Theoretical Computer Science 953 (2023) 113429

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Reversible computing from a programming language perspective ☆

Robert Glück [a,*], Tetsuo Yokoyama [b]

[a] DIKU, Department of Computer Science, University of Copenhagen, Denmark
[b] Department of Electronics and Communication Technology, Nanzan University, Japan

ARTICLE INFO

Article history:
Received 21 February 2022
Received in revised form 16 April 2022
Accepted 6 June 2022
Available online 22 December 2022

Keywords:
Reversible computing
Reversible programming
Function injectivization
Program reversibilization
Program inversion
Inverse interpretation
Compute-uncompute
Metacomputation

ABSTRACT

Software plays a central role in all aspects of reversible computing systems, and a variety of reversible programming languages have been developed. This presentation highlights the principles and main ideas of reversible computing viewed from a programming language perspective with a focus on clean reversible languages. They are the building material for software that can reap the benefits of reversible hardware and interesting in their own right.

Reversible computing is situated within programming languages in general, and the relevant concepts are elaborated, including computability, injectivization and reversibilization. Features representative for many reversible languages are presented, such as reversible updates, reversible iterations, and access to a program's inverse semantics. Metaprogramming methods of particular importance to reversible programming, are introduced, including program inversion and inverse interpretation. Our presentation is independent of a particular language, although primarily the reversible language, Janus, will be used in examples.

1. Introduction

Software plays a central role in all aspects of reversible computing systems, and a variety of reversible programming languages have been developed. In this contribution, we highlight the principles and main ideas of reversible computing viewed from a programming language perspective with a focus on clean (garbage-free) reversible languages. As with the interpretation of quantum mechanics, there has been no uniform interpretation of what reversible computing is. What we present is the "Copenhagen interpretation" of reversible computing.

Reversible computing is the study of computation models that are deterministic in both the forward and backward directions. It is a distinct computation paradigm that complements but does not replace conventional (irreversible) computing. Exploring this domain fills a blank spot on the map of computing and is interesting in its own right. Sometimes reversibility is a necessity, as in the case of quantum- and bio-inspired computing models when erasure is considered harmful (*cf.* [70]).

Landauer's discovery of a physical limit of computing is the original motivation for reversible computing [73]—experimentally observed later [19]—namely, that information loss by irreversibility directly contributes to heat dissipation, and thus power consumption, and Bennett's finding that there is nothing inherently irreversible about computation [17]. Following

☆ This article belongs to Section C: Theory of natural computing, Edited by Lila Kari.

* Corresponding author.
E-mail addresses: glueck@acm.org (R. Glück), tyokoyama@acm.org (T. Yokoyama).

https://doi.org/10.1016/j.tcs.2022.06.010

Paradigms:	fwd →	bwd ←
• Logic languages	N	N
• Mainstream languages	D	N
• Reversible languages	D	D

Fig. 1. Programming language paradigms and the forward and backward determinism of their computation models (N...nondeterministic, D...deterministic).

the investigations into reversible logic circuits [32,36,107] and reversible cellular automata [63,89,106], reversible processors for programmable devices have been considered [13,35,104,105,110]. Reversible programming languages are the material required to build the novel software that can actuate energy-efficient reversible hardware. Even without reversible hardware, there are applications such as high-performance computing [21,101,102] and reverse debugging, *e.g.*, supported by The GNU Project Debugger GDB [54].

We aim at presenting a coherent view of reversible computing using familiar programming language concepts. This gentle exposition grew out of an invited talk with the same title [53] that reflected upon our investigations of reversible languages during the past decade (*e.g.*, [14,114]) and related explorations of inversion problems (*e.g.*, [2,45]). Knowing that many topics cannot be dealt with in a talk and its written form, and that, occasionally, a choice is necessary, we provide a selection of references to the literature for more details.

The unusualness of reversible programming motivates developing an explanatory framework from first principles. We hope that the concepts of clean reversible computing are presented in sufficient clarity and without overburdening the reader with formalization. An effective and meaningful ordering instead of merely collecting some observations and results can enable further advances and developments in reversible computing. A deeper understanding of reversible computing could also be of great benefit in quantum computing because operations on a quantum state are unitary and thus reversible [23,31].

The present contribution offers a fresh and coherent view of reversible computing from a programming language perspective and complements previous presentations on the subject (*e.g.*, [23,89,97]).

The presentation is organized as follows. We begin by situating reversible computing within programming languages in general (Sect. 2), develop our main hypotheses (Sect. 3), elaborate on computability, injectivization, and reversibilization (Sect. 4), set cornerstones of reversible languages (Sect. 5), introduce metacomputation methods of particular relevance for reversible programming (Sect. 6), present a variety of reversible languages (Sect. 7), and conclude that reversible languages are a rich field of its own (Sect. 8). Throughout this contribution, we assume that readers are familiar with the basic concepts of programming language theory and practice.

2. Situating reversible computing

Mainstream languages, such as C and Java, are deterministic in the forward direction of computation and nondeterministic in the backward direction.[1] That is, at each state during the computation of a program, the next state, if any, is uniquely determined but not always the previous state. For instance, it is clear which branch of a conditional to choose, but after the conditional, in general, we cannot say which of the branches was chosen. Because their computation is forward deterministic (D) and backward nondeterministic (N), we categorize many of today's mainstream languages as (D, N).

Logic languages overcome the directionality of mainstream languages by giving no preference to any of the computation directions and by specifying the relationship between the input and output. Given the possibility of nondeterminism in both directions, we can classify logic languages as (N, N).[2] These languages aim at separating the logic and control aspects of programs [69].

Reversible languages are deterministic in both directions. At each state during a computation, both the next *and* the previous states, if any, are uniquely determined. We categorize reversible languages as (D, D). Reversible computing studies computation models that are deterministic in both the forward and backward directions.

Sometimes reversibility is a necessity, as in the case of quantum computing models [23][3] and overcoming Landauer's physical limit [73]. When computers are built based on fundamentally irreversible models, information loss carries over into the physical implementation. Thermodynamics tells us that these irreversible machines necessarily dissipate heat [24,32]. Consequently, the part of the power costs associated with destructive, information lossy computations can theoretically be avoided. For example, Turing machines, which are (D, N)-machines, are information lossy, and information destruction is an intrinsic part of this computation model, while reversible Turing machines [17], which are (D, D)-machines, are not.

Figure 1 summarizes, in a table, how the three computation models can be distinguished according to their forward and backward determinism.[4]

[1] We focus on the core of these languages. Note that constructs having a notion of parallelism, such as threads, are inherently forward nondeterministic.

[2] In general, logic programs allow for multiple directions of computation, not only two, depending only on the queries that are feasible for the program.

[3] Operations on a quantum state are unitary, hence invertible and reversible.

[4] Languages classified as (N, D) are those that have the transitions of mainstream languages "turned around," that is, their forward direction is nondeterministic, while their backward direction is deterministic. Their application has not been studied in depth; the closest is the idea of using nondeterministic languages with deterministic backtracking [33]. Another example is the reversible nondeterministic finite automaton (REV-NFA) [60].

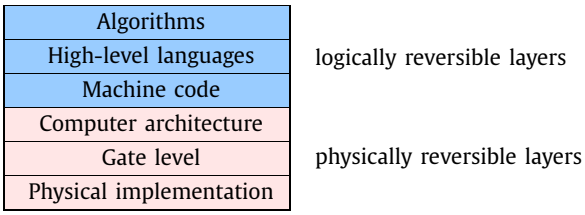


Fig. 2. The hardware and software stack of a reversible computing system.

3. Five main hypotheses

First, we hypothesize that reversible computing principles manifest themselves in different guises on *all levels of a reversible computing system*, ranging from reversible hardware to reversible software, and across all language abstractions from low-level machine code to high-level languages and algorithms (illustrated in Fig. 2).

Second, our investigation into reversible computing in a *100% clean reversible setting* follows the observation that it is easier to dilute than to distill essences. This is consistent with the scientific principle of effectively studying the "subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects" [27].

Third, we advocate that *reversible computing systems be programmable* so that software makes the underlying hardware perform a variety of functions. Software is easier to create, modify, and duplicate than a physical implementation and can be investigated independently of the reversible hardware. This in turn requires the design and development of suitable reversible computer architectures as an interface between the hardware and the software components that make up a reversible computing system, including domain-specific and subuniversal reversible systems.

Fourth, the linguistic material from which reversible software is built are *reversible programming languages*. As is all too well known from mainstream languages, there is no "best" reversible language. We hypothesize that the development of reversible software systems is best served by investigating reversible languages horizontally across different language paradigms (*e.g.*, object-oriented, functional, and imperative languages) and vertically across different abstraction levels of the computation stack (*e.g.*, machine, intermediate, and high-level languages). Their *unusual computing properties* are the primary focus on our investigation, and we postpone concerns for feature-complete languages.

Our focus is on semantically clean reversible languages and computation models, like the *reversible Turing machines*, whose programs are composed in ways that preserve the injectivity of their program units and compute injective functions without semantic modification. This guarantees full transparency and control over garbage data at the programming language level. The rationale behind this approach is straightforward: We want to be able to build 100% clean reversible computing systems that can compute injective functions without global "undo" mechanisms such as tracing or rollbacks.

As pointed out in Sect. 2, (D, D)-languages are a distinct and unique class of programming languages; they reflect the properties of the underlying reversible computation model and can only implement injective (1:1) computable functions. Naturally, we are particularly interested in the *computationally most powerful* of the reversible languages, *i.e.*, those that are computationally as powerful as the reversible Turing machines, and acknowledge specific subclasses, such as reversible automata [99] and primitive recursive functions [95], and several types of reversible Turing machines [91,92].

Fifth, *reversible programs are data objects that can be generated, analyzed, and transformed* in various ways through programs. Of particular relevance in this context are efficient inverse interpreter and program inversion, clean (*i.e.*, semantics-preserving) translation between reversible languages on all levels [15,116], and the non-trivial reversibilization of irreversible programs. Known metacomputation methods, such as the Futamura projections [37], establish the relation of inverse interpreters and program inverters and lend themselves to reversible-program specific transformation schemes (*e.g.*, [4]).

The cornerstones of the "Copenhagen interpretation" of reversible computing are summarized by our five main hypotheses that

1. reversible computing principles manifest themselves in different guises on all levels of reversible computing systems, independently of language paradigms and abstraction levels,
2. their essence is best studied in depth in a 100% clean reversible setting,
3. programmable systems will be of chief importance to their application,
4. the linguistic material from which reversible software is built are reversible programming languages, and
5. reversible programs are effectively treatable as data objects that can be generated, analyzed, and transformed.

From these considerations, it follows that the investigations into reversible software systems comprise a wide range of interrelated aspects, including computability and complexity theory, computation models, algorithms, software systems, and programming languages and techniques. On the part of the reversible hardware, the nature of the concrete physical implementations including their design, fabrication, and architectures is a necessary complementary research endeavor.

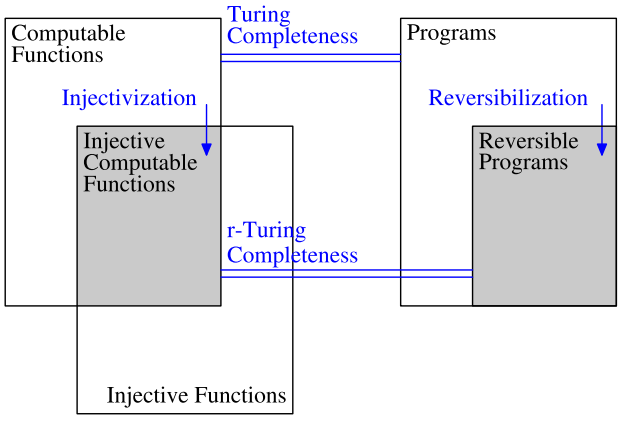


Fig. 3. Mathematical functions and reversible programs.

4. Computability, injectivization, and reversibilization

From a computational perspective, mainstream languages are Turing-complete, *i.e.*, they are as powerful as classical Turing machines. Whereas reversible languages are at best *r-Turing-complete* [9], *i.e.*, they are as powerful as *reversible Turing machines* [17]. Given unbounded resources, reversible languages can realize any *injective computable functions*, which are only a proper subset of the computable functions. This may seem a severe limitation, but any computable function can be embedded in an injective computable function. For reversible programming, this means that either the functional specification of a program must be *injectivized* before writing the reversible program or an irreversible program must be *reversibilized* into a reversible program, *e.g.*, by recording the computation history.

Injectivization of functions and reversibilization of programs are always possible but at the expense of semantic modifications and operational overhead. The relationships between the discussed concepts are shown in Fig. 3.

In the remainder of this section, we shall discuss, in more detail, how *injectivization* and *reversibilization* bring us from the familiar mainstream world of programming to the reversible world.

Notation The notation is adapted from [61]. For any program text p, written in language L, we let $[\![p]\!]_L\, x$ denote the application of L-program p to its input x. Note that this notation distinguishes between the program text p and the function $[\![p]\!]_L$ implemented by p. In the rest of this contribution, we suppose that languages R and R' are reversible, and the other languages such as L, T, P, and so forth are irreversible.

As is customary, we use the same universal data domain D for all languages and for representing all programs (*e.g.*, lists are a suitable choice, as known from Lisp). Mappings between different data domains are straightforward and not essential for our discussion.

Equality between program applications shall always mean strong equivalence: either both sides of an equation are defined and equal, or both sides are undefined. Suppose we have two functionally equivalent programs p and q written in languages P and Q, respectively. Then, we have $\forall x:\ [\![p]\!]_P\, x = [\![q]\!]_Q\, x$, and we write in short $[\![p]\!] = [\![q]\!]$. We often omit the language subscripts when they are clear from the context. A program p is said to be *injective* iff, whenever $[\![p]\!]\, x_1 = [\![p]\!]\, x_2$ and both sides are defined, it must be the case that $x_1 = x_2$.

We use conventional notation for denoting functions. Unless stated otherwise, they are assumed to be partial and defined over countable domains.

4.1. Injectivization of functions

Given a function f, we can define a function $\overline{f}^g$ by pairing f with some function g,

$$\overline{f}^g\, x \stackrel{\text{def}}{=} (f\, x, g\, x), \tag{1}$$

and we call $\overline{f}^g$ an *injectivization of f with respect to g* if $\overline{f}^g$ is injective.[5] We consider three injectivizations of f: the trivial embedding, where g preserves x; the function-specific injectivization, where g exploits the specific properties of f; and the reversible update, where g is a projection. We write $f \trianglelefteq h$ if and only if h is an injectivization of f. The symbol $\trianglelefteq$ reflects the intuition that f is "less than or equal to" h after an injectivization. We write $\overline{f}$ if g is clear from the context or not relevant to the discussion.

[5] Note that g can be a constant function, such as $g(x) = 0$, if f is injective, and that x and y can be tuples $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$, $m, n \geq 0$. For the sake of concreteness, we use scheme (1). There are other schemes, *e.g.*, a permutation $(g\, x,\ f\, x)$.

Every (non-injective) function f can be embedded in an injective function $\overline{f}^{id}$ by pairing input x and output $f\ x$ such that $\overline{f}^{id}$ is defined by

$$\overline{f}^{id}\ x \stackrel{\text{def}}{=} (f\ x, x). \tag{2}$$

This *trivial embedding* of f in $\overline{f}^{id}$ is always possible: $\forall f : f \trianglelefteq \overline{f}^{id}$. Although it is conceptually easy to pair the input with the desired output, this can be impractical when composing multiple trivially injectivized functions.

More sophisticated embeddings are often feasible by exploiting the specific properties of a function. They may serve particular applications better than a trivial embedding. A *function-specific injectivization* can be closer to the intention of the original specification than preserving the entire input in the output of the function. For example, let *fib* x be the Fibonacci function on non-negative integers. *fib* is not injective because the first two integers are mapped to the same Fibonacci number.[6] Then, the definition

$$\mathit{fibpair}\ x \stackrel{\text{def}}{=} (\mathit{fib}\ x, \mathit{fib}(x+1)) \tag{3}$$

embeds the non-injective function *fib* in an injective function *fibpair* that computes two consecutive Fibonacci numbers: $\mathit{fib} \trianglelefteq \mathit{fibpair}$.[7] The new function *fibpair* is injective because there is a unique Fibonacci pair for each index ($x \geq 0$): $\mathit{fibpair}\ 0 = (1, 1)$, $\mathit{fibpair}\ 1 = (1, 2)$, and so forth.

In practice, injectivization (3) may come handy when generating a sequence of Fibonacci numbers in a reversible program because it avoids the need for preserving the input x in the output as in a trivial embedding (2). We will use this injectivization in Sect. 5.1 to show how to implement an injective function in a reversible language.

There are many ways of injectivizing functions, and the choice depends on the particular application and the properties of the function. Sometimes it is useful to "think outside the box" because a problem can have an injective specification, although it is not considered as such. For example, the string-matching problem is to find all positions *pos* of a pattern p in a text t. Normally, we do not want to delete t and p but keep them, so we can locate p in t. Thus, the classic string-matching problem has a natural input-preserving specification [50]:

$$\mathit{match}(t, p) = (\mathit{pos}, (t, p)). \tag{4}$$

We use the term *injectivization* in a general sense. We regard any injective function $\overline{f}$ as an injectivization of f, and write $f \trianglelefteq \overline{f}$, provided we can extract from $\overline{f}\ x$ the result of $f\ x$ for all x in the domain of f. There can also be mappings between the domains of the functions. For example, $\overline{f}$ can be between two mappings α and β, that is, $\beta(\overline{f}(\alpha\ x)) = f\ x$. This means that $\overline{f}$ embeds the full functionality of f in one way or another. An injectivization is not required to modify an injective function. If f is injective, we can have $f = \overline{f}$, implying an injectivization can be idempotent, *i.e.*, $\overline{f} = \overline{\overline{f}}$ in $\overline{f} \trianglelefteq \overline{\overline{f}}$.

For reversible computing theory, it is important that there always exists an embedding of a (non-injective) function f in an injective function $\overline{f}$:

$$\forall f : \exists \overline{f} : f \trianglelefteq \overline{f}. \tag{5}$$

It should be kept in mind that the embedding of a non-injective function in an injective function inevitably entails a *semantic change*, that is, the non-injective function and its injective version have different codomains. Consider the two embeddings (2), (3), we clearly have the inequalities

$$f \neq \overline{f}^{id} \text{ and } \mathit{fib} \neq \mathit{fibpair}. \tag{6}$$

No matter how a non-injective function is injectivized, some extra information is needed to disambiguate all those cases where multiple inputs are mapped to the same output. It is the nature of a non-injective function that the size of its image is smaller than that of its domain of definition. Consider the two cases $\mathit{fib}\ 0 = 1$ and $\mathit{fib}\ 1 = 1$, where the original input, 0 or 1, cannot be uniquely determined from the output 1.

Minimizing garbage An important practical goal is to minimize the quantity of information added to the output. Reducing the extra output ("*garbage*") is essential to save resources such as space and time when computing an injectivized function. Instead of indiscriminately adding the entire input to the output, as in the trivial embedding (2), one can aim at *minimizing* the quantity of disambiguating information measured in bits for instance.

In general, there is no upper bound on the extra data that is needed for the injectivization of a function. Calculating the minimum amount needed is an undecidable problem. It is not even decidable whether a function is injective, that is no extra information is needed for the injectivization.[8]

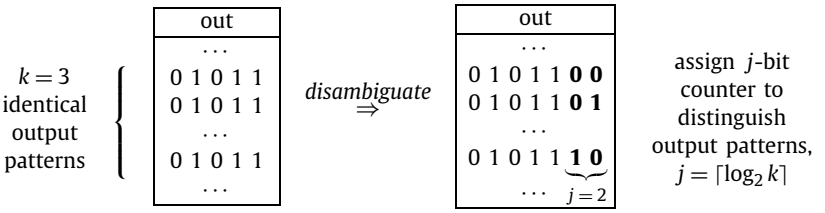
In the special case of a total function over a finite domain, for instance, in the case of the Boolean function of a combinational circuit, the *minimum number of garbage bits* can be calculated [107].

[6] For simplicity of the presentation, we start the sequence from 1 and 1 as the first two Fibonacci numbers, *i.e.*, 1, 1, 2, 3, 5, ..., and the indexing from 0.

[7] Here, $\mathit{fibpair} = \overline{\mathit{fib}}^{g}$, where $g(x) = \mathit{fib}(x+1)$.

[8] Consider function $\mathit{zero}(0) \stackrel{\text{def}}{=} 0, \mathit{zero}(1) \stackrel{\text{def}}{=} 0$ if p. In general, it is undecidable whether predicate p terminates and, thus, whether *zero* is injective [49].

Table 1
Minimum number of garbage bits added to disambiguate identical output patterns of a Boolean function.



Example 1 *(Minimum garbage bits).* Consider a Boolean function that maps a vector of n bits to a vector of m bits ($n > m$). Because of the reduction of the domain, the function cannot be injective. If the maximum number of occurrences of identical output patterns is k, the minimum number of garbage bits required to make the truth table injective is $\lceil \log_2 k \rceil$. This is the minimum number of bits needed to injectivize the function. For example, if $k = 3$, then $\lceil \log_2 3 \rceil \approx \lceil 1.6 \rceil = 2$ garbage bits suffice to disambiguate this and all other identical output patterns, as illustrated by the truth table in Table 1. All output patterns are padded with the same number of garbage bits because bit vectors are used; in general, this is not a requirement.

An important observation is that non-injective functions can be *injective in some of their arguments*. This can be exploited to improve the trivial embedding (2) by not adding all inputs to the output but just those that make the function injective. A classic example is addition.

Example 2 *(Injectivization of addition).* Integer addition $+(a, b) \stackrel{\text{def}}{=} a + b$ is not injective but *injective in its first (resp. second) argument*.[9] Instead of a trivial embedding that returns both inputs,

$$\overline{+}^{id}(a, b) \stackrel{\text{def}}{=} (a + b, (a, b)), \tag{7}$$

an injectivization only needs to return one of them:

$$\overline{+}^{fst}(a, b) \stackrel{\text{def}}{=} (a + b, a) \quad \text{or} \tag{8}$$

$$\overline{+}^{snd}(a, b) \stackrel{\text{def}}{=} (a + b, b). \tag{9}$$

Injective functions that modify some of their inputs and leave others unchanged are collectively known as *reversible updates* [13, Sect. 3]. A reversible update is an instance of the injectivization scheme (1), where g is a *projection* that selects some of the inputs. An example is $\overline{+}^{fst}$ in (8) that updates b by adding a to b and returns the first argument $a = fst(a, b)$ unchanged. Similarly, for $\overline{+}^{snd}$ in (9), with $b = snd(a, b)$.

Example 3 *(Toffoli gate as reversible update).* The Boolean function $f_3(x_1, x_2, x_3) = x_1 \oplus (x_2 \wedge x_3)$ is *injective in its first argument*. Thus, the projection $g(x_1, x_2, x_3) = (x_2, x_3)$ that drops the first argument is sufficient for the injectivization of f_3:

$$\overline{f_3}^{g}(x_1, x_2, x_3) \stackrel{\text{def}}{=} (f_3(x_1, x_2, x_3), (x_2, x_3)). \tag{10}$$

The reversible update $\overline{f_3}^{g}$ in (10) is exactly the function of a 3-bit *Toffoli reversible logic gate* [107], also *controlled-controlled-not gate* ($CCNOT = \overline{f_3}^{g}$), where x_1 is updated with exclusive-or $\oplus$, and x_2, x_3 are the two control lines passed through the gate. The gate is *self-inverse*, meaning that it is its own inverse, and *universal* for reversible combinational logic circuits, meaning that any such circuit can be built entirely from Toffoli gates, *i.e.*, from one particular reversible update. Reversible combinational circuits compute exactly the *bijective Boolean functions*, a proper subset of the partial injective functions.

Reversible updates are well suited to update parts of a computation state. They play a central role in reversible computing as elementary statements in reversible languages, such as the reversible assignments `x += e` and `x -= e`, and as elementary gates in reversible logic circuits, such as the Toffoli and Fredkin reversible logic gates [36,107]. All reversible logic gates, whether classical or quantum,[10] are cases of reversible updates [115].

In general, a reversible update is a *projection-based injectivization* of a function defined by a composition of (i) an update function that is injective in some of its arguments and (ii) an arbitrary function [13]. For example, the Toffoli gate $\overline{f_3}^{g}$ is a projection-based injectivization of f_3 defined by composing the update function $\oplus$ and expression $x_2 \wedge x_3$. We will meet reversible updates again when we introduce the reversible language Janus in Sect. 5.

[9] A partial function $f : (A \times B) \rightharpoonup C$ is *injective in its first argument* iff $\forall a, a' \in A : \forall b \in B :$ if $f(a, b)$ and $f(a', b)$ are defined, $f(a, b) = f(a', b) \implies a = a'$. This definition can be stated for any argument.

[10] For example, the standard transformation of quantum computing, $|y\rangle|x\rangle \mapsto |y\rangle|x \oplus f(y)\rangle$, is a reversible update.

4.2. Reversibilization of programs

In the previous section, we considered the injectivization of a *function*, so that it can be implemented as a program in a reversible language without semantic changes. This section considers the case when the functional specification is already implemented as a *program* in a mainstream language, and the irreversible program should be transformed into a reversible program. This reversibilization generally entails an injectivization of the function implemented by the irreversible program and thus a semantic change.

4.2.1. Transforming irreversible to reversible programs

Every program p written in an irreversible language L can be *reversibilized* into a program q written in a reversible language R provided R is r-Turing-complete. If p implements a non-injective function, then the reversibilization of p into q must *injectivize* the function implemented by p, that is, $[\![p]\!]_L \trianglelefteq [\![q]\!]_R$.

The reversibilization of programs can be carried out by a *reversibilizer*, i.e., a P-program *revbiz* such that for all p and q

$$[\![revbiz]\!]_P\ p = q \quad \text{implies} \quad [\![p]\!]_L \trianglelefteq [\![q]\!]_R. \tag{11}$$

A reversibilizer is a metaprogram like a translator that transforms programs from one language to another language. Unlike a translator, a reversibilizer does not require p and q to be functionally equivalent, $[\![p]\!]_L = [\![q]\!]_R$, but $[\![p]\!]_L \trianglelefteq [\![q]\!]_R$. The optimization problems are also quite different, such as minimizing the data and operations required for reversibility. Like a translator, a reversiblizer is characterized by a source, target, and implementation language, that is, L, R, and P in (11). The choice of the implementation language P is not essential for the functionality of the reversibilizer.

In general, a reversibilized program q of p returns some additional data z which ensures that $[\![p]\!]_L \trianglelefteq [\![q]\!]_R$:

$$[\![q]\!]_R\ x = ([\![p]\!]_L\ x, z). \tag{12}$$

Clearly, z can take many forms and represent a variety of information about p. Sometimes we write $\overline{p}$ as a shorthand notation for a reversibilization of p, that is, $[\![revbiz]\!]_P\ p = \overline{p}$.

In general, it is undecidable whether p implements an injective function because any non-trivial program property is undecidable by Rice's theorem. Thus, there are always injective programs whose semantics is modified by a reversibilizer even though, theoretically, this would not be necessary (for every injective function $[\![p]\!]_L$ there exists a functionally equivalent program q' in R, $[\![p]\!]_L = [\![q']\!]_R$, which is a *clean* implementation of p in R.)

Reversibilization is not the familiar semantics-preserving translation of a program from one language to another. A program's irreversible trajectory in the state space must be converted into a backward as well as forward deterministic trajectory. That is, in each state of a computation not only its successor state is uniquely defined but also its predecessor state [116].[11] This can be achieved by instrumenting the program with a trace, which will modify its semantics and complexity, whereas a translator must preserve both. In this case, z in (12) is a trace of p with respect to x.

Some reversibilization methods, such as Bennett's original method [17] (described in Sects. 4.2.2 and 6.3), are injective transformations, and thus they can be directly implemented in a reversible language.[12] The input-preserving L-to-R reversibilizer *ben* written in a reversible language R' is a program that transforms any L-program p into an R-program such that for all inputs x we have the characteristic equation

$$[\![[\![ben]\!]_{R'}\ p]\!]_R\ x = ([\![p]\!]_L\ x, x). \tag{13}$$

4.2.2. Two universal reversibilization methods

Two universally applicable methods for reversibilization are the trace-generating Landauer embedding and the input-preserving Bennett method.

Landauer embedding A way to reversibilize an irreversible program p is to record in a computation trace the (otherwise unrecoverable) information lost because of the non-injective operation of p, such as the values overwritten by assignments and the control-flow information of conditionals and loops. Trace-based reversibilizations are collectively known as *Landauer embedding* [73].

The reversible R-program $\overline{p}^{lan}$ that we obtain from an irreversible L-program p by a Landauer embedding yields a computation trace in addition to the original output $[\![p]\!]_L\ x$ for an input x:

$$[\![\overline{p}^{lan}]\!]_R\ x = ([\![p]\!]_L\ x, trace). \tag{14}$$

The contents of *trace* depend on how and in which language a program is implemented. If L is a flowchart language, tracing may include recording on a stack every conditional branch that is taken and the value of every variable before it is overwritten by an assignment [119, Sect. 2.5]. In the case of Turing machines, this may include recording on an extra tape the

[11] The initial and final states may not have predecessor and successor states, respectively.

[12] The source program is expanded into a reversibilized program; thus, Bennett's original method is injective: $\forall p_1, p_2 : [\![ben]\!]\ p_1 = [\![ben]\!]\ p_2 \implies p_1 = p_2$.

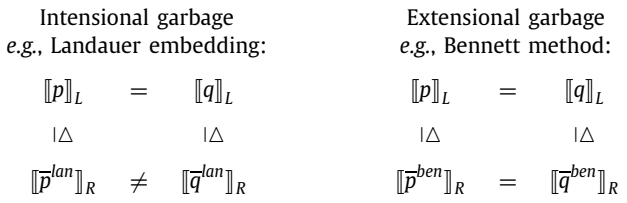


Fig. 4. Reversibilization of two functionally equivalent programs p, q with intensional (left) and extensional (right) garbage, and the program relations.

number of every transition rule that is applied during the computation [17, (9–11)]. The reverse C compiler [96] uses two tapes for tracing control-flow operators and destructive assignments; tape compression is used [97, Ch. 10]. Preprocessors can extend programs with deterministic reversal actions [20,77].

Bennett method Another way to reversibilize a program p is to preserve its input in the output. The reversibilized version $\overline{p}^{ben}$ then returns the original output $[\![p]\!]_L\ x$ together with the input x:

$$[\![\overline{p}^{ben}]\!]_R\ x = ([\![p]\!]_L\ x, x). \tag{15}$$

Although adding x to the output of $\overline{p}^{ben}$ may seem like a simple change from a functional viewpoint, implementing it in a reversible language is not. We should keep in mind that $\overline{p}^{ben}$ must be implemented in a reversible language R, which means it cannot be built from destructive (non-injective) statements like p, only from reversible statements. It is not sufficient to add a statement to $\overline{p}^{ben}$ that copies x to the output and otherwise run p with its irreversible statements.

Bennett showed how to construct a reversibilized program $\overline{p}^{ben}$ for every program p [17]: $\overline{p}^{ben}$ is constructed from $\overline{p}^{lan}$ by undoing the tracing by composing $\overline{p}^{lan}$ with its inverse program $(\overline{p}^{lan})^{-1}$ and copying the output $[\![p]\!]_L\ x$ returned by $\overline{p}^{lan}$ into the output of $\overline{p}^{ben}$.

This universal construction, also called *compute-uncompute*, which undoes the tracing of a program $\overline{p}^{lan}$ by composition with its inverse program $(\overline{p}^{lan})^{-1}$ and takes advantage of program inversion being straightforward in a reversible language, is one of the fundamental programming methods in reversible computing and also known as *Bennett's trick*. It will be explained in more detail later (Sect. 6.3) once we have introduced a reversible language (Sect. 5) and the principles of program inversion (Sects. 6.1 and 6.2).

4.2.3. Resource consumption, garbage, and backward runs

We now discuss aspects of reversibilization that are of particular relevance to reversible programming: the impact on the resource consumption, the properties of two types of garbage, and the backward run of reversibilized programs.

Resource consumption It is clear that reversiblization affects the resource consumption of a program because additional operations and space are needed at runtime to preserve the information that would otherwise be lost.

There are linear-time reversibilizations and linear-space reversibilizations. A Landauer embedding consumes time and space proportional to the run time of a program. For example, an iteration of irreversible statements over constant space $O(1)$ is turned into a traced iteration that consumes space proportional to the number of iterations k, hence $O(k)$. There are also reversibilizations that do not change the space complexity but consume exponential time [75]. In general, there is no input-independent bound on the resource overhead added to a program. Regarding input-dependent resources, the question remains whether there is a universal reversibility method that has both linear-time *and* linear-space overhead.

Methods that minimize the additional resource consumption are essential, and they come with the well-known trade-off between time and space (*e.g.*, [12,117]). We shall not go further into possible optimizations but refer to the various space-time trade-offs that can be achieved for the Bennett method by reversible pebbling strategies [18,98,111]. There are reversible pebbling strategies for directed acyclic graphs [71] that have also applications in the optimization of quantum circuits (*e.g.*, [83]). Another method is to transform programs into reversible programs with minimal garbage using a generate-and-test approach, but at the cost of a considerable runtime overhead [49].

Types of garbage The output added to a program is usually only required for reversibility and not interesting to a user, hence, often called *garbage*. We distinguish between two types of garbage: *intensional* and *extensional*.

1. *Intensional garbage* is specific to the *operations* performed by a program. The Landauer embedding is an example of this type of garbage. It traces the operations of a program and returns the trace in the program's output. Intensional garbage not only reveals information about the internals of a program, but two functionally equivalent programs, p and q, result in functionally different reversibilized programs, $\overline{p}^{lan}$ and $\overline{q}^{lan}$, when their implementations differ (*e.g.*, one being iterative and the other recursive). Thus, in general, intensional garbage turns a *functional equivalence* $[\![p]\!]_L = [\![q]\!]_L$ into a *functional non-equivalence* $[\![\overline{p}^{lan}]\!]_R \neq [\![\overline{q}^{lan}]\!]_R$, as depicted in Fig. 4. Neither is this theoretically welcome, nor practical, *e.g.*, when we want to upgrade to a new version q of p in a reversible context. An advantage of a Landauer embedding is the relative simplicity of adding a trace to a program.

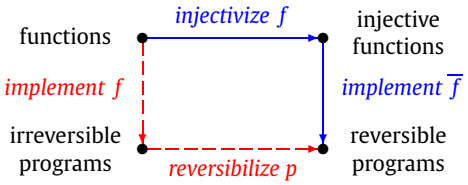


Fig. 5. Two strategies that lead from functional specifications to reversibilized programs.

2. *Extensional garbage* is specific to the *function* implemented by a program. An example is the Bennett method that adds the input to the output of a program. Two functionally equivalent programs, p and q, are also functionally equivalent after reversibilization with extensional garbage. In other words, a functional equivalence $[\![p]\!]_L = [\![q]\!]_L$ that is valid in the irreversible world remains valid, $[\![\overline{p}^{ben}]\!]_R = [\![\overline{q}^{ben}]\!]_R$, after reversibilization with extensional garbage as depicted in Fig. 4. What makes the Bennett method so useful is that it is a universal method to transform intensional into extensional garbage. Thereby, it avoids some of the disadvantages of the Landauer embedding. Examples of extensional garbage include the injectivizations in Sect. 4.1, *e.g.*, *fibpair*, $\overline{+}^{id}$, $\overline{+}^{fst}$, $\overline{+}^{snd}$, and $\overline{f_3}^{g}$, and comparison sorts with garbage in the form of permutations, factorial representations, and ranks [12].

Backward runs An important aspect of reversible computing is the possibility to run programs backward. We will now examine reversibilization from this perspective. A problem that affects both the Landauer embedding and the Bennett method: When a reversibilized program is run backward, additional data is required, either the trace or the original input.

Let y be the output of an L-program p for input x, that is, $[\![p]\!]_L\ x = y$. Given y, we can compute x by running a reversibilized program backward provided we also know the garbage associated with y. Here, we use the inverse programs $(\overline{p}^{lan})^{-1}$ of $\overline{p}^{lan}$ and $(\overline{p}^{ben})^{-1}$ of $\overline{p}^{ben}$ to perform the backward runs:

$$x = [\![(\overline{p}^{lan})^{-1}]\!]_R(y, trace), \tag{16}$$

$$x = [\![(\overline{p}^{ben})^{-1}]\!]_R(y, x). \tag{17}$$

There are practical obstacles in both cases with the backward runs:

In the first case (16), *trace* requires that we first run $\overline{p}^{lan}$ forward before we can run it backward. This is necessary because it is not easy to determine the correct *trace* for y without running $\overline{p}^{lan}$. Thus, practically speaking, $(\overline{p}^{lan})^{-1}$ can only serve as a *left inverse* of $\overline{p}^{lan}$. Also, the trace is specific to $\overline{p}^{lan}$ and cannot be reused for the backward run of another $\overline{q}^{lan}$. These issues apply to almost all of the reversibilized programs that return intensional garbage. A rare exception in which a trace can be reused with another program is the case of an encoder and a decoder with identical control flow [117].

In the second case (17), we are in the paradox situation that we must know x to compute x from y. While this can be very useful for reversible programming (*e.g.*, to uncompute y using x, which means we clear y), it is not practical for some important applications where both directions are needed (*e.g.*, to decode y to obtain an unknown x).

Thus, the two classic reversibilization methods are not always useful, in particular not in the case of stand-alone applications of $(\overline{p}^{lan})^{-1}$ and $(\overline{p}^{ben})^{-1}$.

4.2.4. Two strategies for developing reversible programs

We conclude from our discussion of injectivization (Sect. 4.1) and reversibilization (Sect. 4.2) above that it is generally more preferable in the development of reversible programs to

1. *injectivize* the functional specification of a program, and to
2. *implement* the injective specification in a reversible language

than to

1. *implement* the functional specification in an irreversible language, and to
2. *reversibilize* the irreversible program into a reversible language.

The two strategies for developing reversible programs are illustrated in Fig. 5 where the *injectivization strategy* is shown with a blue solid line, and the *reversibilization strategy* is shown with a red dotted line. In case an irreversible program is to be reversibilized during program development, then in many cases it is preferable that the reversibilized version

- generates *extensional* garbage rather than *intensional* garbage,

since we would like to talk about functionality and ignore the internal workings of a program.

The reversible programs that we obtain with these two strategies will generally differ in their implementation, resource consumption, and the garbage they generate. In general, their functionality is different: $[\![\overline{p}]\!]_R \neq \overline{[\![p]\!]_L}$, where $\overline{p}$ is a revers-

ibilization of p that implements the desired function $[\![p]\!]_L$, and $\overline{[\![p]\!]_L}$ is an injectivization of that function. Regardless of which strategy we choose, the original functionality is preserved: $fst([\![\overline{p}]\!]_R) = fst(\overline{[\![p]\!]_L})$. Clearly, when developing reversible programs, there is also a trade-off between the time and effort required to write a new program by hand in a reversible language and the time to automatically reversibilize an already existing irreversible program by a reversibilizer and the quality of the resulting programs.[13]

These considerations have been the guiding principles for our investigations into the development of reversible programs and algorithms.

4.2.5. Reversibilizing language semantics

Every programming language has a precise formal semantics to be executable. This opens a metalevel alternative to reversibilizing programs. Instead of reversibilizing each L-program individually, a new *reversible semantics* can be given to the implementation language L.

Reversibilizing formal semantics The meaning of the elementary statements of L can be extended such that the execution of a program becomes reversible, *e.g.*, by tracing each of the non-injective statements of L. This can be achieved by manually *reversibilizing the formal semantics* of L. Examples of this metalevel approach include the reversibilized semantics of a logic language [109] and term rewriting [93].

We mention on passing that the semantics of an irreversible language is formalized in an irreversible metalanguage (*e.g.*, operational semantics), while the metalanguage used to formalize the semantics of a reversible language can be reversible, which has the advantage that properties relevant for reversible languages can be guaranteed by the *reversible metalanguage* [52].

Reversibilizing interpreters Given an interpreter *int* for L, the reversibilization of the semantics into a reversible semantics L_{rev} of L can be carried out programmatically by applying a reversibilizer to *int*.

For any irreversible language L, an interpreter *int* for L can be implemented in a Turing-complete language T such that for all L-programs p and data x, the well-known functional equivalence holds:

$$[\![int]\!]_T(p, x) = [\![p]\!]_L\, x. \tag{18}$$

Reversibilizing *int* then gives a *reversible interpretation* of L-programs in R,

$$[\![\overline{int}^{lan}]\!]_R(p, x) = ([\![p]\!]_L\, x, trace), \tag{19}$$

$$[\![\overline{int}^{ben}]\!]_R(p, x) = ([\![p]\!]_L\, x, (p, x)), \tag{20}$$

where

$$[\![lan]\!]\ int = \overline{int}^{lan}, \tag{21}$$

$$[\![ben]\!]\ int = \overline{int}^{ben}. \tag{22}$$

The T-to-R reversibilizers *lan* and *ben* use the Landauer embedding and the Bennett method, respectively. As can be seen from the two reversibilized interpreters, we have $[\![\overline{int}^{lan}]\!]_R(p, x) \neq [\![p]\!]_L\, x \neq [\![\overline{int}^{ben}]\!]_R(p, x)$.

If we regard a reversibilized interpreter, such as $\overline{int}^{lan}$, as an implementation of a *reversible semantics* L_{rev} of L, then we have two different meanings for the same program text p, the *standard semantics*, $[\![p]\!]_L$, and a *non-standard semantics*, $[\![p]\!]_{L_{rev}}$, with the properties

$$[\![p]\!]_L \neq [\![p]\!]_{L_{rev}},\ [\![p]\!]_L \trianglelefteq [\![p]\!]_{L_{rev}},\ \text{and}\ [\![p]\!]_{L_{rev}} : \text{injective}. \tag{23}$$

Note that p is not modified in (23), only the semantics of its implementation language L. This approach "sweeps the garbage under the carpet" by using a reversibilized interpreter that takes full control of producing the additional data to ensure reversibility. The first approach to reversibilization in (13) has a "translative" character because p is transformed, whereas the second approach in (19), (20) has an "interpretive" character because the interpreter running p is modified.

Early examples for reversibilized interpreters are the Ψ-Lisp interpreter [16] that uses a hidden conditional stack and the SEMCD abstract machine [66] that uses two additional runtime structures for tracing irreversible operations. Program $\overline{int}^{lan}$ is related to reverse debuggers which are instrumented interpreters that can be run forward and backward. The GNU Project Debugger (GDB) [54], the C/C++ debugger (RR) [94], the OCaml debugger (`ocamldebug`), and the Erlang debugger (CauDEr) [74] support reversible debugging, which is an application of reversible interpretation of irreversible languages. They can control program execution unconventionally, for example, stepping backward through the program and changing the direction of computation. When an abnormal state is reached, the state immediately before can be analyzed by not performing the entire computation again.

[13] The *dereversibilization* of reversible into irreversible programs, the converse of reversibilization, is important for an efficient implementation on conventional hardware, *e.g.*, [88].

1. **Specification:** Injectivization of non-injective function.

$$\begin{array}{lcl} \text{Fibonacci} & & \text{Fibonacci pair (injective)} \\ \mathit{fib}\ n & \overset{\mathit{injectivize}}{\longrightarrow} & \mathit{fibpair}\ n = (\mathit{fib}\ n, \mathit{fib}(n+1)) \end{array}$$

2. **Implementation** in a reversible programming language. Algorithm design, programming methods, optimizations, compilation.

Fig. 6. Fibonacci example: Two main steps for developing a reversible program.

Reversibilizing on two levels As usual with interpretive and translative approaches in programming, both can be combined. For example, some statements might be easier to instrument at the level of the source programs, while others might be easier to instrument once and for all at the level of the formal semantics. Then, we expect for the *partially reversibilized semantics* L' of L and for all *partially reversibilized programs* p' of p that the following holds for their combination:

$$[\![p]\!]_L \trianglelefteq [\![p']\!]_{L'} \text{ and } [\![p']\!]_{L'} : \text{injective}. \tag{24}$$

Whether the reversibilization is fully carried out at the source-program level (13) or at the semantic metalevel (19), (20), or by a combination of both (24), depends on practical considerations and the application scenario. If a reversible target language R is available, it may be easier to reversibilize the existing L-programs and to perform program-specific optimizations at the source-program level instead of reversibilizing the formal semantics of L or developing a new L-interpreter for reversibilization. A reversibilized source program may be more efficient than interpreting irreversible source programs by a reversibilized interpreter. These trade-offs are familiar from conventional translators and interpreters. We mention on passing that program specialization (*e.g.*, [61]) is well suited for reducing the overhead of interpretive reversibilization. Recent examples of two-level source-semantic reversibilizations (24) done manually are [30,59].

5. Cornerstones of reversible programming languages

Reversible programs operate on the same data structures as irreversible programs (*e.g.*, stacks, arrays, lists), but they cannot overwrite data and can only perform *reversible updates* of data [13] (Sect. 4.1). This makes reversible languages fundamentally different from their irreversible counterparts, such as C or Java, which can perform destructive updates by overwriting data.

Control-flow operators, such as conditionals and loops, must be made backward deterministic in order for programs to become reversible. For this, each join point in the control flow is equipped with a predicate that asserts where the control came from. Reversible conditionals, iterations, and recursions are all available for programming.

As a *bare minimum*, a reversible language consists of three elements [47]:

1. a reversible assignment,
2. a reversible while-loop, and
3. dynamic data structures.

As a minimum, constructor-based data structures are built from a single binary constructor and a single symbol. This reversible core language, `R-CORE`, is computationally as powerful as reversible Turing machines [47].

Because of their backward determinism, reversible languages can provide features that invoke the *inverse computation* of a program unit (*e.g.*, by uncalling a procedure, while the standard computation of the same procedure is invoked as usual by a call.) The possibility to call and uncall the same procedure enables a new form of reliability through *code sharing* [114]. Instead of writing two separate procedures that are inverse to each other, *e.g.*, a lossless encoder and decoder [48], it is sufficient to implement, test, and verify one of them and to obtain the effect of the other by an uncall. One of the earliest works on reversible subroutines was [100]. Reversible programming demands certain sacrifices because data cannot be overwritten and join points in the control flow require explicit assertions, a technique known from program inversion [55].

5.1. Fibonacci pairs: a reversible program

We present the imperative reversible language Janus by an implementation of the familiar Fibonacci function. This example allows us to present features representative of many reversible languages, including store updates, conditionals, and access to a program's inverse semantics.

A reversible program in two steps We use the Fibonacci function to illustrate the preferred strategy of implementing a non-injective function in a reversible language (Fig. 6). In Sect. 4, we argued that it is preferable to first injectivize the function and then to implement it directly in a reversible language, such as Janus, instead of implementing the function first in an irreversible language, such as C or Java, and then reversibilizing the irreversible implementation. For every injective function there exists a semantically faithful and usually more efficient direct implementation in a reversible language.

```
int n x1 x2     // variable declarations

procedure fibpair
  if n=0 then x1 += 1
              x2 += 1
         else n -= 1
              call fibpair
              x1 += x2
              x1 <=> x2
  fi x1=x2
```

Fig. 7. Fibonacci-pair function [39] written in Janus (from [114]).

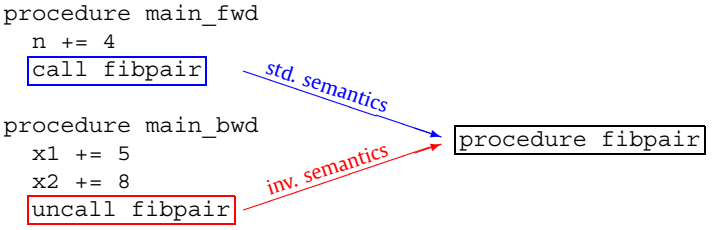


Fig. 8. Calling and uncalling of the Fibonacci-pair procedure in Fig. 7.

The first step is to injectivize the function. We know that an injectivization is always possible and entails a semantic modification. In general, a function-specific injectivization is preferable over an input-preserving one, as discussed in Sect. 4.1. Instead of computing for a given n, a single Fibonacci number *fib* $n, n \geq 0$, we define a function *fibpair* n that computes two neighboring Fibonacci numbers (*fib* n, *fib* $(n+1)$). The new function is injective because for each n the Fibonacci pair is unique. Computing a pair of numbers has the potential advantage that we can compute them efficiently if we need a sequence of Fibonacci numbers. A function-specific injectivization is often closer to the intention of the original specification than to indiscriminately preserve the input in the output. Obviously, there is no general rule as to which injectivization is the best, and the choice also depends on how the function is going to be used in a particular application.

The second step, the implementation of the Fibonacci-pair function in a reversible language, concerns familiar considerations, such as algorithm design, programming techniques, optimizations, and efficiency, and poses new challenges to a reversible programming methodology, such as finding predicates for backward deterministic control flow.

We will implement the Janus program from the recursively defined function *fibpair*. We can obtain a recursive definition by dividing *fibpair* n into the cases $n = 0$ and $n \geq 1$, unfolding *fibpair* and *fib*, and then folding *fibpair*:

$$\mathit{fibpair}\ n = \begin{cases} (1,1) & (n=0) \\ \text{let } (x_1, x_2) = \mathit{fibpair}(n-1) \text{ in } (x_2, x_1 + x_2) & (n \geq 1) \end{cases} \tag{25}$$

By mathematical induction, we can confirm that the output value (x_1, x_2) is returned by the base case if $x_1 = x_2 = 1$, and by the recursive case if $x_1 \neq x_2$. Based on this observation, we have an assertion $x_1 = x_2$ to distinguish the values returned by the base and recursive cases.

Deletion considered harmful The implementation of the injectivized function in *Janus* [78,114] is shown in Fig. 7. Janus is a reversible imperative language with a global store, integers and arrays as data structures, parameterless procedures, and two structured control-flow operators. Here, we use the syntax of the first formalization [114]. An extended version of Janus features recursive procedures with reference parameters, local variables, and stacks as dynamic data structures. This makes the extended version r-Turing complete [115].

At first, the Janus implementation looks like a conventional program with a conditional, assignments, and a recursive procedure call in the body of procedure declaration `fibpair`.

A second look reveals that the conditional `if` has an *exit assertion* marked by `fi`. This assertion at the exit of the conditional, the predicate `x1=x2`, is checked at run time, just like the predicate `n=0` in the entry of the conditional. The predicate must evaluate to true when control *comes from* the then-branch and to false when control *comes from* the else-branch. If it does not evaluate as expected, the conditional is undefined and the program aborts.

All assignments in the body of the procedure `fibpair` are *non-destructive*: they only increment or decrement a value by a C-like compound assignment operator (`+=`, `-=`).[14] The last statement `x1 <=> x2` of the else-branch swaps the values

[14] A reversible assignment, x *op*= e, is a shorthand notation for $x = x$ *op* e, where *op* is an operator that is injective in its first argument, such as `+`, `-`, or bitwise exclusive-or `^`, and x is a variable that does not occur in expression e to ensure reversibility. It reversibly updates x leaving all variables in e unchanged. A Toffoli gate can be written as `x1 ^= x2 && x3`, where `&&` is the logical operator $\wedge$ on integers `0` and `1` (Sect. 4.1).

Forward run: `call fibpair`	n	x1	x2
`if n=0`	2	0	0
`n -= 1`	1	0	0
`  if n=0`	1	0	0
`  n -= 1`	0	0	0
`    if n=0`	0	0	0
`    x1 += 1`	0	1	0
`    x2 += 1`	0	1	1
`    fi x1=x2`	0	1	1
`  x1 += x2`	0	2	1
`  x1 <=> x2`	0	1	2
`  fi x1=x2`	0	1	2
`x1 += x2`	0	3	2
`x1 <=> x2`	0	2	3
`fi x1=x2`	0	2	3

Backward run: `uncall fibpair`	n	x1	x2
`if x1=x2`	0	2	3
`x1 <=> x2`	0	3	2
`x1 -= x2`	0	1	2
`  if x1=x2`	0	1	2
`  x1 <=> x2`	0	2	1
`  x1 -= x2`	0	1	1
`    if x1=x2`	0	1	1
`    x2 -= 1`	0	1	0
`    x1 -= 1`	0	0	0
`    fi n=0`	0	0	0
`  n += 1`	1	0	0
`  fi n=0`	1	0	0
`n += 1`	2	0	0
`fi n=0`	2	0	0

Fig. 9. Forward and backward trace of procedure `fibpair` (Fig. 7) starting from index $n = 2$ and ending with Fibonacci pair $x1 = 2$, $x2 = 3$, and *vice versa*. Procedure calls and returns are visualized by statement indentation; the store is shown after statement execution.

of `x1` and `x2`. Thus, no values are overwritten; variables are only updated reversibly. For simplicity, the procedure has no parameters and modifies three global variables, where `n` is the index of the Fibonacci pair to be computed and `x1` and `x2` are the final numbers of the `n`th Fibonacci pair. Initially, all variables in the global store are zero.

Inverse computation on the fly Figure 8 shows how our procedure is invoked by `call fibpair` after initializing variable `n` to `4` by incrementing the zero-cleared `n` in the procedure `main_fwd`. When `fibpair` returns, `n` is zero, and `x1` and `x2` hold the `4`th Fibonacci pair, that is, numbers `5` and `8`, respectively.[15]

Figure 9 (Left) shows the trace of the forward run of procedure `fibpair` in Fig. 7 with the initial state `n=2`, `x1=0`, `x1=0` and the final state `n=0`, `x1=2`, `x1=3`. It zero-clears the index `n` and computes the 2nd Fibonacci pair on `x1` and `x2`. Figure 9 (Right) shows the trace of the backward run of procedure `fibpair`. Each step of the backward run reverts each state, and hence, both computations have the same pair of input and output.

Because reversible programming languages are forward and backward deterministic, the same procedure can run forward and backward. Backward (inverse) computation can be invoked at runtime by an *uncall*, while forward (standard) computation is invoked by a *call*.

Inverse computation of procedure `fibpair` is invoked by `uncall fibpair`. Given a Fibonacci pair (`x1`, `x2`), this uncall determines the pair's index (`n`). For instance, after setting the variable pair (`x1`, `x2`) to (`5`, `8`) in procedure `main_bwd`, the uncall resets the pair to zero and `n` to `4` because this is the 4th Fibonacci pair. In short, the uncall computes the inverse function $\mathit{fibpair}^{-1}(x_1, x_2) = n$ instead of function $\mathit{fibpair}(n) = (x_1, x_2)$. Given a pair for which function $\mathit{fibpair}^{-1}$ is undefined, say (`7`, `2`), the uncalled procedure is undefined and aborts at run time.

It is noteworthy that *recursion is reversible*: the procedure `fibpair` is run forward and backward. If a procedure has local variables then they are zero when it is called and must be zero when it returns. Also observe in Fig. 9 that both runs take the same number of steps with the same pair of input and output, respectively. Thus, call and uncall are equally efficient.

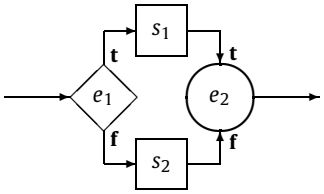
Code sharing, reliability, and trade-offs Our example demonstrates how the same procedure text can be given two different meanings in a reversible language. Provided that procedure `fibpair` is a correct implementation of function *fibpair*, `uncall fibpair` correctly computes the inverse function $\mathit{fibpair}^{-1}$. There is no need for backtracking because of the backward determinism of the language. This allows new forms of code sharing and can improve the reliability of programs. Instead of writing, testing, and verifying two procedures separately, it is sufficient to do this for one of them and obtain the other direction by invoking that procedure's inverse semantics. The correctness of the inverse direction follows from the semantics of the reversible language. This means that even if the procedure has a bug and we obtain an unexpected output, running it backward reconstructs the original input.

The merit of code sharing by call and uncall is the guaranteed consistency in both directions. For example, a new version of an encoder simultaneously updates the decoder that uncalls that encoder. Of course, the responsibility for the correctness of the encoder is still with the programmer.

Reversible programming also demands certain sacrifices. Values cannot be discarded, and join points in the control flow require explicit assertions. The usual irreversible programming methods, such as overwriting the value of a variable are not available. These requirements, which are enforced by the syntax and semantics of the language, guarantee that the execution of a reversible program is deterministic in both directions. The cost-benefit trade-off of reversible *vs.* irreversible programming is ultimately context-dependent.

[15] Janus uses *modulo arithmetic*, which wraps around at the smallest and largest representable integers, *e.g.*, for 32-bit integers at $(2^{31} - 1) + 1 = -2^{31}$.

Reversible flowchart diagram:

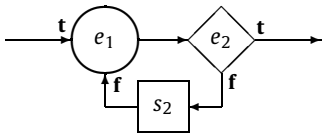


Textual representation:

```
if e1 then s1 else s2 fi e2
```

Fig. 10. Structured control flow: Reversible conditional (Janus) [114].

Reversible flowchart diagram:



Textual representation:

```
from e1 loop s2 until e2
```

Fig. 11. Structured control flow: Reversible while loop (R-CORE) [47].

5.2. How to control reversible programs?

Reversible flowcharts are a useful model of imperative reversible languages because they naturally represent the programs' control flow and structure.

Reversible flowchart languages Reversible flowcharts have the great advantage that they accommodate rather diverse programming language concepts in the same framework. They can represent high-level aspects of block-structured general-purpose languages, as presented above, and low-level aspects of compiler-generated machine code, such as jumps and assembly-level instructions. By adding assertions to join points and restricting atomic steps to injective functions, classic flowcharts become *reversible flowcharts* [119]. These changes ensure that the transition function of a reversible flowchart is injective, and that flowchart computation is forward and backward deterministic. The same data domains, such as integers, arrays, stacks, and lists, are available to reversible and classic flowcharts. The data domains are agnostic to the type of flowchart; it is the operations on the data domains that must be injective.

Everything that can be computed by an unstructured reversible flowchart can be computed by a structured reversible flowchart using only three specific reversible constructs (sequence, selection, loop), *i.e.*, the *Structured Reversible Program Theorem* holds in reversible computing [116,119]. Every unstructured reversible flowchart can even be simulated by a structured reversible flowchart with exactly one reversible loop.

Reversible conditionals Figure 10 shows the flowchart diagram of a *reversible conditional*. The reversible flowchart looks similar to the classic flowchart of a conditional with a *test* (rhombus) and two branches with *steps* (rectangles), but has an additional *assertion* (circle) at the join point of the control flow. In a reversible flowchart, every join point carries an assertion. Another difference to a classic flowchart is the limitation of all steps to locally invertible operations such as reversible assignments (`+=`, `-=`) and value swaps (`<=>`). These properties make the flowchart reversible.

As usual, the test dispatches control along with one of the two outgoing edges. If predicate e_1 is *true*, the true-edge (labeled **t**) is selected, and the step s_1 is performed (then branch); otherwise, the false-edge (labeled **f**) is selected, and the step s_2 is performed (else branch). The assertion is a new flowchart operator. The predicate e_2 must be true when the control flow reaches the join point along the true-edge and false when the control flow reaches the join point along the false-edge; otherwise, the assertion is undefined. The test and the assertion are evaluated when dispatching and joining the control during the execution of the conditional, respectively. The textual representation of the reversible flowchart is shown below the diagram and is the conditional used in the Janus program in Fig. 7.

Reversible loops A *reversible while loop* is depicted in Fig. 11. Here, the assertion e_1 is located at the entry of the loop, which is the join point of two incoming edges, and the test e_2 with two outgoing edges at the loop exit.

An iteration of the reversible while loop has three phases:

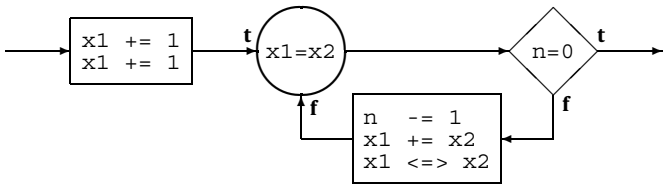
1. Enter: When entering the loop, e_1 must be true (**t**).
2. Repeat: The body of the loop s_2 is repeated if e_2 is false (**f**), and after each execution of the body e_1 must be false (**f**).
3. Exit: The loop is exited when e_2 is true (**t**). If e_2 does not become true, the loop does not terminate.

The body s_2 of the loop is skipped when both e_1 and e_2 are initially true. To understand the role of e_1 and e_2 better, let us look at the sequence of their truth values during the execution of a loop:

$$\begin{array}{rcccc} & & \text{Enter} & \text{Repeat} & \text{Exit} \\ \text{assertion } e_1 & = & \mathbf{t} & \mathbf{f} \cdots \mathbf{f} & \mathbf{f} \\ \text{test } e_2 & = & \mathbf{f} & \mathbf{f} \cdots \mathbf{f} & \mathbf{t} \end{array} \tag{26}$$

The sequence of truth values can be read from left to right (forward execution) and *vice versa* (backward execution). When the direction of execution is changed, the assertion and the test switch roles as the loop's entry and exit "doors."

Reversible flowchart diagram:



Textual representation:

```
x1 += 1
x2 += 1
from  x1=x2
loop  n  -= 1
      x1 += x2
      x1 <=> x2
until n=0
```

Fig. 12. Fibonacci-pair function written using a reversible while loop.

The assertion becomes the test, and the test becomes the assertion. This symmetry is the basis for the reversibility of the control flow of loops, and we will use it later when we invert loops (Sect. 6.2).

Example 4 *(Reversible while loop).* To illustrate *reversible iteration*, we use the Fibonacci-pair function, which we now implement using a reversible while loop (Fig. 12) instead of a recursive procedure (Fig. 7).

Variable `n` is the index of the Fibonacci pair to be computed, and variables `x1` and `x2` are initialized to `1`, that is, the values of the initial Fibonacci pair. Thus, initially, assertion `x1=x2` is true. In each iteration, `n` is decremented, and the next pair is computed by swapping the sum of the current pair in `x1` with the current value in `x2`. After the first iteration, `x1=x2` will become false. The computation of the next pair is repeated until `n=0`. Thus, in the exit state, `x1` and `x2` contain the values of the `n`th pair and `n=0`; *cf.* (26).

This reversible loop acts like a conventional, irreversible loop except for the additional assertion `x1=x2` and the exclusive use of reversible statements. Again, a procedure having this loop as its body can be called and uncalled.

Iteration variants The *general reversible loop*, also called the reversible half loop, available in Janus and other reversible languages, has an additional step s_1 that is executed between the assertion e_1 and the test e_2. In all other respects, it behaves like the reversible while loop that we have seen above. The textual representation of the general loop:

$$\texttt{from}\ e_1\ \texttt{do}\ s_1\ \texttt{loop}\ s_2\ \texttt{until}\ e_2$$

The main advantage is that the general loop can act as a reversible *while-do loop* when s_1 is a `skip`, which is a statement that performs the identity function, and as a *repeat-until loop* when s_2 is `skip`. As syntactic sugar, `DO skip` and `LOOP skip` can be omitted.

Thus, we have now three forms of reversible iteration at our disposal:

1. a general loop,
2. a while loop, and
3. a repeat loop.

Techniques and analyses This completes the presentation of features representative of many reversible languages: store updates, structured control-flow operators, and access to a program's inverse semantics. We are now, at least in principle, in a position to write reversible programs.

The next step is programming techniques and programs analyses (*e.g.*, [25,50]). Programming techniques include zero-cleared copying, zero-clearing by constants, code sharing by call and uncall, the use of the Bennett method, and reversible data structures [22,48,114]. Translations from high-level reversible languages, like Janus, to low-level reversible assembly languages, must be strictly semantics-preserving (clean) and preserve complexities [15,116].

An unconventional program analysis concerns termination. Termination of a reversible loop is guaranteed if the loop's memory usage is bounded [119, Prop. 1]. During an iteration, the state is updated to a new state that differs from any of the previous states in the loop. Hence, a loop with bounded memory usage eventually terminates. For example, the while loop in Fig. 12 is guaranteed to terminate because it uses three variables of fixed size.

6. Metacomputation

We begin by remarking that the results discussed in this section show the power of a small set of fundamental meta-operations, including the inversion and specialization of programs [42]. To be familiar with standard and non-standard semantics as well as the principles of staging and unstaging programs is quite useful, in particular in the field of reversible computing. In many ways, reversible computing is a sweet spot for studying metacomputation by inverse interpretation and program inversion. Some of the unconventional methods discussed in this section, which seem difficult to perform in irreversible computing, are easy and useful in reversible computing.

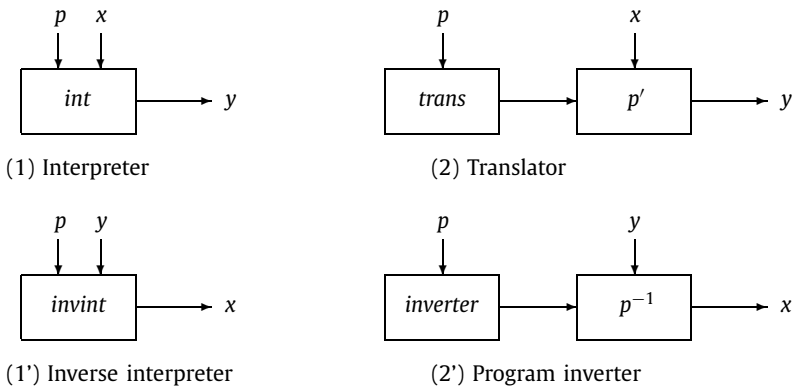


Fig. 13. Standard computation $[\![p]\!]\, x = y$ by an interpreter (1) and a translator (2). Solving the inverse problem of $[\![p]\!]\, x = y$ by an inverse interpreter (1') and a program inverter (2'). (Figure adapted from [2].)

6.1. Inverse interpreters and program inverters

There are only two ways to implement the semantics of a language, namely by an interpreter or a translator. The corresponding two metaprograms also exist for implementing the *inverse semantics* of a language, namely by an *inverse interpreter* or a *program inverter* [4]. Like a translator, a program inverter takes a program as input but, instead of performing an equivalence transformation of the program, transforms the program into its inverse. An inverse interpreter performs the inverse computation of a program.

In the case of implementing the semantics of reversible languages, it was shown that inverse interpreters and program inverters are efficient and straightforward [114]. It should be stressed that both metaprograms exist for *any* language, including irreversible ones [4]. In all probability, the first inverse interpreter was indeed for irreversible Turing machines [81] and the first automatic program inverter for Lisp [67] (*cf.* [41]). In practice, various useful combinations exist, such as program inverters built into translators and inverse interpreters running interpreters. The former is an application where programs written in a reversible language, such as Janus with an uncall feature, are translated to an irreversible target language. The latter is a semantics modifier that performs inverse interpretation in another language via an interpreter [4].

This section focuses on the tools particularly important to reversible programming. Our primary focus is on their functionality, and we take the liberty to disregard the involved source, target, and implementation languages unless stated otherwise. A detailed exposition of the concepts presented here can be found in the publications [1,3,4].

Here, we present the metaprograms in their pure form. The language in which they are implemented is secondary to our discussion, but if they are to be implemented in a reversible language, then the exact same considerations about injectivization and reversibilization of Sect. 4 apply.[16]

Relating inverse interpreters and program inverters We begin with familiar principles to discuss analogies to inverse interpreters and program inverters.

An interpreter is a program written in one language that runs programs in another language. It can also be a self-interpreter written in the language that it interprets, an example being a universal Turing machine. A translator is a program that translates programs from one language to another. It can also be a source-to-source translator, an example being a program optimizer. An interpreter and a translator implement the *standard semantics* of the language in which their source programs are written. Both metaprograms are illustrated in Fig. 13 (1, 2), where a program p with input x is performed in one and two stages by an interpreter *int* and a translator *trans*, respectively.

More formally, a program *int* and a program *trans* are an *interpreter* and a *translator*, respectively, iff for all source programs p and for all values x, y:

$$[\![int]\!]\,(p, x) = y \iff [\![p]\!]\, x = y, \tag{27}$$

$$p' = [\![trans]\!]\, p \text{ and } [\![p']\!]\, x = y \iff [\![p]\!]\, x = y. \tag{28}$$

Using equations (27), (28), we obtain for all programs p the well-known functional equivalence between interpretation and translation:

$$\underbrace{[\![int]\!]\,(p, x)}_{\text{one stage}} = \underbrace{[\![\,[\![trans]\!]\, p]\!]\, x}_{\text{two stages}}. \tag{29}$$

[16] An example is the reversible self-interpreters [47,114], which preserve the source program as part of the output; see also [47, Sect. 2]. Other examples include the self-interpreters for reversible Turing machines [8,11] and their subclasses [91,92].

For simplicity, let us now assume that p is an *injective program*, *i.e.*, a program that implements an injective function.[17] Computation of y by applying p to x is described by $[\![p]\!]\, x = y$. The determination, for a given p and y, of an x such that $[\![p]\!]\, x = y$ is the *inverse computation*.[18] A program that performs the inverse computation of programs is an inverse interpreter. A program that transforms programs into their inverse programs is a program inverter.

More formally, a program *invint* and a program *inverter* are an *inverse interpreter* and a *program inverter* for *injective programs*, respectively, iff for all injective source programs p and for all values x, y:

$$[\![invint]\!]\,(p, y) = x \iff [\![p]\!]\, x = y, \tag{30}$$

$$p^{-1} = [\![inverter]\!]\, p \text{ and } [\![p^{-1}]\!]\, y = x \iff [\![p]\!]\, x = y. \tag{31}$$

Using equations (30), (31), we obtain for all programs p the functional equivalence between inverse interpretation and program inversion:

$$\underbrace{[\![invint]\!]\,(p, y)}_{\text{one stage}} = \underbrace{[\![\,[\![inverter]\!]\, p]\!]\, y}_{\text{two stages}}. \tag{32}$$

Equation (32) for inverse computation is analogous to equation (29) for standard computation. As for interpreters and translators, inverse computation can be performed in one or two stages by an inverse interpreter and a program inverter, respectively. The application of *invint* and *inverter* for solving the inversion problem of $[\![p]\!]\, x = y$ is illustrated in Fig. 13 (1', 2').

The reason for using a program inverter instead of an inverse interpreter is similar to using a translator instead of an interpreter, namely that an inverse program can solve the inversion problem much more efficiently than an inverse interpreter. Similar to an interpreter, an inverse interpreter adds a layer of non-standard interpretation to the run of a program.

Existence of the metaprograms Inverse interpreters and program inverters are less familiar metaprograms, and thus it is important to know whether they can be constructed for any Turing-complete language.

If we disregard efficiency and focus on their existence, then a *trivial inverse interpreter* can be easily implemented using a *generate-and-test* method that runs—for a finite but ever increasing time budget—the program p for all possible inputs in search for those enumerated inputs $x_1, x_2, \ldots$ that yield the given output y. Clearly, this inverse-interpretation method, which was first discussed by McCarthy for irreversible Turing machines [81]—remarkably before his design of the list-processing language Lisp [82]—is much too inefficient for practical use, but suffices to show constructively the existence of an inverse interpreter for any Turing-complete language. From this follows the existence of program inverters for any Turing-complete language [4]. It requires only a language that can express a timed self-interpreter and enumerate and compare input and output data. This is the case for reversible Turing machines. Even better, the inverse interpretation can be performed significantly more efficiently for reversible Turing machines and be built into reversible languages and even into reversible hardware.

Historically, and this is an interesting fact, methods for inverse interpretation and program inversion were considered independently of each other in the 50s and 70s [26,55,81]. Eventually they solve the same inverse computation problems, so for a full understanding of this area, it is fruitful to investigate them together, in particular in the field of reversible computing.

Staging inverse interpreters Inverse computation can be performed in one or two stages using an inverse interpreter or a program inverter, respectively, as shown in (32). From standard computation it is known that an interpreter can be staged into a translator by a self-applicable program specializer according to the *Futamura projections* [37]. Similarly, an inverse interpreter can be staged into a program inverter by the same transformation [4].

The staging relation between the metaprograms (29), (32) can be expressed formally by a program-generator generator. A program *cog* is a *program-generator generator*[19] iff for all programs p and for all values x_1, x_2, and y:

$$[\![cog]\!]\, p = g \text{ and } [\![[\![g]\!]\, x_1]\!]\, x_2 = y \iff [\![p]\!]\,(x_1, x_2) = y. \tag{33}$$

From this equation, we see that *cog* transforms p into a program generator g that, given x_1, generates a program which, given x_2, computes the same result y as p given x_1 and x_2 at the same time. The program g is also called a *generating extension* of p [29].

Without going into further technical details, they can be found in the partial evaluation literature (*e.g.*, [61]), we note that *cog* is a program that stages a program with two inputs[20] into a program with *two computation stages* and that the

[17] A program p is *injective* if its meaning $[\![p]\!]$ is injective, as defined in Sect. 4, Notation.

[18] In general, when p is not injective, the solution to the inversion problem is not uniquely defined, and an existential solution (one of the inputs) or a universal solution (all of the inputs) can be sought. Algorithms solutions for solving inversion problems are the generate-and-test method [81], URA [2,4,5], and logic programming systems in general [68].

[19] Often called a *compiler generator* for historical reasons; thus, the abbreviation *cog*.

[20] Each input can also be a tuple of inputs: $x_1 = (x_{1_1}, \ldots x_{1_n})$, $x_2 = (x_{2_1}, \ldots x_{2_m})$.

classic application of *cog* is the staging of an interpreter into a translator. Likewise, *cog* can stage an inverse interpreter into a program inverter.[21] That is, the *generating extension of an inverse interpreter* is a program inverter. The following two applications of *cog* turn *int* and *invint* into their respective generating extensions *trans* and *inverter*:

$$[\![cog]\!]\ int = trans, \tag{34}$$

$$[\![cog]\!]\ invint = inverter. \tag{35}$$

The formal relationship we have established here shows that *invint* and *inverter* are two sides of the same coin. From this we conclude that it is productive to study them together rather than in isolation. It should not come as a surprise that reversible computing makes good use of both in various ways regardless of whether reversible hardware or software is concerned. As a reversible-language example in Janus, we have seen that inverse interpretation of a procedure can be invoked at runtime by an `uncall` (Fig. 8) and that a procedure can be inverted textually before it is run (later, Fig. 14).

Advanced metacomputation topics Many reversible languages provide a built-in feature to invoke inverse computation at runtime. It is, therefore, not necessary to write an inverse interpreter for these languages as built-in inverse computation by an `uncall` is as efficient as standard computation by a `call`.

There are reversible languages that do not provide built-in access to inverse computation at runtime, such as reversible Turing machines. In this case, one can always obtain an inverse interpreter by *program inversion of a self-interpreter* for the respective language. An example would be the program inversion of a universal reversible Turing machine, *e.g.*, the ones given in [8,89]. See also the discussion of standard *vs.* reversible interpreters [46].

For the sake of completeness, we point out that the staging relation (35) can also be established in the other direction, namely by *degeneration of a program inverter into an inverse interpreter* by program composition [43]. While experimental results for staging inverse interpreters are known [51], the unstaging of inverters has not yet been attempted, although the program inversion of program inverters was [65].

Finally, inverse computation has an intriguing property that parallels the construction of the well-known *towers of interpreters* [82]. We know, and have performed such experiments [3,4,44], that inverse computation of programs written in another language can be performed via an ordinary interpreter for that language, hence without writing a new inverse interpreter. This *non-standard interpreter tower* [1] has an inverse interpreter at the bottom and an interpreter on top and can be collapsed by program specialization, at least in principle. The specialization of a tower of interpreters is an optimization that can substantially reduce the order-of-magnitude time cost of multiple levels of interpretation [61, p.141]. *Inverse computation via standard interpreters* is possible because inverse semantics fall into the class of *robust non-standard semantics*; for theoretical results and experiments, see, for example, [1,3,4,51].

6.2. A program inverter for a reversible language

Program inversion performs a textual transformation of a program into its inverse program. In general, it is not easy to obtain an efficient inverse program (*e.g.*, [26,28,40]) but program inversion is straightforward in reversible languages. This is one of their most important language properties.

Example 5 *(Program inversion).* To illustrate program inversion in a reversible language, Fig. 14 shows the Fibonacci-pair procedure and its inverse. Each construct of the Fibonacci-pair procedure—conditional, call, assignments, and their composition—is inverted locally by a recursive descent over the structure of the procedure text.

The inversion rules of a *syntax-directed program inverter* are not complicated. Figure 15 shows the rules of the program inverter $\mathcal{I}$ for Janus [114].

The rules are applied locally by a recursive descent over the structure of a program. Each procedure $proc_i$ of the program is inverted for itself; the program's variable declarations d^* remain unchanged. A procedure named *id* is renamed to id^{-1}, and its body *s* is inverted by $\mathcal{I}[\![s]\!]$. Note that id^{-1} is a shorthand notation for a new procedure name generated from *id*. The name itself is not important as long as it is a valid identifier and the same name is generated when inverting a procedure invocation of *id*.

The control-flow operators `if-fi` and `from-until` are inverted by swapping the roles of the predicates, e_1 and e_2, and inverting the body statements by $\mathcal{I}[\![s_1]\!]$ and $\mathcal{I}[\![s_2]\!]$. A sequence of statements $s_1\ s_2$ is reversed, and each statement is inverted individually. The inverse of a `call` *id* is a call to its inverse id^{-1}; likewise for an `uncall`. What remains are the compound assignment operators `+=` and `-=`, which are inverse to each other. The bitwise exclusive-or assignment operator `^=` is self-inverse as are the swap statement `<=>` and the identity statement `skip`. This is the complete program inverter.

[21] An application of the inversion projections (*e.g.*, [3]).

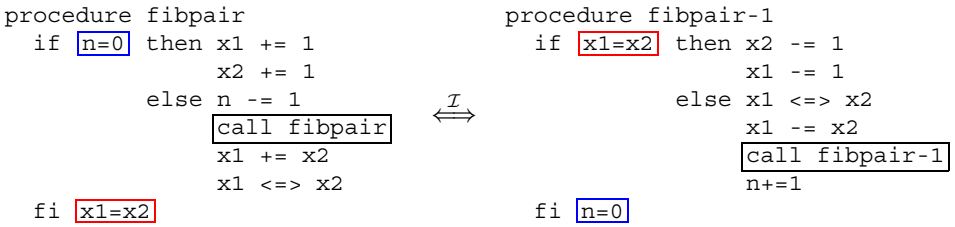


Fig. 14. Program inversion by $\mathcal{I}$: The Fibonacci-pair procedure and its inverse [114].

$$
\begin{array}{rll}
\textit{Program:} & \mathcal{I}[\![d^*\ \textit{proc}_1 \cdots \textit{proc}_n]\!] & = d^*\ \mathcal{I}[\![\textit{proc}_1]\!] \cdots \mathcal{I}[\![\textit{proc}_n]\!] \\
\textit{Procedure:} & \mathcal{I}[\![\texttt{procedure}\ \textit{id}\ s]\!] & = \texttt{procedure}\ \textit{id}^{-1}\ \mathcal{I}[\![s]\!] \\
\textit{Control-flow operators:} & \mathcal{I}\left[\!\!\left[\begin{array}{l}\texttt{if}\ e_1\ \texttt{then}\ s_1 \\ \texttt{else}\ s_2\ \texttt{fi}\ e_2\end{array}\right]\!\!\right] & = \begin{array}{l}\texttt{if}\ e_2\ \texttt{then}\ \mathcal{I}[\![s_1]\!] \\ \texttt{else}\ \mathcal{I}[\![s_2]\!]\ \texttt{fi}\ e_1\end{array} \\
 & \mathcal{I}\left[\!\!\left[\begin{array}{l}\texttt{from}\ e_1\ \texttt{do}\ s_1 \\ \texttt{loop}\ s_2\ \texttt{until}\ e_2\end{array}\right]\!\!\right] & = \begin{array}{l}\texttt{from}\ e_2\ \texttt{do}\ \mathcal{I}[\![s_1]\!] \\ \texttt{loop}\ \mathcal{I}[\![s_2]\!]\ \texttt{until}\ e_1\end{array} \\
\textit{Sequence:} & \mathcal{I}[\![s_1\ s_2]\!] & = \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!] \\
\textit{Procedure invocation:} & \mathcal{I}[\![\texttt{call}\ \textit{id}]\!] & = \texttt{call}\ \textit{id}^{-1} \\
 & \mathcal{I}[\![\texttt{uncall}\ \textit{id}]\!] & = \texttt{uncall}\ \textit{id}^{-1} \\
\textit{Atomic statements:} & \mathcal{I}[\![x\ \texttt{+=}\ e]\!] & = x\ \texttt{-=}\ e \\
 & \mathcal{I}[\![x\ \texttt{-=}\ e]\!] & = x\ \texttt{+=}\ e \\
 & \mathcal{I}[\![x\ \texttt{\^{}=}\ e]\!] & = x\ \texttt{\^{}=}\ e \\
 & \mathcal{I}[\![x_1\ \texttt{<=>}\ x_2]\!] & = x_1\ \texttt{<=>}\ x_2 \\
 & \mathcal{I}[\![\texttt{skip}]\!] & = \texttt{skip}
\end{array}
$$

Fig. 15. Program inverter $\mathcal{I}$ for Janus statements [114].

Inverters for reversible languages Three observations about the program inverter $\mathcal{I}$, which apply to many program inverters for reversible languages:

(i) *Self-inverse statements.* A statement can be self-inverse, which means it requires no transformation at all. Here, three statements are self-inverse: `^=`, `<=>`, and `skip`. An example of a self-inversion is the Toffoli gate written in Janus: $\mathcal{I}[\![$`x1 ^= x2 && x3`$]\!] =$ `x1 ^= x2 && x3`. All other statements benefit from the availability of their direct inverse in the language (*e.g.*, the pair of assignment statements `+=` and `-=`).

(ii) *Not every statement component needs inversion.* Of the four components of the control-flow operators `if-fi` and `from-until`, only the two statements, s_1 and s_2, require inversion, while the two predicates e_1 and e_2, only swap their roles as test and assertion. Thus, e_1 and e_2 can include non-injective operators with no unique inverses, such as Boolean *and* and *or*. Likewise, the expression e in an assignment, such as x `+=` e, is unchanged when the assignment operator is inverted, x `-=` e. Thus, e can include non-injective arithmetic operators, such as `*` and `/`. The inversion of three assignment operators (`+=`, `-=`, `^=`) suffices to invert all reversible assignments. The fact that non-injective expressions can be used in reversible languages is important for their usability. The theory behind this is the reversible update discussed in Sect. 4.1, with details in [13, Sect. 3].

(iii) *Multiple options for inverting statements.* In our inverter, this is the case for `call` *id*, which can be inverted to either of the two functionally-equivalent statements, `call` $\textit{id}^{-1}$ and `uncall` *id*. How to choose? We are interested in producing small and self-contained inverse programs, so `call` *id* is inverted into a call to the inverse procedure: `call` $\textit{id}^{-1}$. If one had chosen `uncall` *id*, the inverter would need to include both procedure definitions in the inverse program, one for $\textit{id}^{-1}$ and one for *id*, which would increase the size of the program.

$\mathcal{I}$ does not add or delete any statements during the transformation (all metavariables over the program text are used linearly). Applying $\mathcal{I}$ twice returns the original program, provided that the generation of procedure names is self-inverse, *i.e.*, $(\textit{id}^{-1})^{-1} = \textit{id}$. $\mathcal{I}$ is total, bijective, and compositional and can be implemented in a reversible language like Janus without additional garbage. This case is common for reversible languages on all levels of the computation stack including reversible machine code that can be inverted on-the-fly by a reversible processor (*e.g.*, [13,35,105]). These properties are central for the construction of efficient reversible computing systems. In conventional languages, this case is rare because program inversion usually requires global analysis and has to deal with nondeterministic inverse programs (*e.g.*, [28,64,108]).

Program inversion (performed by $\mathcal{I}$) and inverse computation (invoked by `uncall`) result in equally efficient computations in Janus, and there is no need to write an inverse interpreter to perform inverse computation in Janus.

It was shown that self-interpreters for reversible languages can be constructed [8,46,114]. They perform standard interpretation, but because they are written in a reversible language, they can be inverted into inverse self-interpreters (*e.g.*, the self-interpreter SINT for Janus [114, Fig. 7] can be inverted by $\mathcal{I}$ into an equally efficient inverse self-interpreter SINT^{-1}.)

These are mind-boggling, while unfamiliar possibilities in mainstream languages. They again support the point made about the close relation of inverse computation and program inversion, and that it is, therefore, advantageous to study them together rather than in isolation.

6.3. Revisiting Bennett's classical reversibilization

Now that we have defined inverse computation and program inversion, we can describe the Bennett method within our explanatory framework.

Let p be a program written in an irreversible language L. A reversibilization *revbiz* of p extends the program into a reversible program $\overline{p}$ implemented in a reversible language R (Sect. 4.2). A program inversion of $\overline{p}$ then yields an inverse program $(\overline{p})^{-1}$ implemented in R:

$$[\![inverter]\!]\,([\![revbiz]\!]\,p) = [\![inverter]\!]\,\overline{p} = (\overline{p})^{-1}. \tag{36}$$

If the reversible language R is Janus, then the program inverter $\mathcal{I}$ presented in Fig. 15 can be used for the source-to-source transformation of $\overline{p}$ into $(\overline{p})^{-1}$. Note that (36) defines a two-step transformation of an irreversible program: reversibilization followed by program inversion.

Because we construct $\overline{p}$ and $(\overline{p})^{-1}$ using the same reversibilizer *revbiz*, we have for all x, y, and *trace* (which reminds of isolating a variable x in an algebraic equation):

$$[\![\overline{p}]\!]_R\, x = (y, trace) \iff x = [\![(\overline{p})^{-1}]\!]_R\,(y, trace). \tag{37}$$

The key idea now is to compose $\overline{p}$ and its inverse $(\overline{p})^{-1}$ with a reversible program *copy* in between to save y. For the latter, we have for all y: $[\![copy]\!]_R\, y = (y, y)$. The input of the composed program $\overline{p}^{ben}$ is x, and the output is a pair (y, x) consisting of the desired result y and the input x. The program $\overline{p}^{ben}$ has three phases. The first and the third phases run $\overline{p}$ and $(\overline{p})^{-1}$, respectively, whereby the third phase undoes whatever $\overline{p}$ computes in the first phase (*cf.* Eq. (37)). The second phase copies the value of y. Bennett's trick is that the *trace* of $\overline{p}$ is uncomputed by $(\overline{p})^{-1}$.

Here, we show $\overline{p}^{ben}$ written in reversible pseudocode with syntactic sugar:

$$\begin{array}{ll}
input\ x & \\
(y, trace) \Leftarrow [\![\overline{p}]\!]_R\, x & \text{Phase 1: compute} \\
(y, y') \Leftarrow [\![copy]\!]_R\, y & \text{Phase 2: copy} \\
x \Leftarrow [\![(\overline{p})^{-1}]\!]_R\,(y', trace) & \text{Phase 3: uncompute} \\
output\ (y, x) &
\end{array} \tag{38}$$

Except for the input variable x, all variables y, y', and *trace* are initially cleared. All variables except for the output variables y and x must be finally cleared. A *reversible replacement*, such as $(y, trace) \Leftarrow [\![\overline{p}]\!]_R\, x$ in the first phase, reversibly computes the value $[\![\overline{p}]\!]_R\, x$, thereby clears x, then updates y and *trace*, which both must be cleared, with the value's components according to the pattern $(y, trace)$. The reversible replacement is a language construct introduced in `R-WHILE` [46] and convenient in reversible programming, *e.g.*, [48], because it allows using the same variables on both sides.

Using the equations above, it is not difficult to see that for all x and y:

$$[\![p]\!]_L\, x = y \iff [\![\overline{p}^{ben}]\!]_R\, x = (y, x). \tag{39}$$

All three programs are injective: $\overline{p}$, *copy*, and $(\overline{p})^{-1}$. Their composition is injective and, thus, can be implemented in a reversible language, which was first demonstrated by Bennett for reversible Turing machines [17].

Notably, the construction in (38) is programmatic. Given an irreversible L-program p, the reversibilization $\overline{p}$ and its inverse $(\overline{p})^{-1}$ can be generated, as shown in (36). Program *copy* needs to be written only once. The trivial composition of the three programs can easily be done in R, and we get the desired R-program $\overline{p}^{ben}$ from the L-program p. Thus, an automatic input-preserving L-to-R reversibilizer *ben* can be built that satisfies Eq. (13).

The significance of the Bennett method is that it converts the implementation-dependent (intensional) garbage *trace* of the reversibilized program $\overline{p}$ into an implementation-independent (extensional) garbage x of $\overline{p}^{ben}$, and this works for *any* irreversible program p. If p is modified and replaced by a functionally equivalent but different implementation q, that is, $[\![p]\!] = [\![q]\!]$, the construction can be repeated for q, and the functionality of $\overline{q}^{ben}$ remains unchanged, $[\![\overline{p}^{ben}]\!] = [\![\overline{q}^{ben}]\!]$, even when internally the traces of $\overline{p}$ and $\overline{q}$ differ. These properties make this method so useful and widely applicable.

Inverse computation as alternative If we implement $\overline{p}^{ben}$ in a Janus-like reversible language R, then instead of using a program inverter $\mathcal{I}$, the built-in inverse computation can be invoked by an `uncall`. Formally, instead of applying the *inverse program* $[\![(\overline{p})^{-1}]\!]_R$ in the third phase of (38), we give $\overline{p}$ its *inverse semantics* $[\![\overline{p}]\!]_R^{-1}$, and use the functional equivalence

$$[\![(\overline{p})^{-1}]\!]_R = [\![\overline{p}]\!]_R^{-1}. \tag{40}$$

This means we have an inverse-computation alternative to the program inverter-based construction (38). Instead of using a program inverter, we can simply invoke the inverse computation of $\overline{p}$ in the third phase:

$$\begin{array}{ll} \mathit{input}\ x & \\ (y, \mathit{trace}) \Leftarrow [\![\overline{p}]\!]_R\, x & \text{Phase 1: compute} \\ (y, y') \Leftarrow [\![\mathit{copy}]\!]_R\, y & \text{Phase 2: copy} \\ x \Leftarrow [\![\overline{p}]\!]_R^{-1}\, (y', \mathit{trace}) & \text{Phase 3: uncompute} \\ \mathit{output}\ (y, x) & \end{array} \tag{41}$$

Thus, there are two ways, (38) and (41), to implement the Bennett method

1. $[\![(\overline{p})^{-1}]\!]_R$ using program inversion (*e.g.*, program inverter $\mathcal{I}$ in Fig. 15),
2. $[\![\overline{p}]\!]_R^{-1}$ using inverse computation (*e.g.*, `uncall` as in Fig. 8).

The second option is the programmer's preferred choice because the first and third phases can share $\overline{p}$ using inverse computation [114, Sect. 3.3]. If a reversible language R does not provide a way to invoke inverse computation, and we do not have an inverse self-interpreter for R, then the first option is the only choice, as was originally required for reversible Turing machines. This means that both programs, $\overline{p}$ and $(\overline{p})^{-1}$, must be included, which doubles the textual size of (38) compared to (41). It should be noted that *any* reversibilization $\overline{p}$ of p can be used in (38) and (41), while Bennett's original program inverter-based solution for reversible Turing machines relied on a Landauer embedding $\overline{p}^{lan}$.

The above constructions generalize the original solution [17], showing that it is independent of the reversibilization method and the programming languages and can be implemented using program inversion and inverse computation. It is not even necessary to copy y in the second phase. In some cases, it may be sufficient to make a partial copy or test only a property of y [50]. This generality makes the method versatile for reversible programming.

7. Reversible programming languages in general

As with conventional languages, there is no "best" reversible language, but a variety of possibilities as summarized in this section.

General-purpose languages (typically r-Turing-complete) such as imperative, functional, and object-oriented languages.
Domain-specific languages (not necessarily r-Turing-complete) such as hardware description languages, robot control, and hybrid reversible-irreversible languages.

Typically, general-purpose languages are broadly applicable and computationally as powerful as possible, that is, r-Turing complete, while domain-specific languages are specialized to a particular application domain and not necessarily r-Turing complete. The languages can be characterized by their abstraction level, programming paradigm, and application domain. There are structured and unstructured reversible flowcharts, low-level reversible assembler languages and high-level languages with imperative, functional, and object-oriented features supported by dynamic memory management on reversible hardware, as well as domain-specific languages, such as reversible hardware description languages, and theoretically-oriented languages for abstract computation models such as reversible pushdown automata. Moreover, hybrid languages may combine irreversible and reversible features in novel ways.

The reversible language that we used in this paper, that is, Janus, is a general-purpose language that relies on a combination of statically allocated data structures, such as fixed-size arrays, and dynamically allocated data on global stacks and on the call stack local to an invoked procedure. The advantage of this memory model is the simplicity of managing the linear allocation scheme of stacks at runtime. It is possible to work with other data structures in Janus as exemplified by the use of abstract syntax trees in a recursive self-interpreter for Janus [114, Fig. 7], but the overhead of representing and manipulating these structures through arrays and stacks tends to add to the intricacies of the reversible programs.

A step towards higher abstraction is reversible languages, where variables bind to dynamically allocated constructor terms, such as the first-order functional language `RFUN` [118]. The imperative language `R-WHILE` [46] operates on lists known from Lisp. These languages require the management of tree-structured data at runtime in the heap of a reversible machine without producing additional garbage [10,22].

Dynamic memory management considerably broadens the applicability of reversible languages by enabling the realization and manipulation of high-level data structures and algorithms that were previously difficult to realize directly in reversible programming systems. Dynamic memory management supports modern programming paradigms such as object-oriented

languages and in fact any paradigm with dynamically allocated records of varying sizes. Two examples of reversible object-oriented languages are Joule [103] and ROOPL [57,58]. They provide high-level abstractions such as encapsulation, class inheritance, and subtype polymorphism, which require a dynamic memory management to manipulate objects at runtime. A recent reversible memory manager uses a reversible version of the buddy memory algorithm and reference counting for safe, shallow reference copying [22].

On the other end of the abstraction spectrum are reversible machine languages, that is, the low-level instruction set architectures that directly control the components of a reversible computing device comprising the processing unit and the main memory. They constitute the linguistic interface between the reversible hardware and the reversible software. They are the target languages of the compilers for the high-level languages mentioned above, such as Janus and ROOPL. Examples of reversible machine languages for von-Neumann machine architectures are PISA [35,110], which is the first reversible machine language, and the minimalist machine language Bob [105]. All-in-all, this demonstrates that the design and construction of a complete reversible computing architecture is possible and can serve as the core of a *programmable reversible computing system* [10,13,14,22,57,86,104,105].

Several reversible languages have been studied for specific problems in each of their own domains. Here, we present some relatively recent ones.

First, a language for robotic assembly, called SCP-RASQ (Simple C++ Reversible Assembly SeQuences), enables automatic error recovery based on reversing the assembly process [76]. Unlike the inversion discussed in this contribution, robot assembly and disassembly programs are not exactly inverses of each other. Second, a translator from Janus to the intermediate languages including RSSA [85], *i.e.*, a reversible variant of the SSA, has been designed and implemented [72]. Inversion enables a part of the translator to unconventionally share its code. For example, the code for popping stacks is made by generating the RSSA instructions for push, which is the inverse of pop and then inverting the generated instructions. Third, a reversible combinator language, called Agni, efficiently processes arrays in parallel on reversible vector processors [87]. Fourth, the reversible version of hardware description languages has been designed and implemented for reversible circuit synthesis (*e.g.*, [112,113]).

An early point-free functional language with a relational semantics, in which all functions definable are injective, is Inv [90]. The reversibility of a class of term rewriting systems can be characterized by biorthogonality [6]. Additionally, there are hybrid languages, attempting to weaken the conditions on reversibility (*e.g.*, [20,76,80]), advanced reversible automata including nondeterminism (*e.g.*, [60,89]), and semantic foundations for reversible languages [38,52,62]. For recent activities see also [7,84].

8. Reversible computing is a field of its own

We conclude that reversible computing is a distinct computation model that complements, but does not replace, established (irreversible) paradigms. The unusualness of reversible programming motivated starting the journey from first principles. After situating reversible computing within programming languages in general, we developed the five main hypotheses that have been guiding our investigations during the past decade. We elaborated foundational concepts such as injectivization and reversibilization, and practical features representative of many reversible languages, such as reversible updates, reversible iteration, and access to a program's inverse semantics. We studied the metacomputation methods of particular importance to the field and dissected Bennett's classical reversibilization within our explanatory framework.

Our aim was the presentation of a coherent and self-supporting framework rather than a scrapbook approach merely collecting more or less related observations and results. An effective and meaningful ordering of the relevant concepts and knowledge can be an enabler for further advances and developments of the theory, algorithms, software systems, and programming languages of reversible computing. We believe that this constitutes an original view and complements previous presentations (*e.g.*, [23,89,97]).

As with research in programming languages in general and with reversible languages in particular, an overarching motivation is the constructive potential of the investigations that can eventually lead to practical software and real-world uses. Even without reversible computers, there is a great potential to the investigations because they bring new theories and tools to the table of computing.

The idea and results of the field of reversible computing are related to adjacent fields dealing with some notion of reversibility in computation in various ways. For example, a bidirectional transformation [34] consists of a pair of asymmetric forward and backward programs. Though it also has the notion of forward and backward computation, on the contrary to reversible programs, forward programs are not injective. The lost information by the forward programs is supplemented by the corresponding backward programs. Quantum versions of reversible gates can only perform a unitary transformation, *i.e.*, a bijective map over a quantum bit (qubit). As an unexpected connection, the computation power of a DNA-based computation model is demonstrated by constructing a theoretically reversible computing device [56], which is a reversible logic element with memory [89]. Reversible Turing machines are not the most restrictive computation model that has rich expressiveness, and examples are the self-inverse Turing machines [91] and linear transformations [79].

In short, reversible computation is an emerging field of computer science that encompasses all aspects of computing (theoretical, practical, technical, and applied aspects) and complements many of the traditional fields.

It is here to stay.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Many thanks to Markus Holzer and Martin Kutrib for the opportunity to present the talk at RPLA, FM Week, Porto, 2019, and to prepare this written exposition. Special thanks to Keisuke Nakano, Antonina Nepeivoda, and the anonymous reviewers for their useful feedback. It is a pleasure to thank Sergei Abramov, Holger Bock Axelsen, Poul Clementsen, Alexis De Vos, Robin Kaarsgaard, Masahiko Kawabe, Andrei V. Klimov, Torben Mogensen, Kenji Moriyama, and Michael Kirkedal Thomsen for many years of fruitful discussions. The authors also wish to thank all their collaborators, including the students and participants of the "*MicroPower*" research project [14].

The presentation has also benefited from the courses "*Program Inversion and Reversible Computation*" (PIRC) held in various ways since 2004 at the University of Copenhagen and "*Special Lecture in Information Science and Technology VII*" on reversible computation held 2016–2019 at the University of Tokyo by the first author. The first author would like to thank Yoshihiko Futamura, Masami Hagiya, Akihiko Takano, and Kanae Tsushima for their hospitality in Japan. The second author was supported by JSPS KAKENHI Grant No. 22K11983 and Nanzan University Pache Research Subsidy I-A-2 for the 2022 academic year.

References

[1] S.M. Abramov, R. Glück, Combining semantics with non-standard interpreter hierarchies, in: S. Kapoor, S. Prasad (Eds.), Foundations of Software Technology and Theoretical Computer Science. Proceedings, in: LNCS, vol. 1974, Springer, 2000, pp. 201–213, https://doi.org/10.1007/3-540-44450-5_16.

[2] S.M. Abramov, R. Glück, The universal resolving algorithm: inverse computation in a functional language, in: R. Backhouse, J.N. Oliveira (Eds.), Mathematics of Program Construction. Proceedings, in: LNCS, vol. 1837, Springer, 2000, pp. 187–212, https://doi.org/10.1007/10722010_13.

[3] S.M. Abramov, R. Glück, From standard to non-standard semantics by semantics modifiers, Int. J. Found. Comput. Sci. 12 (2001) 171–211, https://doi.org/10.1142/S0129054101000448.

[4] S.M. Abramov, R. Glück, Principles of inverse computation and the universal resolving algorithm, in: T.Æ. Mogensen, et al. (Eds.), The Essence of Computation: Complexity, Analysis, Transformation, in: LNCS, vol. 2566, Springer, 2002, pp. 269–295, https://doi.org/10.1007/3-540-36377-7_13.

[5] S.M. Abramov, R. Glück, The universal resolving algorithm and its correctness: inverse computation in a functional language, Sci. Comput. Program. 43 (2002) 193–229, https://doi.org/10.1016/S0167-6423(02)00023-0.

[6] S. Abramsky, A structural approach to reversible computation, Theor. Comput. Sci. 347 (2005) 441–464, https://doi.org/10.1016/j.tcs.2005.07.002.

[7] B. Aman, et al., Foundations of reversible computation, in: I. Ulidowski, I. Lanese, U.P. Schultz, C. Ferreira (Eds.), Reversible Computation: Extending Horizons of Computing, in: LNCS, vol. 12070, Springer, 2020, pp. 1–40, https://doi.org/10.1007/978-3-030-47361-7_1.

[8] H.B. Axelsen, R. Glück, A simple and efficient universal reversible Turing machine, in: A.-H. Dediu, S. Inenaga, C. Martín-Vide (Eds.), Language and Automata Theory and Applications. Proceedings, in: LNCS, vol. 6638, Springer, 2011, pp. 117–128, https://doi.org/10.1007/978-3-642-21254-3_8.

[9] H.B. Axelsen, R. Glück, What do reversible programs compute?, in: M. Hofmann (Ed.), Foundations of Software Science and Computation Structures. Proceedings, in: LNCS, vol. 6604, Springer, 2011, pp. 42–56, https://doi.org/10.1007/978-3-642-19805-2_4.

[10] H.B. Axelsen, R. Glück, Reversible representation and manipulation of constructor terms in the heap, in: G.W. Dueck, D.M. Miller (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 7948, Springer, 2013, pp. 96–109, https://doi.org/10.1007/978-3-642-38986-3_9.

[11] H.B. Axelsen, R. Glück, On reversible Turing machines and their function universality, Acta Inform. 53 (2016) 509–543, https://doi.org/10.1007/s00236-015-0253-y.

[12] H.B. Axelsen, T. Yokoyama, Programming techniques for reversible comparison sorts, in: X. Feng, S. Park (Eds.), Programming Languages and Systems. Proceedings, in: LNCS, vol. 9458, Springer, 2015, pp. 407–426, https://doi.org/10.1007/978-3-319-26529-2_22.

[13] H.B. Axelsen, R. Glück, T. Yokoyama, Reversible machine code and its abstract processor architecture, in: V. Diekert, et al. (Eds.), Computer Science – Theory and Applications. Proceedings, in: LNCS, vol. 4649, Springer, 2007, pp. 56–69, https://doi.org/10.1007/978-3-540-74510-5_9.

[14] H.B. Axelsen, R. Glück, A. De Vos, M.K. Thomsen, MicroPower: towards low-power microprocessors with reversible computing, ERCIM News, Special Theme: Towards Green ICT 79 (2009) 20–21.

[15] H.B. Axelsen, Clean translation of an imperative reversible programming language, in: J. Knoop (Ed.), Compiler Construction. Proceedings, in: LNCS, vol. 6601, Springer, 2011, pp. 144–163, https://doi.org/10.1007/978-3-642-19861-8_9.

[16] H.G. Baker, NREVERSAL of fortune — the thermodynamics of garbage collection, in: Y. Bekkers, J. Cohen (Eds.), International Workshop on Memory Management. Proceedings, in: LNCS, vol. 637, Springer, 1992, pp. 507–524, https://doi.org/10.1007/BFb0017210.

[17] C.H. Bennett, Logical reversibility of computation, IBM J. Res. Dev. 17 (1973) 525–532, https://doi.org/10.1147/rd.176.0525.

[18] C.H. Bennett, Time/space trade-offs for reversible computation, SIAM J. Comput. 18 (1989) 766–776, https://doi.org/10.1137/0218053.

[19] A. Bérut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, E. Lutz, Experimental verification of Landauer's principle linking information and thermodynamics, Nature 483 (2012) 187–189, https://doi.org/10.1038/nature10872.

[20] J.S. Briggs, Generating reversible programs, Softw. Pract. Exp. 17 (1987) 439–453, https://doi.org/10.1002/spe.4380170703.

[21] C.D. Carothers, K.S. Perumalla, R.M. Fujimoto, Efficient optimistic parallel simulations using reverse computation, ACM Trans. Model. Comput. Simul. 9 (1999) 224–253, https://doi.org/10.1145/347823.347828.

[22] M.H. Cservenka, R. Glück, T. Haulund, T.Æ. Mogensen, Data structures and dynamic memory management in reversible languages, in: J. Kari, I. Ulidowski (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 11106, Springer, 2018, pp. 269–285, https://doi.org/10.1007/978-3-319-99498-7_19.

[23] A. De Vos, Reversible Computing: Fundamentals, Quantum Computing, and Applications, Wiley, 2010, https://doi.org/10.1002/9783527633999.

[24] A. De Vos, Endoreversible models for the thermodynamics of computing, Entropy 22 (2020) 660, https://doi.org/10.3390/e22060660.

[25] N. Deworetzki, U. Meyer, Program analysis for reversible languages, in: L.N.Q. Do, C. Urban (Eds.), International Workshop on the State of the Art in Program Analysis. Proceedings, ACM, 2021, pp. 13–18, https://doi.org/10.1145/3460946.3464314.

[26] E.W. Dijkstra, Program inversion, in: F.L. Bauer, M. Broy (Eds.), Program Construction: International Summer School, in: LNCS, vol. 69, Springer, 1978, pp. 54–57, https://doi.org/10.1007/BFb0014657.

[27] E.W. Dijkstra, On the role of scientific thought (EWD447), in: Selected Writings on Computing: A Personal Perspective, Springer, 1982, pp. 60–66, https://doi.org/10.1007/978-1-4612-5695-3_12.
[28] D. Eppstein, A heuristic approach to program inversion, in: A.K. Joshi (Ed.), International Joint Conference on Artificial Intelligence (IJCAI-85). Proceedings, vol. 1, Morgan Kaufmann, Inc., 1985, pp. 219–221.
[29] A.P. Ershov, On the essence of compilation, in: E. Neuhold (Ed.), Formal Description of Programming Concepts, North-Holland, 1978, pp. 391–420.
[30] M. Fernández, I. Mackie, A reversible operational semantics for imperative programming languages, in: S. Lin, Z. Hou, B.P. Mahony (Eds.), Formal Methods and Software Engineering. Proceedings, in: LNCS, vol. 12531, Springer, 2020, pp. 91–106, https://doi.org/10.1007/978-3-030-63406-3_6.
[31] R.P. Feynman, Quantum mechanical computers, Found. Phys. 16 (1986) 507–531, https://doi.org/10.1007/BF01886518.
[32] R.P. Feynman, Reversible computation and the thermodynamics of computing (chapter 5), in: A.J.G. Hey, R.W. Allen (Eds.), Feynman Lectures on Computation, Addison-Wesley, 1996, pp. 137–184, https://doi.org/10.1201/9780429500442.
[33] R.W. Floyd, Nondeterministic algorithms, J. ACM 14 (1967) 636–644, https://doi.org/10.1145/321420.321422.
[34] J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, A. Schmitt, Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem, ACM Trans. Program. Lang. Syst. 29 (2007) 17, https://doi.org/10.1145/1232420.1232424.
[35] M.P. Frank, Reversibility for Efficient Computing, Ph.D. thesis, MIT, Cambridge, Massachusetts, 1999.
[36] E. Fredkin, T. Toffoli, Conservative logic, Int. J. Theor. Phys. 21 (1982) 219–253, https://doi.org/10.1007/BF01857727.
[37] Y. Futamura, Partial evaluation of computing process – an approach to a compiler-compiler, Syst. Comput. Controls 2 (1971) 45–50.
[38] R. Glück, R. Kaarsgaard, A categorical foundation for structured reversible flowchart languages: soundness and adequacy, Log. Methods Comput. Sci. 14 (2018) 16, https://doi.org/10.23638/LMCS-14(3:16)2018.
[39] R. Glück, M. Kawabe, A program inverter for a functional language with equality and constructors, in: A. Ohori (Ed.), Programming Languages and Systems. Proceedings, in: LNCS, vol. 2895, Springer, 2003, pp. 246–264, https://doi.org/10.1007/978-3-540-40018-9_17.
[40] R. Glück, M. Kawabe, A method for automatic program inversion based on LR(0) parsing, Fundam. Inform. 66 (2005) 367–395.
[41] R. Glück, M. Kawabe, Revisiting an automatic program inverter for Lisp, SIGPLAN Not. 40 (2005) 8–17, https://doi.org/10.1145/1071221.1071222.
[42] R. Glück, A.V. Klimov, Metasystem transition schemes in computer science and mathematics, World Futures 45 (1995) 213–243, https://doi.org/10.1080/02604027.1995.9972561.
[43] R. Glück, A.V. Klimov, On the degeneration of program generators by program composition, New Gener. Comput. 16 (1998) 75–95, https://doi.org/10.1007/BF03037321.
[44] R. Glück, M. Leuschel, Abstraction-based partial deduction for solving inverse problems: a transformational approach to software verification, in: D. Bjørner, M. Broy, A.V. Zamulin (Eds.), Perspectives of System Informatics. Proceedings, in: LNCS, vol. 1755, Springer, 2000, pp. 93–100, https://doi.org/10.1007/3-540-46562-6_8.
[45] R. Glück, V.F. Turchin, Application of metasystem transition to function inversion and transformation, in: International Symposium on Symbolic and Algebraic Computation. Proceedings, ACM Press, 1990, pp. 286–287, https://doi.org/10.1145/96877.96953.
[46] R. Glück, T. Yokoyama, A linear-time self-interpreter of a reversible imperative language, Comput. Softw. 33 (2016) 108–128, https://doi.org/10.11309/jssst.33.3_108.
[47] R. Glück, T. Yokoyama, A minimalist's reversible while language, IEICE Trans. Inf. Syst. E100-D (2017) 1026–1034, https://doi.org/10.1587/transinf.2016EDP7274.
[48] R. Glück, T. Yokoyama, Constructing a binary tree from its traversals by reversible recursion and iteration, Inf. Process. Lett. 147 (2019) 32–37, https://doi.org/10.1016/j.ipl.2019.03.002.
[49] R. Glück, T. Yokoyama, Making programs reversible with minimal extra data, New Gener. Comput. 40 (2022) 467–480, https://doi.org/10.1007/s00354-022-00169-z.
[50] R. Glück, T. Yokoyama, Reversible programming: a case study of two string-matching algorithms, in: G.W. Hamilton, T. Kahsai, M. Proietti (Eds.), Horn Clauses for Verification and Synthesis, Verification and Program Transformation. Proceedings, in: Electronic Proceedings in Theoretical Computer Science, vol. 373, 2022, pp. 1–13, https://doi.org/10.4204/EPTCS.373.1.
[51] R. Glück, Y. Kawada, T. Hashimoto, Transforming interpreters into inverse interpreters by partial evaluation, in: Partial Evaluation and Semantics-Based Program Manipulation. Proceedings, ACM Press, 2003, pp. 10–19, https://doi.org/10.1145/777388.777391.
[52] R. Glück, R. Kaarsgaard, T. Yokoyama, From reversible programming languages to reversible metalanguages, Theor. Comput. Sci. 920 (2022) 46–63, https://doi.org/10.1016/j.tcs.2022.02.024.
[53] R. Glück, Reversible computing from a programming language perspective (extended abstract), in: E. Sekerinski, et al. (Eds.), Formal Methods. FM 2019 International Workshops. Part II. Proceedings, in: LNCS, vol. 12233, Springer, 2020, pp. 410–412.
[54] GNU Project, GDB: the GNU project debugger. GDB and reverse debugging, https://www.gnu.org/s/gdb/news/reversible, 2009. (Accessed November 2022).
[55] D. Gries, The Science of Programming, Springer, 1981, pp. 265–274, Chapter 21: Inverting Programs, Texts and Monographs in Computer Science.
[56] M. Hagiya, S. Wang, I. Kawamata, S. Murata, T. Isokawa, F. Peper, K. Imai, On DNA-based gellular automata, in: O.H. Ibarra, L. Kari, S. Kopecki (Eds.), Unconventional Computation and Natural Computation. Proceedings, Springer, 2014, pp. 177–189, https://doi.org/10.1007/978-3-319-08123-6_15.
[57] T. Haulund, T.Æ. Mogensen, R. Glück, Implementing reversible object-oriented language features on reversible machines, in: P. Iain, R. Hafizur (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 10301, Springer, 2017, pp. 66–73, https://doi.org/10.1007/978-3-319-59936-6_5.
[58] L. Hay-Schmidt, R. Glück, M.H. Cservenka, T. Haulund, Towards a unified language architecture for reversible object-oriented programming, in: S. Yamashita, T. Yokoyama (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 12805, Springer, 2021, pp. 96–106, https://doi.org/10.1007/978-3-030-79837-6_6.
[59] J. Hoey, I. Ulidowski, Reversing an imperative concurrent programming language, Sci. Comput. Program. 223 (2022) 102873, https://doi.org/10.1016/j.scico.2022.102873.
[60] M. Holzer, M. Kutrib, Reversible nondeterministic finite automata, in: I. Phillips, H. Rahaman (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 10301, Springer, 2017, pp. 35–51, https://doi.org/10.1007/978-3-319-59936-6_3.
[61] N.D. Jones, C.K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, https://www.itu.dk/people/sestoft/pebook/pebook.html, 1993.
[62] R. Kaarsgaard, H.B. Axelsen, R. Glück, Join inverse categories and reversible recursion, J. Log. Algebraic Methods Program. 87 (2017) 33–50, https://doi.org/10.1016/j.jlamp.2016.08.003.
[63] J. Kari, Reversible cellular automata: from fundamental classical results to recent developments, New Gener. Comput. 36 (2018) 145–172, https://doi.org/10.1007/s00354-018-0034-6.
[64] M. Kawabe, R. Glück, The program inverter LRinv and its structure, in: M. Hermenegildo, D. Cabeza (Eds.), Practical Aspects of Declarative Languages. Proceedings, in: LNCS, vol. 3350, Springer, 2005, pp. 219–234, https://doi.org/10.1007/978-3-540-30557-6_17.
[65] M.H. Kirkeby, R. Glück, Semi-inversion of conditional constructor term rewriting systems, in: M. Gabbrielli (Ed.), Logic-Based Program Synthesis and Transformation. Proceedings, in: LNCS, vol. 12042, Springer, 2020, pp. 243–259, https://doi.org/10.1007/978-3-030-45260-5_15.
[66] W. Kluge, A reversible SE(M)CD machine, in: P. Koopman, C. Clack (Eds.), Implementation of Functional Languages. Proceedings, in: LNCS, vol. 1868, Springer, 2000, pp. 95–113, https://doi.org/10.1007/10722298_6.

[67] R.E. Korf, Inversion of applicative programs, in: International Joint Conference on Artificial Intelligence (IJCAI-81). Proceedings, William Kaufmann, Inc., 1981, pp. 1007–1009.
[68] R. Kowalski, Predicate logic as programming language, in: J.L. Rosenfeld (Ed.), Information Processing, vol. 74, North-Holland, 1974, pp. 569–574.
[69] R. Kowalski, Algorithm = logic + control, Commun. ACM 22 (1979) 424–436, https://doi.org/10.1145/359131.359136.
[70] M. Krakovsky, Taking the heat, Commun. ACM 64 (2021) 18–20, https://doi.org/10.1145/3460214.
[71] R. Královič, Time and space complexity of reversible pebbling, RAIRO Theor. Inform. Appl. 38 (2004) 137–161, https://doi.org/10.1051/ita:2004008.
[72] M. Kutrib, U. Meyer, N. Deworetzki, M. Schuster, Compiling Janus to RSSA, in: S. Yamashita, T. Yokoyama (Eds.), Reversible Computation. Proceedings, 2021, pp. 64–78, https://doi.org/10.1007/978-3-030-79837-6_4.
[73] R. Landauer, Irreversibility and heat generation in the computing process, IBM J. Res. Dev. 5 (1961) 183–191, https://doi.org/10.1147/rd.53.0183.
[74] I. Lanese, N. Nishida, A. Palacios, G. Vidal, CauDEr: a causal-consistent reversible debugger for Erlang, in: J. Gallagher, M. Sulzmann (Eds.), Functional and Logic Programming. Proceedings, in: LNCS, vol. 10818, Springer, 2018, pp. 247–263, https://doi.org/10.1007/978-3-319-90686-7_16.
[75] K.-J. Lange, P. McKenzie, A. Tapp, Reversible space equals deterministic space, J. Comput. Syst. Sci. 60 (2000) 354–367, https://doi.org/10.1006/jcss.1999.1672.
[76] J.S. Laursen, L.-P. Ellekilde, U.P. Schultz, Modelling reversible execution of robotic assembly, Robotica 36 (2018) 625–654, https://doi.org/10.1017/S0263574717000613.
[77] G.B. Leeman, A formal approach to undo operations in programming languages, ACM Trans. Program. Lang. Syst. 8 (1986) 50–97, https://doi.org/10.1145/5001.5005.
[78] C. Lutz, Janus: a time-reversible language, A Letter to R. Landauer, Caltech, https://www.tetsuo.jp/ref/janus.html, 1986.
[79] A.B. Matos, Linear programs in a simple reversible language, Theor. Comput. Sci. 290 (2003) 2063–2074, https://doi.org/10.1016/S0304-3975(02)00486-3.
[80] K. Matsuda, M. Wang, Sparcl: a language for partially-invertible computation, Proc. ACM Program. Lang. 4 (2020) 118, https://doi.org/10.1145/3409000.
[81] J. McCarthy, The inversion of functions defined by Turing machines, in: C.E. Shannon, J. McCarthy (Eds.), Automata Studies, Princeton University Press, 1956, pp. 177–181, https://doi.org/10.1515/9781400882618-009.
[82] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Commun. ACM 3 (1960) 184–195, https://doi.org/10.1145/367177.367199.
[83] G. Meuli, M. Soeken, M. Roetteler, N. Bjørner, G. De Micheli, Reversible pebbling game for quantum memory management, in: Design, Automation & Test in Europe Conference & Exhibition (DATE). Proceedings, EDAA, 2019, pp. 288–291, https://doi.org/10.23919/DATE.2019.8715092.
[84] C.A. Mezzina, et al., Software and reversible systems: a survey of recent activities, in: I. Ulidowski, I. Lanese, U.P. Schultz, C. Ferreira (Eds.), Reversible Computation: Extending Horizons of Computing, in: LNCS, vol. 12070, Springer, 2020, pp. 41–59, https://doi.org/10.1007/978-3-030-47361-7_2.
[85] T.Æ. Mogensen, RSSA: a reversible SSA form, in: M. Mazzara, A. Voronkov (Eds.), Perspectives of System Informatics. Proceedings, in: LNCS, vol. 9609, Springer, 2016, pp. 203–217, https://doi.org/10.1007/978-3-319-41579-6_16.
[86] T.Æ. Mogensen, Reversible garbage collection for reversible functional languages, New Gener. Comput. 36 (2018) 203–232, https://doi.org/10.1007/s00354-018-0037-3.
[87] T.Æ. Mogensen, Reversible functional array programming, in: S. Yamashita, T. Yokoyama (Eds.), Reversible Computation. Proceedings, Springer, 2021, pp. 45–63, https://doi.org/10.1007/978-3-030-79837-6_3.
[88] T.Æ. Mogensen, Hermes: a reversible language for lightweight encryption, Sci. Comput. Program. 215 (2022) 102746, https://doi.org/10.1016/j.scico.2021.102746.
[89] K. Morita, Theory of Reversible Computing, Monographs in Theoretical Computer Science, Springer, 2017, https://doi.org/10.1007/978-4-431-56606-9.
[90] S.-C. Mu, Z. Hu, M. Takeichi, An injective language for reversible computation, in: D. Kozen (Ed.), Mathematics of Program Construction. Proceedings, in: LNCS, vol. 3125, Springer, 2004, pp. 289–313, https://doi.org/10.1007/978-3-540-27764-4_16.
[91] K. Nakano, Involutory Turing machines, in: I. Lanese, M. Rawski (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 12227, Springer, 2020, pp. 54–70, https://doi.org/10.1007/978-3-030-52482-1_3.
[92] K. Nakano, Time-symmetric Turing machines for computable involutions, Sci. Comput. Program. 215 (2022) 102748, https://doi.org/10.1016/j.scico.2021.102748.
[93] N. Nishida, A. Palacios, G. Vidal, Reversible computation in term rewriting, J. Log. Algebraic Methods Program. 94 (2018) 128–149, https://doi.org/10.1016/j.jlamp.2017.10.003.
[94] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, N. Partush, Engineering record and replay for deployability, in: USENIX Annual Technical Conference. Proceedings, USENIX Association, 2017, pp. 377–389.
[95] L. Paolini, M. Piccolo, L. Roversi, On a class of reversible primitive recursive functions and its Turing-complete extensions, New Gener. Comput. 36 (2018) 233–256, https://doi.org/10.1007/s00354-018-0039-1.
[96] K.S. Perumalla, R.M. Fujimoto, Source-code transformations for efficient reversibility, Technical Report GIT-CC-99-21, Georgia Institute of Technology, 1999.
[97] K.S. Perumalla, Introduction to Reversible Computing, CRC Press, 2014, https://doi.org/10.1201/b15719.
[98] T. Pesu, I. Phillips, Real-time methods in reversible computation, in: J. Krivine, J.-B. Stefani (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 9138, Springer, 2015, pp. 45–59, https://doi.org/10.1007/978-3-319-20860-2_3.
[99] J.-E. Pin, On the languages accepted by finite reversible automata, in: T. Ottmann (Ed.), Automata, Languages and Programming. Proceedings, in: LNCS, vol. 267, Springer, 1987, pp. 237–249, https://doi.org/10.1007/3-540-18088-5_19.
[100] E.D. Reilly Jr., F.D. Federighi, On reversible subroutines and computers that run backwards, Commun. ACM 8 (1965) 557–558, 578, https://doi.org/10.1145/365559.365593.
[101] M. Schordan, T. Oppelstrup, D. Jefferson, P.D. Barnes Jr., Generation of reversible C++ code for optimistic parallel discrete event simulation, New Gener. Comput. 36 (2018) 257–280, https://doi.org/10.1007/s00354-018-0038-2.
[102] M. Schordan, T. Oppelstrup, M.K. Thomsen, R. Glück, Reversible languages and incremental state saving in optimistic parallel discrete event simulation, in: I. Ulidowski, I. Lanese, U.P. Schultz, C. Ferreira (Eds.), Reversible Computation: Extending Horizons of Computing, in: LNCS, vol. 12070, Springer, 2020, pp. 187–207, https://doi.org/10.1007/978-3-030-47361-7_9.
[103] U.P. Schultz, H.B. Axelsen, Elements of a reversible object-oriented language, in: S. Devitt, I. Lanese (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 9720, Springer, 2016, pp. 153–159, https://doi.org/10.1007/978-3-319-40578-0_10.
[104] M.K. Thomsen, R. Glück, H.B. Axelsen, Reversible arithmetic logic unit for quantum arithmetic, J. Phys. A, Math. Theor. 43 (2010) 382002, https://doi.org/10.1088/1751-8113/43/38/382002.
[105] M.K. Thomsen, H.B. Axelsen, R. Glück, A reversible processor architecture and its reversible logic design, in: A. De Vos, R. Wille (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 7165, Springer, 2012, pp. 30–42, https://doi.org/10.1007/978-3-642-29517-1_3.
[106] T. Toffoli, N. Margolus, Invertible cellular automata: a review, Physica D 45 (1990) 229–253, https://doi.org/10.1016/0167-2789(90)90185-R.
[107] T. Toffoli, Reversible computing, in: J.W. de Bakker, J. van Leeuwen (Eds.), Automata, Languages and Programming. Proceedings, in: LNCS, vol. 85, Springer, 1980, pp. 632–644, https://doi.org/10.1007/3-540-10003-2_104.
[108] V.F. Turchin, R. Nirenberg, D. Turchin, Experiments with a supercompiler, in: Lisp and Functional Programming. Proceedings, ACM Press, 1982, pp. 47–55, https://doi.org/10.1145/800068.802134.

[109] G. Vidal, Reversible computations in logic programming, in: I. Lanese, M. Rawski (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 12227, Springer, 2020, pp. 246–254, https://doi.org/10.1007/978-3-030-52482-1_15.
[110] J. Vieri, M.J. Ammer, M.P. Frank, N. Margolus, T. Knight, A fully reversible asymptotically zero energy microprocessor, in: ISCA Workshop on Power-Driven Microarchitecture. Proceedings, 1998, pp. 138–142.
[111] P. Vitányi, Time, space, and energy in reversible computing, in: Conference on Computing Frontiers. Proceedings, ACM Press, 2005, pp. 435–444, https://doi.org/10.1145/1062261.1062335.
[112] R. Wille, R. Drechsler, Towards a Design Flow for Reversible Logic, Springer, 2010, https://doi.org/10.1007/978-90-481-9579-4.
[113] R. Wille, E. Schönborn, M. Soeken, R. Drechsler, SyReC: a hardware description language for the specification and synthesis of reversible circuits, Integration 53 (2016) 39–53, https://doi.org/10.1016/j.vlsi.2015.10.001.
[114] T. Yokoyama, R. Glück, A reversible programming language and its invertible self-interpreter, in: Partial Evaluation and Program Manipulation. Proceedings, ACM Press, 2007, pp. 144–153, https://doi.org/10.1145/1244381.1244404.
[115] T. Yokoyama, H.B. Axelsen, R. Glück, Principles of a reversible programming language, in: Conference on Computing Frontiers. Proceedings, ACM Press, 2008, pp. 43–54, https://doi.org/10.1145/1366230.1366239.
[116] T. Yokoyama, H.B. Axelsen, R. Glück, Reversible flowchart languages and the structured reversible program theorem, in: L. Aceto, et al. (Eds.), Automata, Languages and Programming. Proceedings. Part II, in: LNCS, vol. 5126, Springer, 2008, pp. 258–270, https://doi.org/10.1007/978-3-540-70583-3_22.
[117] T. Yokoyama, H.B. Axelsen, R. Glück, Optimizing reversible simulation of injective functions, J. Mult.-Valued Log. Soft Comput. 18 (2012) 5–24.
[118] T. Yokoyama, H.B. Axelsen, R. Glück, Towards a reversible functional language, in: A. De Vos, R. Wille (Eds.), Reversible Computation. Proceedings, in: LNCS, vol. 7165, Springer, 2012, pp. 14–29, https://doi.org/10.1007/978-3-642-29517-1_2.
[119] T. Yokoyama, H.B. Axelsen, R. Glück, Fundamentals of reversible flowchart languages, Theor. Comput. Sci. 611 (2016) 87–115, https://doi.org/10.1016/j.tcs.2015.07.046.