangle\ \mathbf{of} \\
 & & \langle \overline{q_s}, \langle l, \overline{b}, r\rangle\rangle \rightarrow \langle \overline{q_1}, \langle l, \overline{b}, r\rangle\rangle \\
 & & \langle \overline{q_1}, t'\rangle \qquad\quad \rightarrow \mathbf{rlet}\ t' = move_l\ t''\ \mathbf{in}\ \langle \overline{q_2}, t''\rangle \\
 & & \langle \overline{q_2}, \langle l, \overline{0}, r\rangle\rangle \rightarrow \langle \overline{q_3}, \langle l, \overline{1}, r\rangle\rangle \\
 & & \langle \overline{q_2}, \langle l, \overline{1}, r\rangle\rangle \rightarrow \langle \overline{q_1}, \langle l, \overline{0}, r\rangle\rangle \\
 & & \langle \overline{q_2}, \langle l, \overline{b}, r\rangle\rangle \rightarrow \langle \overline{q_3}, \langle l, \overline{b}, r\rangle\rangle \\
 & & \langle \overline{q_3}, t'\rangle \qquad\quad \rightarrow \mathbf{let}\ t'' = move_l\ t'\ \mathbf{in}\ \langle \overline{q_4}, t''\rangle \\
 & & \langle \overline{q_4}, \langle l, \overline{0}, r\rangle\rangle \rightarrow \langle \overline{q_3}, \langle l, \overline{0}, r\rangle\rangle \\
 & & \langle \overline{q_4}, \langle l, \overline{b}, r\rangle\rangle \rightarrow \langle \overline{q_f}, \langle l, \overline{b}, r\rangle\rangle
\end{array}$$

Fig. 9. Function *step* generated from the transition rules in Fig. 8

Function $move_l$ moves the tape head one cell to the left. Thus, when it is called in an **rlet**-expression, it moves the tape head to the right. Function $move_l$ is defined as:

$$\begin{array}{lll}
move_l\ \langle l, s, r\rangle & \triangleq & \mathbf{let}\ r' = pushtape\ \langle s, r\rangle\ \mathbf{in} \\
 & & \mathbf{rlet}\ l = pushtape\ \langle s', l'\rangle\ \mathbf{in} \\
 & & \langle l', s', r'\rangle
\end{array} \tag{29}$$

where *pushtape* $\langle s, stk\rangle$ pushes symbol s to stack (list) *stk*:

$$\begin{array}{lll}
pushtape\ \langle s, stk\rangle & \triangleq & \mathbf{case}\ \langle s, stk\rangle\ \mathbf{of} \\
 & & \langle \overline{b}, [\,]\rangle\ \rightarrow [\,] \\
 & & \langle s', tl\rangle \rightarrow s' : tl
\end{array} \tag{30}$$

When symbol s is a blank and stack *stk* is empty, *pushtape* leaves *stk* empty. Otherwise, s is pushed on *stk*. Conversely, when *pushtape* is inversely invoked with an empty stack, a blank symbol is popped. This operation can be repeated arbitrary times. This preserves the condition that the bottom element of a stack is non-blank, which enables the representation of the infinite length tape in reversible finite space.

To simulate the RTM we must apply *step* repeatedly until we reach the final state q_f. If this is naïvely implemented, it results in a many-to-one (non-invertible) mapping which is not a reversible function. To cope with this problem we add an extra (intermediate) element to the output. For an RTM running forward, in addition to the result we also return a natural number which counts the number of applications of *step*. A simulation of the RTM is defined by the function rtm_f. Given a pair of the state and tape $\langle q, t\rangle$, rtm_f returns a triple containing the state, tape, and counter.

$$\begin{aligned} rtm_f\ \langle q,t\rangle \ \triangleq\ & \mathbf{case}\ q\ \mathbf{of} \\ & \overline{q_f} \to \langle \overline{q_f}, t, Z\rangle \\ & q_1 \to \mathbf{let}\ \langle q_2, t_2\rangle = step\ \langle q_1, t\rangle\ \mathbf{in} \\ & \qquad \mathbf{let}\ \langle q_3, t_3, n\rangle = rtm_f\ \langle q_2, t_2\rangle\ \mathbf{in} \\ & \qquad \langle q_3, t_3, S(n)\rangle \end{aligned} \tag{31}$$

A simulation of the inverse RTM is similarly defined by rtm_b:

$$\begin{aligned} rtm_b\ \langle q,t\rangle \ \triangleq\ & \mathbf{case}\ q\ \mathbf{of} \\ & \overline{q_s} \to \langle \overline{q_s}, t, Z\rangle \\ & q_1 \to \mathbf{rlet}\ \langle q_1, t\rangle = step\ \langle q_2, t_2\rangle\ \mathbf{in} \\ & \qquad \mathbf{let}\ \langle q_3, t_3, n\rangle = rtm_b\ \langle q_2, t_2\rangle\ \mathbf{in} \\ & \qquad \langle q_3, t_3, S(n)\rangle \end{aligned} \tag{32}$$

Here, **rlet** allows us to access to the inverse semantics explicitly, so that function *step* is used in the backward direction. This unconventional *code sharing* is a unique feature of reversible languages. Seeing as the underlying RTM is the same, rtm_f and rtm_b will recurse exactly the same number of times when applied to an input and the corresponding output, respectively. Because of this we can apply a recent input-erasing reversible simulation which removes the garbage counters, and which is twice as fast as Bennett's general method [19]. In fact, we directly obtain the following optimized RTM simulation:

$$\begin{aligned} rtm\ t\ \triangleq\ & \mathbf{case}\ \lfloor\langle t\rangle\rfloor\ \mathbf{of} \\ & \langle t_1, t_2\rangle \to \mathbf{let}\ \langle \overline{q_f}, t_3, n\rangle = rtm_f\ \langle \overline{q_s}, t_1\rangle\ \mathbf{in} \\ & \qquad \mathbf{rlet}\ \langle \overline{q_s}, t_2, n\rangle = rtm_b\ \langle \overline{q_f}, t_4\rangle\ \mathbf{in} \\ & \qquad \mathbf{case}\ \lfloor\langle t_3, t_4\rangle\rfloor\ \mathbf{of} \\ & \qquad\quad \langle t'\rangle \to t' \end{aligned} \tag{33}$$

which is a function of tapes to tapes (without the counter) corresponding exactly to the RTM defined by *step*.

For any RTM the above translation is obviously possible, and so the language is r-Turing complete: it is a universal reversible language.

3 Conclusion

We proposed a simple first-order reversible functional language. We view reversibility as both *global and local determinism* in both execution directions, and specified corresponding properties for an operational semantics for the functional lanaguage. Reversibility is achieved by reversible matching and syntactic restrictions (including linearity of variables). Using a novel symmetric first-match policy for pattern matching, the backward semantics of the proposed language is deterministic even in the case of overlapping leaf left-expressions, which enables concise code. The proposed reversible functional language is universal, as powerful as reversible Turing machines.

Every reversible computation model, be it reversible Turing machines [5,3], reversible cellular automata [15], or reversible logic circuits [7,17], have their own languages to describe how computation is organized. It is our hope that this language can serve as a basis for further research on reversible computing in the functional setting, similar to how Janus is used in the imperative setting. For example, Janus has been used for partial evaluation of a reversible language [14], synthesizing reversible circuits [17, Chapter 3] and translation of reversible languages [2].

Acknowledgments. This work is in part supported by Nanzan University Pache Research Subsidy I-A-2 for the 2011 academic year.

References

1. Abramov, S.M., Glück, R.: Principles of Inverse Computation and the Universal Resolving Algorithm. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 269–295. Springer, Heidelberg (2002)
2. Axelsen, H.B.: Clean Translation of an Imperative Reversible Programming Language. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 144–163. Springer, Heidelberg (2011)
3. Axelsen, H.B., Glück, R.: What Do Reversible Programs Compute? In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 42–56. Springer, Heidelberg (2011)
4. Baker, H.G.: NREVERSAL of Fortune — The Thermodynamics of Garbage Collection. In: Bekkers, Y., Cohen, J. (eds.) IWMM 1992. LNCS, vol. 637, pp. 507–524. Springer, Heidelberg (1992)
5. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development 17(6), 525–532 (1973)
6. Bowman, W.J., James, R.P., Sabry, A.: Dagger traced symmetric monoidal categories and reversible programming. In: De Vos, A., Wille, R. (eds.) 3rd Workshop on Reversible Computation, pp. 51–56. University of Gent. (2011)
7. De Vos, A.: Reversible Computing: Fundamentals, Quantum Computing, and Applications. Wiley-VCH (2010)
8. Frank, M.P.: Reversibility for efficient computing. Ph.D. thesis, EECS Dept. MIT, Cambridge, Massachusetts (1999)
9. Glück, R., Kawabe, M.: A Program Inverter for a Functional Language with Equality and Constructors. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 246–264. Springer, Heidelberg (2003)
10. Glück, R., Kawabe, M.: A method for automatic program inversion based on LR(0) parsing. Fundamenta Informaticae 66(4), 367–395 (2005)
11. Gries, D.: Inverting Programs. In: The Science of Programming. Texts and Monographs in Computer Science, ch. 21, pp. 265–274. Springer, Heidelberg (1981)
12. Lutz, C.: Janus: a time-reversible language. Letter to R. Landauer (1986), `http://www.tetsuo.jp/ref/janus.html`

13. McCarthy, J.: The inversion of functions defined by Turing machines. In: Shannon, C.E., McCarthy, J. (eds.) Automata Studies, pp. 177–181. Princeton University Press (1956)
14. Mogensen, T.Æ.: Partial evaluation of the reversible language Janus. In: Proceedings of Partial Evaluation and Program Manipulation, pp. 23–32. ACM Press (2011)
15. Morita, K.: Reversible computing and cellular automata — A survey. Theoretical Computer Science 395(1), 101–131 (2008)
16. Mu, S.C., Hu, Z., Takeichi, M.: An Injective Language for Reversible Computation. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 289–313. Springer, Heidelberg (2004)
17. Wille, R., Drechsler, R.: Towards a Design Flow for Reversible Logic. Springer, Heidelberg (2010)
18. Yokoyama, T., Axelsen, H., Glück, R.: Principles of a reversible programming language. In: Proceedings of Computing Frontiers, pp. 43–54. ACM Press (2008)
19. Yokoyama, T., Axelsen, H.B., Glück, R.: Optimizing clean reversible simulation of injective functions. Journal of Multiple-Valued Logic and Soft Computing 18(1), 5–24 (2012)
20. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Proceedings of Partial Evaluation and Semantics-Based Program Manipulation, pp. 144–153. ACM Press (2007)