Theoretical Computer Science 611 (2016) 87–115

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Fundamentals of reversible flowchart languages

Tetsuo Yokoyama [a,*], Holger Bock Axelsen [b,*], Robert Glück [b,*]

[a] Department of Software Engineering, Nanzan University, Seirei-cho 27, Seto city, Aichi 489-0863, Japan
[b] DIKU, Department of Computer Science, University of Copenhagen, DK-2100 Copenhagen, Denmark

ARTICLE INFO

Article history:
Received 22 September 2013
Received in revised form 25 March 2015
Accepted 25 July 2015
Available online 30 July 2015
Communicated by P.-A. Mellies

Keywords:
Flowchart
Reversible computing
Program inversion
Structured programming
Structured program theorem
r-Turing-completeness

ABSTRACT

This paper presents the fundamentals of *reversible flowcharts*. Reversible flowcharts are intended to naturally represent the structure and control flow of reversible (imperative) programming languages in a simple computation model in the same way classical flowcharts do for conventional languages. Although reversible flowcharts are superficially similar to classical flowcharts, there are crucial differences: atomic steps are limited to locally invertible operations, and join points require an explicit orthogonalizing conditional expression.

Despite these constraints, we show that reversible flowcharts are both expressive and robust: reversible flowcharts can simulate irreversible ones by adapting reversibilization techniques to the flowchart model. Thus, reversible flowcharts are *r-Turing-complete*, meaning that they can compute exactly all injective computable functions. Furthermore, *structured* reversible flowcharts are as expressive as *unstructured* ones, as shown by a *reversible* version of the classic Structured Program Theorem.

We illustrate how reversible flowcharts can be concretized with two example programming languages, complete with syntax and semantics: a low-level unstructured language and a high-level structured language. We introduce concrete tools such as program inverters and translators for both languages, which follow the structure suggested by the flowchart model. To further illustrate the different concepts and tools brought together in this paper, we present two major worked examples: a reversible permutation-to-code algorithm attributed to Dijkstra, and a simulation scheme for reversible Turing machines. By exhibiting a wide range of uses, we hope that the proposed reversible flowcharts can serve as a springboard for further theoretical research in reversible computing.

1. Introduction

Reversible flowcharts are intended as a model of reversible programming languages in much the same way that classic flowcharts are a model of many widely used programming languages. Flowcharts naturally represent the control flow and structure of imperative programs. However, it is not always possible for the basic theories of standard computation to directly carry over to reversible languages (cf. [5]). As a consequence, an intuitive understanding of reversible programming languages is not necessarily suggested by studying classical models. For example, reversible programming languages are *not* Turing-complete when we consider which functions are computable, cf. Axelsen and Glück [5]. Moreover, if the configuration space is bounded, then reversible programs always terminate. Furthermore, it is not immediately obvious which of the

* Corresponding authors.
E-mail addresses: tyokoyama@acm.org (T. Yokoyama), funkstar@diku.dk (H.B. Axelsen), glueck@acm.org (R. Glück).

http://dx.doi.org/10.1016/j.tcs.2015.07.046
0304-3975/

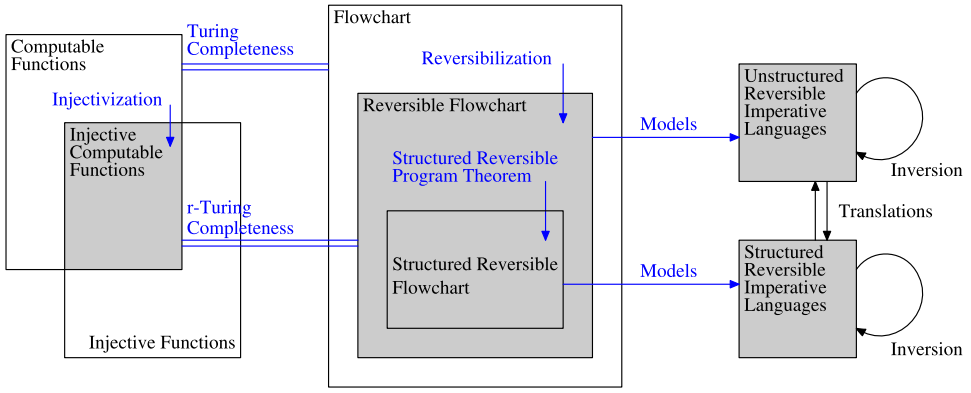


Fig. 1. Mathematical functions, flowcharts, and reversible imperative programming languages.

classic results that we take for granted in standard language theory, e.g., the Structured Program Theorem [11], carry over to a reversible setting.

Although reversible flowcharts are superficially similar to classical flowcharts, there are crucial differences: atomic steps are limited to *locally invertible* operations, and join points require an explicit *orthogonalizing* conditional expression. This ensures that *no information is lost* at any step during the computation process making every step forward and backward deterministic. Owing to these requirements, reversible flowcharts exactly capture the computational power and the distinct properties of reversible programming in a simple, theoretically and practically meaningful formalism that is fruitful to study separate from classic formalisms.

Flowcharts have the great advantage that they accommodate rather diverse programming language concepts in the same framework. The reversible flowchart model presented here can be used for a surprisingly wide range of purposes. It accommodates the low-level aspects of compiler-generated machine code [3,23], such as jumps and other assembly-level instructions, and high-level aspects of block-structured general-purpose languages, such as iteration and conditional statements [23,40,67]. Moreover, the model is independent of the concrete physical realization and organization of the underlying computer hardware, e.g., reversible logic devices [18,61,54] and reversible microprocessors [57]. Its graphical form was found to be useful for describing the heap manipulation of reversible memory management [6] and a universal reversible Turing machine (RTM) [4] as well as in specifying partial evaluation for a reversible language [46] and control flow (and their inverses) in various reversible languages [63,18,66]. Distilling the essence of such languages into a common formalism can serve as a basis for designing reversible languages at different levels of the computation stack and the corresponding tools for reversible programming such as program inverters and compilers for different reversible architectures. Their unique programming methodology may also shed light on a new style and modularity of programming [23,63]. The secondary purpose of this paper is to encourage future work on the theoretical and practical nature of reversible computation.

A separate approach to reversibility is to "hide garbage under the carpet" by having a runtime system take full control of continually producing information to ensure reversibility, without any user interaction or control (e.g., [19,36,16]). We do not follow such an approach here but instead regard the management of garbage data as a problem to be dealt with explicitly at the language level. Even though they often take advantage of reversible concepts, related fields such as program inversion [1,21,29,44] and bidirectional transformation [22,33] apply in a more conventional language context. In general, program inversion can also deal with irreversible programs, and operators for bidirectional transformation have projections on irreversible programs that can return part of a given input. This takes such work out of the scope of this paper.

Our investigation centers around reversible flowcharts as a computation model for reversible programming languages. To aid the reader in understanding the distinct features brought together in this paper, Fig. 1 gives an overview of the main topics and their relations. (The shaded areas represent the parts related to reversible flowcharts.) Compared to classical flowcharts, *reversible* flowcharts place restrictions on join points and step operators that ensure the forward and backward determinism of the computation. As a result, they are no longer Turing-complete but *r-Turing-complete*, which means that they cannot compute all computable functions, but instead the proper subset of *injective computable functions*, as indicated on the left-hand side of the figure. (Clearly, there also exist *injective* functions outside the *computable* functions.) These relations are indicated in the figure, where "Turing Completeness" and "r-Turing Completeness" equate the corresponding sets with the flowchart world.

Every computable function can be embedded in an injective computable function by adding additional output that ensures the required injectivity (a trivial injectivization returns a pair consisting of both the original input and output). The extra, but originally unnecessary, part of the output is called *garbage*. We call the transformation of flowcharts into reversible flowcharts *reversibilization*, which corresponds to the mentioned *injectivization* of functions in the mathematical world. This term highlights that this produces reversible programs instead of injective functions.

As with classic flowcharts, structure is necessary to tame "reversible spaghetti code" that can occur in reversible flowcharts. This is done by limiting the unstructured control to three *structured* reversible control flow operators: sequence,

selection, and loop. The *Structured Reversible Program Theorem*, which we will prove in this paper, guarantees that all reversible flowcharts can be mapped into structured ones. This means that structured reversible flowcharts are as powerful and expressive as their unstructured versions, and thus also r-Turing-complete.

Unstructured reversible flowcharts are well suited as a model of low-level reversible machine code, whereas their structured variety models the class of high-level block-structured reversible programming languages. Tools such as translators from high-level languages to low-level machine code are essential for practical computing systems. In the world of reversible languages, we require that such tools are *clean*; that is, the transformation does not introduce any garbage in the generated target programs. The existence of clean tools will be demonstrated for two small reversible imperative languages—one structured and one unstructured.

Every injective computable function has an inverse [44], and so does every reversible program [10,5]. The inverse program of a reversible program can be mechanically obtained by *program inversion*. Source-to-source program inverters for both reversible programming languages will be given, which are indicated by the two circular arrows labeled "Inversion" on the right-hand side of Fig. 1.

The presentation of the reversible flowchart model is organized as follows. Section 2 introduces the elements of unstructured and structured reversible flowcharts and discusses concepts such as inversion, termination under bounded space, reversibilization, and the r-Turing completeness of reversible flowcharts. As a running example, the Fibonacci-pair function is used. Section 3 proves the Structured Reversible Program Theorem. The proof uses a clean translation method that generates a structured reversible flowchart from an arbitrary reversible flowchart with exactly the same functionality. Section 4 introduces two small reversible imperative programming languages with complete syntax and structural operational semantics as concrete languages based on the flowchart formalism. Section 5 defines two program inverters and a translator from the structured language to the unstructured language. Section 6 presents applications of these tools, including the inversion of Dijkstra's permutation-to-code encoder and a clean translation of RTMs into reversible flowcharts, directly demonstrating the r-Turing completeness of our two languages. Finally, Section 7 discusses related work and Section 8 concludes the paper.

This paper is a substantially extended and revised version of our conference contribution [64]. The flowchart semantics are more detailed, the proof of the Structured Reversible Program Theorem is more precise and its construction improved, and a complete concrete instance of the construction is given. The formal semantics of two example languages is provided, their main properties are proven, and the guidelines for their design are introduced. The clean translation from unstructured to structured flowcharts is improved, and translation is shown for both example languages. The reversible Turing machine simulation is refined, and exemplified by a program in the structured example language. More detailed discussions of the theoretical aspects of the computation model complete the presentation. It is our hope that the fundamentals of reversible flowchart languages presented in this paper will encourage the further development of reversible computing systems. This work is part of a larger effort on the development of reversible computer systems including reversible logic gates and their physical realization [18,61,54], computer architectures [7,58], and programming languages and abstract machines [2,42,48].

2. Reversible flowcharts

Flowcharts have been used extensively in the study of programming languages. Most practical programming languages used today have a control flow that can be easily modeled by flowcharts. They are important analytical tools in programming language theory (e.g., [11,14,28,31,41]). For this reason, we define *reversible flowcharts* that are suitable for modeling the control flow behavior of reversible imperative programming languages in this section.

2.1. Reversible flowcharts

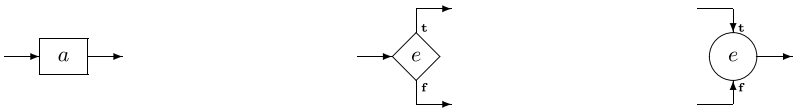
A *reversible flowchart* F is a finite directed graph with three types of nodes, each of which represents an *atomic operation* (Fig. 2):

(a) a *step* performs an elementary operation on a domain specified by an injective function a and passes incoming control flow to the outgoing edge,
(b) a *test* dispatches the control flow depending on the value of the predicate e, and
(c) an *assertion* is a join point that passes incoming control flow depending on the value of the predicate e.

Computation in a flowchart proceeds sequentially through the edges of F. An interpretation of a flowchart F consists of a domain X (e.g., a store) and an appropriate association with the partial step functions ($a : X \rightharpoonup X$) and predicates ($e : X \to \mathit{Bool}$), all of which must be computable.

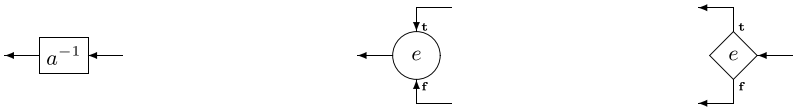
A reversible flowchart that is only constructed by atomic operations with single edges connected to only one *entry point* and one *exit point* that are distinct, is said to be *well formed*. Under a given interpretation, a well-formed reversible flowchart F computes an *injective function* $[\![F]\!] : X \rightharpoonup X$.

The assertion is a new flowchart operator: the predicate e must be **true** when the control flow reaches the join point along the **true**-edge (labeled **t**), and **false** when the control flow reaches the join point along the **false**-edge (labeled **f**); otherwise, the operation is undefined. Furthermore, the step function a of each step in a reversible flowchart must be *locally*



(a) Step (b) Test (c) Assertion

Fig. 2. Atomic operations of reversible flowcharts.



(a) Step (b) Assertion (c) Test

Fig. 3. Inverted atomic operations of reversible flowcharts.

invertible by having an inverse step function a^{-1} that can be determined without referring to the context of a or its location in the flowchart. Reversible updates [7] are locally invertible, and they can serve as step functions of a reversible flowchart.

We have removed all sources of *backward non-determinism* in reversible flowcharts, thereby making them *backward deterministic*. The backward non-determinism of control flow in classic flowcharts is due the absence of information about the direction of incoming control flow at join points. In reversible flowcharts, the replacements for join points, that is, assertions, are each associated with a predicate that provides this information. Because the incoming control flow is unique in any context, the control flow of assertions is backward deterministic.

More formally, a *configuration* of a reversible flowchart F consists of a value in the interpretation domain X and the edge on which control flow is set, from the edge set L. The computation and control flow of each atomic operation in a reversible flowchart are modeled by a transition function $\delta : X \times L \rightharpoonup X \times L$ on configurations:

$\delta(x, l) = (a(x), l')$ if a *step* has function a, incoming edge l and outgoing edge l'

$\delta(x, l) = (x, l')$ if a *test* has predicate e, incoming edge l and outgoing edge l' labeled **t**, and $e(x)$

$\delta(x, l) = (x, l')$ if a *test* has predicate e, incoming edge l and outgoing edge l' labeled **f**, and $\neg e(x)$

$\delta(x, l) = (x, l')$ if an *assertion* has predicate e, incoming edge l labeled **t** and outgoing edge l', and $e(x)$

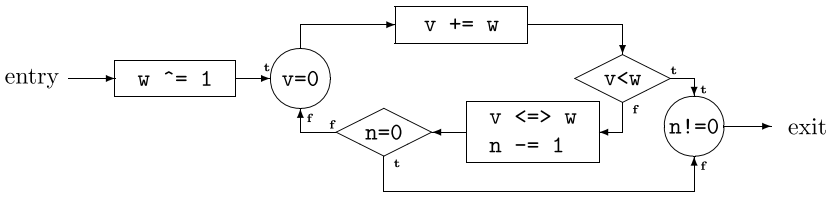
$\delta(x, l) = (x, l')$ if an *assertion* has predicate e, incoming edge l labeled **f** and outgoing edge l', and $\neg e(x)$

The first case corresponds to a step, the second and third to a test, and the last two to an assertion. By case analysis, the transition function δ for a reversible flowchart is easily seen to always be injective. In other words, a flowchart is *reversible* at the atomic-level operations in the sense that δ is an injective function on configurations.

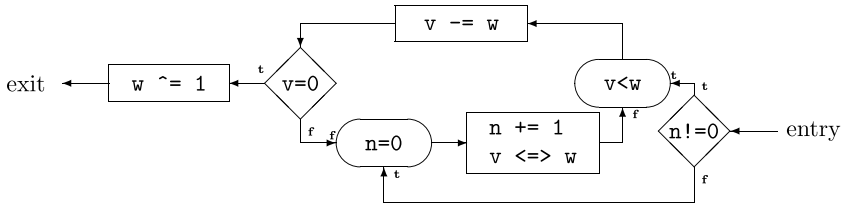
Reversible systems such as RTMs [8] and pattern-matching automata [2] are characterized by biorthogonal term rewriting systems. Let the relation $(x, l) \xrightarrow{\delta} (x', l')$ denote $\delta(x, l) = (x', l')$. We can regard this relation as (conditional) rewriting rules on configurations. The rules are biorthogonal to each other in the sense that a single rewriting rule, at most, can be applicable to a given configuration (x, l), and the previously applied rewriting rule, if any, is uniquely determined. The step-by-step reversible computation in biorthogonal automata then defines the injective computation $[\![F]\!](x) = x'$, where $(x, \mathit{entry}) \xrightarrow{\delta}{}^{*} (x', \mathit{exit})$.

Example As a running example, we consider the *Fibonacci-pair function* [25]. This function takes a positive integer n and returns the pair of the n-th and $(n+1)$-th Fibonacci numbers. For example, the first three Fibonacci pairs are $(1, 1)$, $(1, 2)$, and $(2, 3)$. The original Fibonacci function is not injective because the first and second Fibonacci numbers are both 1; thus, two different arguments map to the same result. On the other hand, the Fibonacci-pair function is injective because all pairs are different.

Fig. 4(a) shows a reversible flowchart with one loop for calculating the n-th Fibonacci pair. The domain X for this flowchart is a store that maps each of the three variables (`n`, `v`, and `w`) to 32-bit integers, which we use for concreteness. The C-like compound assignment operator `x += e` adds the value of the expression `e` to the value of the variable `x`, storing the result modulo 2^{32} in `x`. Note that the variable `x` on the left-hand side must not appear in the expression `e` on the right-hand side. Similarly, `x -= e` subtracts the value of the expression `e` from `x`, and `x ^= e` computes the bit-wise exclusive-or of the value of the expression `e` and `x`, both storing the result modulo 2^{32} in `x`. The operator `v <=> w` swaps the values of variables `v` and `w`. All of these operators compute injective step functions.



(a) An unstructured flowchart for calculating the n-th Fibonacci pair v,w.



(b) The inverse flowchart of (a) for calculating the index n of a Fibonacci pair v,w.

Fig. 4. An unstructured reversible flowchart and its inverse.

Computation starts at the *entry* of the flowchart with an initial store, in which the variable n maps to n, and the variables v and w map to zero. At the *exit*, the two variables v and w will contain the Fibonacci pair, and n will be zero unless overflow is detected. The value of n is decremented in each iteration, `n -= 1`, until it is zero, which is tested by `n=0` (thus, the assertion `n!=0` at the exit is false). An overflow is detected when v "wraps around" (test `v<w`), and the loop terminates. Assertion `v=0` at the loop entry is true only when the computation enters the loop for the first time and is false in the following iterations.

All step operations in Fig. 4(a) are invertible, and all join points carry assertions (`v=0`, `n!=0`). Thus, the flowchart is reversible. The flowchart is *unstructured* because the loop has two exits (tests `n=0` and `v<w`).

The implementations of the Fibonacci-pair function with overflow detection (that we use here to study reversible flowcharts) may be compared to the corresponding invertible functional program [25], and the extant reversible structured programs implemented by means of a reversible loop [67], and recursive reversible procedure calls [63].

Remark Generally, the compound assignment operator of addition is of the form `x += f(y)`, where f is a partial function, and y can be a tuple of variables that do not include x. The operator updates the store (x, y) by the function $g(v, w) = (v + f(w), w)$. Here, $(+)$ is injective in its first argument, and a function g of this form is called a *reversible update* [7]. All reversible updates have inverses; in this case, we have $g^{-1}(v, w) = (v - f(w), w)$. Because $(-)$ and bit-wise exclusive-or are also injective in their first arguments, the operators `x -= f(y)` and `x ^= f(y)` are also reversible updates. (See [7] for a formal definition.)

Remark Assuming properly labeled edges the atomic operations of the flowchart in Fig. 4(a) induce the following transition function on configurations:

$$
\begin{array}{llll}
\delta((n,v,w),\mathit{entry}) = ((n,v,w\ \mathit{xor}\ 1),l_1) & & \delta((n,v,w),l_5) = ((n-1,w,v),l_6) & \\
\delta((n,v,w),l_1) = ((n,v,w),l_3) & \text{if } v=0 & \delta((n,v,w),l_6) = ((n,v,w),l_2) & \text{if } n\neq 0 \\
\delta((n,v,w),l_2) = ((n,v,w),l_3) & \text{if } v\neq 0 & \delta((n,v,w),l_6) = ((n,v,w),l_8) & \text{if } n=0 \\
\delta((n,v,w),l_3) = ((n,v+w,w),l_4) & & \delta((n,v,w),l_7) = ((n,v,w),\mathit{exit}) & \text{if } n\neq 0 \\
\delta((n,v,w),l_4) = ((n,v,w),l_5) & \text{if } v<w & \delta((n,v,w),l_8) = ((n,v,w),\mathit{exit}) & \text{if } n=0 \\
\delta((n,v,w),l_4) = ((n,v,w),l_7) & \text{if } v\geq w & &
\end{array}
$$

Here, the binary operator *xor* is the bit-wise exclusive or. Atomic steps are performed as (injective) operations on the domain (n, v, w). Dispatch of control flow at the tests is performed depending on the given side conditions. Similarly, the correctness of merging the control flow at assertions is checked by the side conditions. It is easily seen that δ is injective as necessary. It is straightforward to formulate and verify the (injective) transition function for other reversible flowcharts in a similar manner.

2.2. Structured reversible flowcharts

As with classical flowcharts, reversible flowcharts allow for unstructured control flow. Needless to say, unstructured reversible flowcharts are apt to be incomprehensible reversible "spaghetti code". Structured control flow makes a program modular and easier to verify [20]. Therefore, we are interested in reversible flowcharts that are structured.

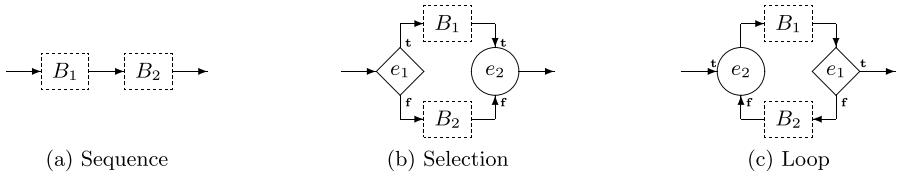


Fig. 5. Structured reversible CFOs.

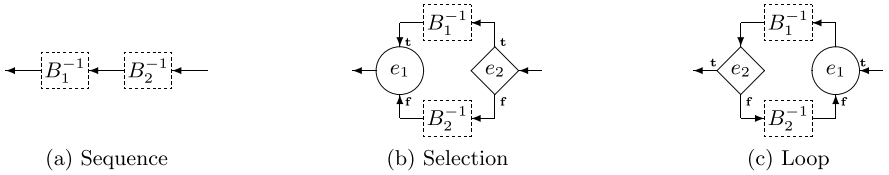


Fig. 6. Inverted structured reversible CFOs.

A *structured control-flow operator* (structured CFO) has exactly one entry and one exit, that is, it is well formed. We define three structured reversible CFOs (Fig. 5):

(a) a *sequence* of blocks B_1 and B_2,
(b) the *selection* of block B_1 or B_2, and
(c) a *loop* over blocks B_1 and B_2.

A block B_i is either an atomic step (as in Section 2.1) or one of the three structured reversible CFOs. Thus, the latter can be nested any number of times to form a complex structured flowchart. The constructs are all symmetric. A *structured reversible flowchart* is one that is constructed solely from one possibly nested block. Because of this construction, structured reversible flowcharts are always well formed. Throughout this paper, dashed boxes represent blocks.

The sequence operator is the same as that for irreversible sequences. The selection corresponds to an irreversible **if**-statement, but it has an exit assertion e_2, which makes the operator reversible. The loop is repeated as long as the test e_1 and assertion e_2 are false. The loop corresponds to a **while**–**do** loop if B_1 is empty, and to a **do**–**until** loop if B_2 is empty. In either case, the assertion at the loop entry and the test at the loop exit make the loop reversible. If a block B_i is empty (e.g., a skip step performing the identity function) in a selection or loop, the box is simply omitted from the flowchart and written as a line. The CFOs impose structure on the control flow, but this does not affect the underlying transition function as introduced for unstructured flowcharts; it directly carries over to structured flowcharts.

Example Returning to our Fibonacci-pair example, without changing its functional behavior, the *unstructured* flowchart in Fig. 4(a) can be rewritten to the *structured* reversible flowchart in Fig. 7(a), which consists of a sequence of a step and a reversible **do**–**until** loop. The two tests of the unstructured flowchart are combined, and the overflow test changed to `v > w`.

2.3. Inverse flowcharts

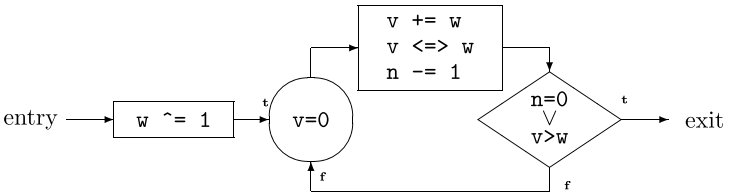
Reversible flowcharts are especially easy to invert. Many important algorithms are inverse to each other, such as printing and parsing, and encryption and decryption. Generally, flowchart inversion is difficult and even undecidable. However, inversion can be effectively realized in reversible flowcharts by means of *local inversion*, as we will see below. Moreover, inverse flowcharts are important constructs for reversible simulations [10,65].

An inverse flowchart can be generated using the following *flowchart inversion* method:

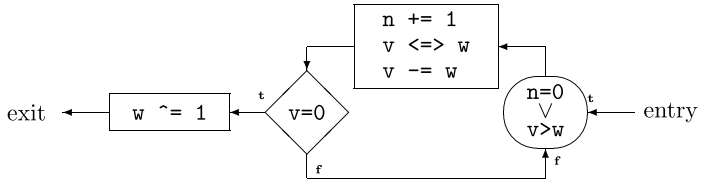
1. change the direction of each arrow and exchange entry points and exit points,
2. replace each step function a with its inverse a^{-1}, and
3. replace each test by an assertion and each assertion by a test (the predicate e remains unchanged).

Fig. 3 shows the inverse of each atomic operator from Fig. 2, where a^{-1} is the inverse step function[1] of a. Similarly, Fig. 6 shows the inverse of each CFO from Fig. 5, where B^{-1} is the inverse flowchart of B. The flowchart resulting from flowchart

[1] Here, we are not concerned with how the inverse a^{-1} is generated because this will be different for every language. In our examples in Section 4, the local inversion of steps is trivial because they directly follow the format of reversible updates [7]. In other languages, the inversion may be much more difficult.



(a) A structured flowchart for calculating the n-th Fibonacci pair v,w.



(b) The inverse flowchart of (a) for calculating the index n of a Fibonacci pair v,w.

Fig. 7. A structured flowchart and its inverse.

inversion is again reversible, and the structure is preserved. Inversion neither adds nor deletes operations or CFOs, and repeating it once more restores the original flowchart!

Such easy inversion is *not* possible for classical flowcharts, where inversion is a difficult problem (stemming from the fact that the injectivity is undecidable [44]).

Remark Let δ be the transition function (modeling the atomic operations) of a reversible flowchart F. The inverted atomic operations are modeled by the function $\eta(x,l) = \delta^{-1}(x, \mathit{flip}(l))$, where the function *flip* exchanges the entry and exit points and keeps all other labels. If we have a computation sequence $(x,l) \xrightarrow{\delta}{}^*(x',l')$, we also have a sequence of the inverse computation $(x', \mathit{flip}(l')) \xrightarrow{\eta}{}^*(x, \mathit{flip}(l))$. In particular, $[\![F]\!](x) = x'$, where $(x, \mathit{entry}) \xrightarrow{\delta}{}^*(x', \mathit{exit})$ iff $[\![F^{-1}]\!](x') = x$, where $(x', \mathit{entry}) \xrightarrow{\eta}{}^*(x', \mathit{exit})$, and where F^{-1} is the (reversible) inverse of the reversible flowchart F. Further formal discussion of inversion for a structured instance of reversible flowcharts will be given in Section 5.

Example The reversible flowcharts in Fig. 4(b) and Fig. 7(b) compute the inverse functions of Fig. 4(a) and Fig. 7(a), and vice versa. The operator `x += e` is an inverse to `x -= e`, and vice versa. The operators `x ^= e` and `v <=> w` are self-inverse. The concrete transition function for the inverse flowchart in Fig. 4(b) can easily be obtained by inverting the transition function δ for the original flowchart in Fig. 4(a), as in the above remark. We only need to swap the domain and codomain of the function, exchange the labels *entry* and *exit*, and replace the expressions $v + w$ and $n - 1$ with $v - w$ and $n + 1$.

2.4. Termination properties

Reversible flowcharts have further novel properties, e.g., with respect to termination. If the configuration space is bounded and each atomic operation terminates, well-formed reversible flowcharts always terminate. In general, this does not hold for classical flowcharts, suggesting that programming reversible flowchart languages must, in some sense, be quite different from programming classical flowchart languages. The bounded termination of reversible flowcharts is guaranteed as follows.

The trajectory of any nonterminating (forward) deterministic computation must be either infinite or *cyclic* in the computation configuration space. Thus, if the computation space is *bounded*, the trajectory must be a cycle and return somewhere to an earlier computation state. Such cyclic computation is nonterminating because once a cycle is entered, it is impossible to deterministically break out of it.

Further, if a deterministic computational trajectory *does* contain a nonterminating cycle, and if the starting state for the computation is *not* in this cycle, then there is necessarily a point in the cycle that must have more than a single predecessor; hence, the trajectory is *not backward* deterministic; that is, such a computation cannot be reversible. Thus, the trajectory of a nonterminating reversible computation on a bounded computation space must be a cycle containing the initial state.[2] Now, the starting state of the computation of a well-formed flowchart does *not* have an edge leading into it; that is, control can never return to the entry arrow after the computation starts. Therefore, the starting state cannot be part of a nonterminating cycle. This means that reversible computation trajectories for well-formed flowcharts are *always* acyclic (regardless of whether the configuration space is bounded).

[2] Note that this does not hold in irreversible computation with bounded space because a computation can have a cycle with an incoming edge without breaking determinism.

Proposition 1 *(Termination under bounded space). Well-formed reversible flowcharts with finite atomic operations and finite configuration spaces always terminate.*

Proof. Let F be a well-formed reversible flowchart and δ be its (injective) transition function on the configurations $X \times L$. Suppose there is a nonterminating computation. This implies that either an atomic operation in F does not terminate or there is an infinite sequence $(x_0, \mathit{entry}) \xrightarrow{\delta} (x_1, l_1) \xrightarrow{\delta} \cdots \xrightarrow{\delta} (x_i, l_i) \xrightarrow{\delta} \cdots$. The atomic operations are assumed to be finite; thus, the former case cannot occur, and we must have an infinite sequence. Now, control can never return to the entry because F is well-formed; therefore, $l_i \neq \mathit{entry}$ for all l_i in the sequence. Further, there are only finitely many distinct configurations because the configuration space is finite; thus, there must be some smallest indices i and j $(> i)$ in an infinite sequence such that $(x_i, l_i) = (x_j, l_j)$. However, we have $(x_{i-1}, l_{i-1}) = \delta^{-1}(x_i, l_i) = \delta^{-1}(x_j, l_j) = (x_{j-1}, l_{j-1})$ by the injectivity of δ, and this contradicts the assumption that i and j are the smallest such indices. Thus, the assumption that there is a nonterminating computation is false. □

Thus, if the configuration space for some reversible flowchart is finite, it is guaranteed to terminate. Note the generality of the statement: if we are able to prove *by any means* that the configuration space is bounded, then termination is implied. For instance, if it can be proved by a program analysis that a reversible flowchart loop can only access a bounded portion of the store, then the loop terminates, even if the configuration space is infinite.

Example In the case of the Fibonacci-pair example, the state space (the variables `n`, `v`, and `w` and the control flow information) is finite and the reversible flowchart is well formed; hence, its computation always terminates.

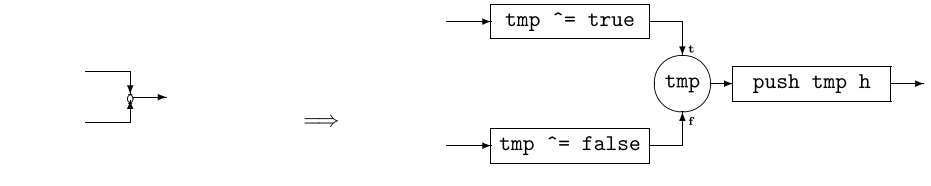
2.5. Reversibilization of classical flowcharts

Reversible flowcharts with unbounded space are *Turing-complete*, provided that the generation of *garbage data* is ignored. This can be accomplished by *reversibilizing* classical flowcharts (that is, with irreversible join points and non-injective steps), which are Turing-complete, into reversible flowcharts with the same functionality.[3] Such a reversibilization effectively changes the *type* of the computed function: if the classical flowchart F computes the function $f : X \to Y$, then the flowchart reversibilized from F will compute a function $f_h : X \to Y \times H$, where H is some domain of garbage data, necessary to guarantee that f_h is an injective function. One way to do this is to instrument the irreversible flowchart with a history stack, known as the Landauer embedding [38].

Theorem 1 *(Landauer embedding). For any well-formed classical flowchart F, a reversible flowchart F_h with the same functionality modulo the accumulation of garbage data can be constructed.*

Proof. Classical flowcharts differ from reversible flowcharts by having join points without assertions and irreversible steps (with non-injective transition functions). They are the sources of the irreversibility. The translation of a classical flowchart F into a reversible flowchart F_h proceeds by the simple substitution of irreversible joins with reversible ones, and the same for steps. The configuration space is extended with a fresh (temporary) variable, `tmp`, not occurring in F, and a *history stack* `h` to record the information required to reconstruct the previous computation state for each atomic operation performed in F. Let δ and η be the transition functions (modeling atomic operations) for F and F_h, respectively. The function δ is defined on the configurations $X \times L$, where X is a domain of data, and L is a set of labels, and thus the function η is defined on extended configurations $(X \times \mathbb{B} \times H) \times L$ where $\mathbb{B}$ is a Boolean domain for `tmp`, and H is a set of histories. (Here, the histories will be stacks.)

Without loss of generality, we may assume that F only has binary joins. A binary irreversible join point in F is reversibilized in F_h as follows:



Irreversible join point of the classical flowchart F

Reversible join point of the reversible flowchart F_h

For this to work, we require that the temporary variable `tmp` is initially and finally cleared to be false; the reversibilization respects this invariant. The functionalities of the above flowcharts are the same, except for the operations on the history

[3] Reversibilization is a transformation from irreversible computations to reversible computations.

stack. Let l_i and l_j be the labels for the incoming edges to the binary join point and associate each with the labels **t** and **f**, respectively. Further, let l_k be the label for the outgoing edge. Then, $(x, l_i) \xrightarrow{\delta} (x, l_k)$ iff $((x, t, h), l_i) \xrightarrow{\eta}{}^*((x, t, \mathit{true} :: h), l_k)$ and $(x, l_j) \xrightarrow{\delta} (x, l_k)$ iff $((x, t, h), l_j) \xrightarrow{\eta}{}^*((x, t, \mathit{false} :: h), l_k)$, where t is the value of `tmp`. Although this has value *false* in the above relations, we wrote t for clarity, as it may change its value to *true* and back to *false*, over the $\xrightarrow{\eta}{}^*$ computation.

The reversibilization of the steps is similar. For each non-injective step transition function a, we can make an injectivized step function a' such that $a'(x) = (a(x), x)$. After the step operation a', the second element x of the result is pushed to the history stack as garbage data; therefore, the reversible step operation a' has the same functionality as a modulo the garbage data. More efficient reversibilizations exist when the given step operations are of a simple form. Without loss of generality, consider the case that X is a store, and every step operation in F is an assignment $x := e$, which overwrites x with the value of the expression e (evaluated over the current store). Every step is replaced in F_h with the reversible sequence

```
push x h ; x ^= e[top h/x]
```

which moves the original value of x on top of `h` and reversibly updates x to the same value that the original irreversible assignment would have done by replacing the occurrences of x with `top h` in e, where `top` is used to view the top element of the stack. Again, the functionalities of the flowcharts before and after the translation are the same, except for the operations on the history stack: $(x, l_i) \xrightarrow{\delta} (a(x), l_j)$ iff $((x, t, h), l_i) \xrightarrow{\eta} ((a(x), t, x :: h), l_j)$, where l_i and l_j are the labels on the in- and outgoing edges of the step, respectively.

The part of the function η associated with tests neither changes the temporary variable nor the history, but behaves as δ: $(x, l_i) \xrightarrow{\delta} (x, l_j)$ iff $((x, t, h), l_i) \xrightarrow{\eta} ((x, t, h), l_j)$, where l_i is the incoming edge to the test, and l_j is an outgoing edge from the test.

All of the individual atomic operations before and after reversibilization have the same behavior, if garbage data is ignored. Thus, F_h has the same overall functionality modulo the accumulation of garbage on the history stack: by straightforward induction over $\xrightarrow{\delta}{}^*$, it follows that $[\![F]\!](x) = y$ where $(x, \mathit{entry}) \xrightarrow{\delta}{}^*(y, \mathit{exit})$ iff $[\![F_h]\!](x) = (y, h)$ where $((x, \mathit{false}, [\,]), \mathit{entry}) \xrightarrow{\delta}{}^*((y, \mathit{false}, h), \mathit{exit})$, and where $[\,]$ denotes an empty stack. □

We can now follow Bennett [8] and use inverse flowcharts to reversibly erase the computation history.

Corollary 1 *(Bennett's trick). The* input embedding $f_i : x \mapsto (f(x), x)$ *of the function f computed by a classical flowchart F can be computed by a reversible flowchart F_i.*

Proof. Given a classical flowchart F computing f, (1) obtain a reversibilized flowchart F_h computing $f_h : x \mapsto (f(x), g)$, where g is the garbage (history) induced by Theorem 1. (2) Invert F_h to obtain F_h^{-1}, which computes $f_h^{-1} : (f(x), g) \mapsto x$. (3) Construct F_i, which executes F_h, copies the values of all output variables to fresh variables, and executes F_h^{-1}. This rolls back the execution of F_h, clearing the history stack and restoring the initial values of all of the variables used by F_h. The flowchart F_i returns both the original input and the output of the executing flowchart F, and therefore, computes f_i. □

2.6. r-Turing completeness

If we choose *not* to ignore the garbage of reversibilization, the computational expressiveness of reversible flowcharts is less clear. However, they clearly cannot be fully Turing-complete because they compute only *injective* functions under this stricter semantic viewpoint. In this case, the most we can hope for is *r-Turing completeness*: being able to compute *all* injective computable functions [5]. Fortunately, this is the case.

Theorem 2 *(r-Turing completeness). If $f : X \to Y$ is an injective, computable function, then there exists a reversible flowchart F that computes f.*

Note, in particular, that the computed function of F does *not* include garbage. Now, this result follows almost directly from the above reversibilizations by a well-established technique making use of Bennett's trick and a general program inverter (such as McCarthy's generate-and-test approach); see [5,8]. We shall omit the details here, but the overview of an alternative direct construction of RTMs, which avoids reversibilization, will be shown in Section 6.2.

The classic *reversibilizations* such as Landauer's history embedding [38] or Bennett's input-saving reversible simulation method [8], are not suited to show r-Turing completeness. A reversibilized program returning garbage data along with the desired output computes a *different* function from the original program, semantically speaking; thus, these reversibilizations are not clean, in the sense of preserving semantics. For the same reason, Theorem 1 was not sufficient for showing the r-Turing completeness of reversible flowcharts. The reversibilization approach is taken in various reversible computation models (e.g., [54,48,8]) and reversible programming languages (e.g., [13,50]).

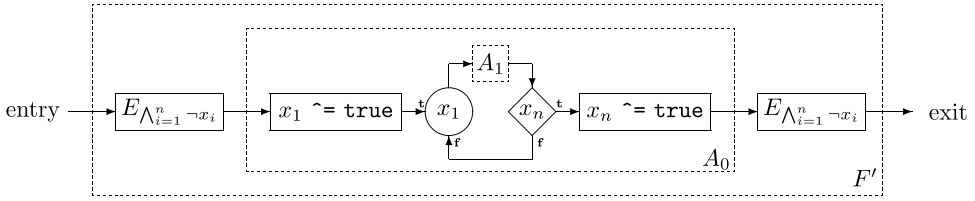


Fig. 8. Structured reversible flowchart F' with main loop A_0. (Assertion flowchart E_e in Fig. 10.)

3. The structured reversible program theorem

Nowadays, it is easy to forget that the merits of *structure* in high-level programming languages were once highly controversial [20,37]. From a purely computational viewpoint, this debate was effectively closed by the *Structured Program Theorem* [11], which showed that structured and unstructured flowcharts have the same expressive power. Thus, the useful benefits of structured high-level languages, including their increased readability and reasonability, have no computational weaknesses.

The same question is relevant to the reversible flowchart paradigm. Reversible computing is sufficiently different from standard computational models; hence, it is unclear whether results from classical (backward non-deterministic) computing carry over to the reversible paradigm, as we have seen in the previous section. Indeed, none of the classic constructions (and therefore, proofs) transform unstructured flowcharts into structured ones because they all lead to classical irreversible flowcharts. Although it may seem intuitive that structure is also "free" in reversible programming, this must be proved.

The proof is constructive and shows a method to generate a structured reversible flowchart from an arbitrary reversible flowchart with exactly the same functionality.

Theorem 3 *(Structured reversible program theorem). For any well-formed reversible flowchart F, a functionally equivalent structured reversible flowchart F', i.e., $[\![F]\!] = [\![F']\!]$, with at most a single reversible loop can be constructed.*

Proof. Let F be a well-formed reversible flowchart and n be the number of edges in F. Let the interpretation domain of the transition functions and predicates of F be X. Below, we construct a functionally equivalent *structured reversible flowchart* A_0 over a trivial extension of X with n (fresh) Boolean variables $x_1, \ldots, x_n$ (Fig. 8). For this, we uniquely label every edge in F by l_i $(1 \leq i \leq n)$. If F contains no step operation, it trivially satisfies the theorem; thus, we exclude this case. Without loss of generality, we label the entry edge l_1, the exit edge l_n, and the two incoming edges of any assertion l_i and l_{i+1}.

The main idea of the proof is as follows. Each node and its incoming edges are translated into a structured equivalent. A main loop simulates the control flow of F one node at a time by keeping track of the edge that the execution follows in F. This *edge state* is modeled by adding a fresh Boolean variable x_i for each edge l_i to the original domain X. Thus, if $[\![F]\!] : X \rightharpoonup X$, then $[\![A_0]\!] : X \times Bool^n \rightharpoonup X \times Bool^n$, where the initial and final values of each x_i must be **false**, written as $\mathbf{false}^n = (\mathbf{false}, \ldots, \mathbf{false}) \in Bool^n$.[4] The edge state is called i if x_i is **true**, and all other values of x_j's are **false**. Thus, if the control flow in F is at the edge l_i, then the edge state in the loop of A_0 should be i.

The edge state is changed from i to j using an injective transition function $P_{i,j} : Bool^n \to Bool^n$, which takes a Boolean array of length n, where element i is **true** and the others are **false**, and returns an array, where element j is **true** and the others are **false**.[5] In F, this corresponds to moving control from the edge l_i to the edge l_j.

First, generate the main loop in Fig. 8 for the entry edge l_1 and the exit edge l_n. The flowchart A_0 contains the reversible loop between an initial step and a final step. The initial step sets x_1 to **true** and keeps the others **false**, where the compound assignment ^= is exclusive-or-equal. If the test x_n is **true**, the loop ends and x_n is set to **false**; otherwise, it continues. The path back to assertion x_1 is a skip operation.

Then, recursively build the structured reversible flowcharts A_i $(0 < i < n)$ by the rules in Fig. 9, where the atomic operation with the incoming edge l_i (or l_i and l_{i+1}) in the left column is translated into the flowchart A_i in the right column. A dashed box A_j inside A_i stands for a well-formed flowchart that simulates the execution of control flow over exactly one node along some edge along edge l_k with $k \geq j$. A node with a bold exterior outline is a node that appeared in the original unstructured flowchart. We now detail the three possible cases.

1. A *step* with function a is executed in the translated flowchart only if the edge state is i. The state is then changed to j, and x_j becomes **true**, simulating the control flow over the step. If the state is not i, then A_{i+1} is entered. By the unique numbering of the edges, the state cannot be j after executing A_{i+1}; thus, an assertion of x_j is sufficient to distinguish between the two possibilities.

[4] The x_i variables are analogous to the use of *ancillae* in reversible and quantum circuits.

[5] This function can be regarded as passing a token from x_i to x_j.

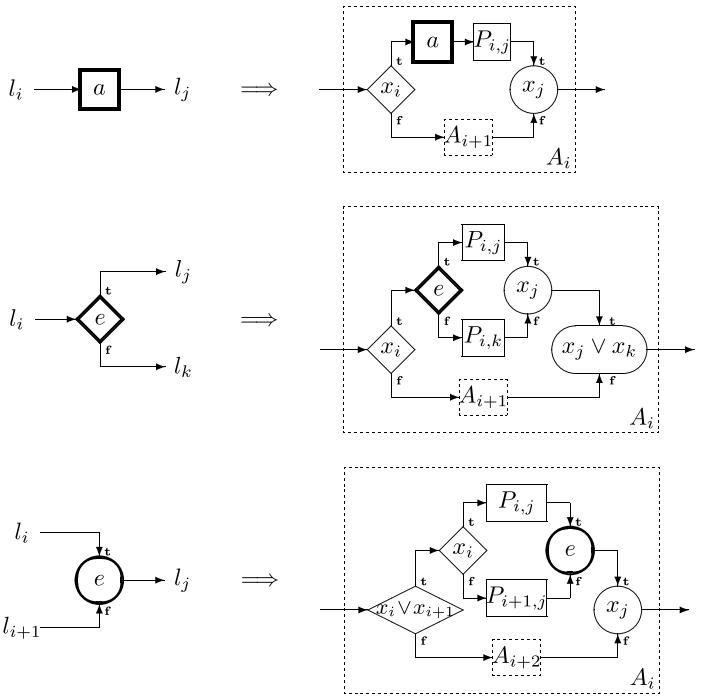


Fig. 9. Unstructured operations with an incoming edge labeled by l_i (and l_{i+1}) transformed into structured reversible flowcharts A_i.

2. A *test* e is similar to a step. $P_{i,j}$ and $P_{i,k}$ change the variables x_i, x_j, and x_k. Thus, depending on e, the edge state is set to either j or k, which can be distinguished by the test x_j. By an argument analogous to the step case, the assertion $x_j \vee x_k$ is sufficient to distinguish the test case from A_{i+1}.
3. The edges l_i and l_{i+1} of an *assertion* e are translated simultaneously. Therefore, A_i does not contain A_{i+1}. x_i is used to distinguish states i and $i+1$; both change to j. The predicate e differentiates between the two possibilities. Similar to the above two cases, x_j is sufficient to distinguish this from A_{i+2}. (The assertion where l_i labels the **true** edge and l_{i+1} the **false** edge is shown; the opposite case is analogous.)

The assertion case is symmetrical to the test case; by renaming labels, the transformed assertion can be seen to be structurally identical to the inverse flowchart of the transformed test. Because a test and an assertion are inverses of each other, it makes sense (yet, it is striking) that their translations should be, and indeed are, inverses of each other.

For well-formedness, we insert into A_{n+1} a block $E_{\texttt{false}}$ (Fig. 10 with e instantiated to `false`), in which the computation abnormally halts, even though it is never reached in the computation. The total structured reversible flowchart A_0 generated by the rules in Figs. 8 and 9 thus simulates the execution of every step, test, and assertion in flowchart F.

Recall that we defined $[\![F]\!](x) = y$ by $(x, l_1) \xrightarrow{\delta}{}^{*} (y, l_n)$, where $\delta : X \times L \rightharpoonup X \times L$ is the transition function (modeling the atomic operations in F) on configurations. Given the body of the main loop of the translated flowchart, A_1 (Fig. 8), and the transition function of the original flowchart, δ, a simple induction shows that we have the correspondence

$$(x, (x_1, \ldots, x_n)) \xrightarrow{[\![A_1]\!]} (y, (x'_1, \ldots, x'_n)) \iff (x, l_i) \xrightarrow{\delta} (y, l_m) \tag{1}$$

where $x_i = x'_m = \textbf{true}$, and the other x_j and x'_j elements are **false**. Note that relation $x \xrightarrow{f} y$ denotes $f(x) = y$. Repeated application of this correspondence (i.e., the computation of the main loop A_0) gives us the computation sequence from entry to exit:

$$(x, (\textbf{true}, \textbf{false}, \ldots, \textbf{false})) \xrightarrow{[\![A_1]\!]}{}^{*} (y, (\textbf{false}, \ldots, \textbf{false}, \textbf{true})) \iff (x, l_1) \xrightarrow{\delta}{}^{*} (y, l_n) \tag{2}$$

This directly gives us $[\![A_0]\!](x, \textbf{false}^n) = ([\![F]\!](x), \textbf{false}^n)$ if $[\![F]\!](x)$ is defined.

Finally, to restrict the domain of $[\![A_0]\!]$ to $X \times \{\textbf{false}^n\} \simeq X \times \mathbb{1} \simeq X$, we require that all values of x_i are **false** at its entry and exit, i.e., $\bigwedge_{i=1}^{n} \neg x_i$. Here, $\mathbb{1}$ represents a unit type that allows only one value. The extra flowchart $E_{\bigwedge_{i=1}^{n} \neg x_i}$ is the identity if all values of x_i's are **false**; otherwise, it abnormally halts. Here, the flowchart E_e is defined in Fig. 10. If forward computation chooses the false edge, it halts at the constant assertion **true**, and if backward computation chooses the false edge, it abnormally halts if e is not false. The flowchart $E_{\bigwedge_{i=1}^{n} \neg x_i}$ is well formed and contains no loops. That is, F' is well formed and contains a single loop, and we have that F' is functionally equivalent to F, $[\![F']\!] = [\![F]\!]$. □

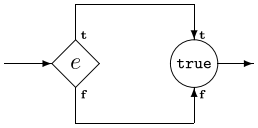


Fig. 10. Assertion flowchart E_e, parameterized with the test e. Both forward and backward computations abnormally halt if the test e does not hold.

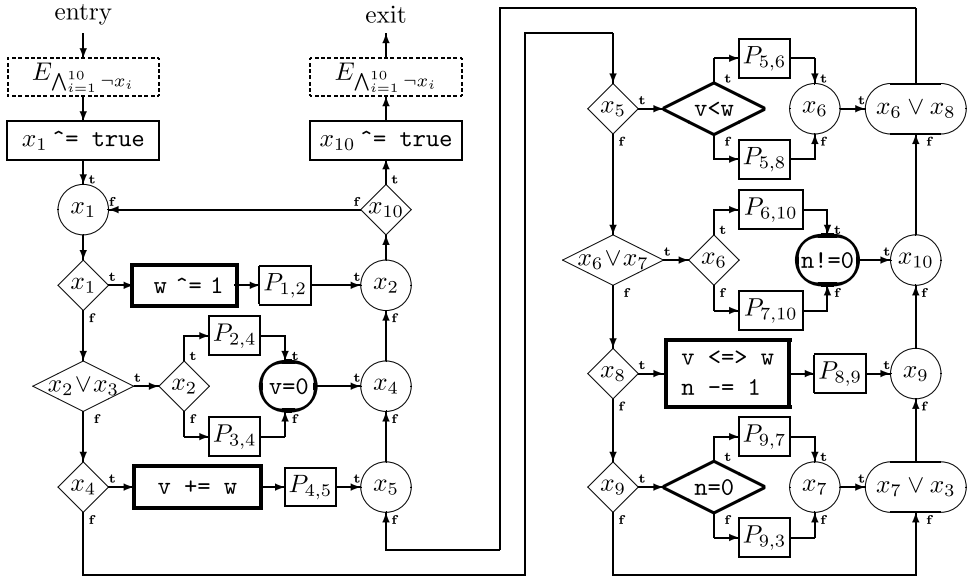


Fig. 11. The structured flowchart generated from the unstructured flowchart in Fig. 4(a) (Fibonacci pairs with overflow checking) using the translator in Fig. 8 and Fig. 9.

Thus, from a computational viewpoint, structured and unstructured flowcharts are equally powerful, even in the reversible computing paradigm. The proof was inspired by Cooper's global proof sketch [14], but was more involved. It requires assertions at all join points in the translated version of every step operation. Moreover, the administrative operations $P_{i,j}$ need to be locally invertible transition functions (destructive assignments are not allowed), and the assertion operator itself must be translated into a structured reversible flowchart. Essentially, the translation can use none of the classical structured conditional and loop operators because they are all inherently irreversible. Somewhat ironically, the translation is oblivious to any use of structure in the original flowchart, which is thus not preserved. However, it should be noted that neither Jacopini's local proof [11], which is the original proof of the Structured Program Theorem and preserves the original structure, nor Cooper's proof sketch [14], is applicable to this theorem.

The translation is optimal in the following sense. In each of the main loop iterations of A_0, control over exactly one node in F will be simulated, and the corresponding transition function of a step or expression evaluation of a test or an assertion is performed only once. Thus, the overhead in the translation is mainly due to the evaluation of the Boolean-edge state variables. This means that the translation does *not* have an impact on the asymptotic complexity because the number of edges in the original flowchart is constant with respect to the input and output.[6] This is a considerable improvement over our original translation in [64], where every assertion expression in the original flowchart F would be evaluated twice in *every* main loop iteration, regardless of which node was being simulated (and each test condition was evaluated twice when simulated), potentially at an asymptotic cost (although not impacting computability). The memory overhead does not add an asymptotic cost. The extra information added by x_i's can be represented by $\lceil \log_2 n \rceil$ bits, which is independent of the size of the input.

Example We show the result of translating the unstructured flowchart in Fig. 4(a) to the corresponding structured flowchart in Fig. 11. For readability, we align the flowchart in a ladder format. The operations with bold exterior outlines appear in the original flowchart. It is easy to see that no operation was duplicated. Note that the number of ladder steps (not counting the top and bottom) is equal to the number of nodes (steps, tests, and assertions) in the original unstructured flowchart in Fig. 4(a).

[6] This overhead, which is linear in the number of edges for each loop, can be further improved by nesting the translation in a balanced tree structure. As a result, the time overhead becomes at worst logarithmic in the number of edges, i.e., $O(\log n)$.

$$p ::= b \qquad b ::= a \mid \texttt{if}\ e\ \texttt{then}\ b\ \texttt{else}\ b\ \texttt{fi}\ e \mid b\ b \mid \texttt{from}\ e\ \texttt{do}\ b\ \texttt{loop}\ b\ \texttt{until}\ e$$

(a) Structured reversible language *SRL*.

$$q ::= d^+ \qquad d ::= l\texttt{:}\ k\ a^*\ j \qquad k ::= \texttt{from}\ l \mid \texttt{fi}\ e\ \texttt{from}\ l\ \texttt{else}\ l \mid \texttt{entry} \qquad j ::= \texttt{goto}\ l \mid \texttt{if}\ e\ \texttt{goto}\ l\ \texttt{else}\ l \mid \texttt{exit}$$

(b) Unstructured reversible language *RL*.

$$a ::= x \oplus\texttt{=}\ e \mid x\texttt{[}e\texttt{]} \oplus\texttt{=}\ e \mid \texttt{push}\ x\ x \mid \texttt{pop}\ x\ x \mid \texttt{skip}$$

$$e ::= c \mid x \mid x\texttt{[}e\texttt{]} \mid e \otimes e \mid \texttt{top}\ x \mid \texttt{empty}\ x \qquad c ::= \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{4294967295} \qquad \otimes ::= \oplus \mid \texttt{*} \mid \texttt{/} \mid \cdots \qquad \oplus ::= \texttt{+} \mid \texttt{-} \mid \texttt{\^{}}$$

(c) Reversible step operations and expressions.

$$\begin{array}{rlllll} \mathit{SRL}: & p \in \text{SRL} & b \in \text{Blk} & & & \\ \mathit{RL}: & q \in \text{RL} & d \in \text{RLblk} & j \in \text{Jump} & k \in \text{From} & l \in \text{Label} \\ \mathit{SRL}, \mathit{RL}: & a \in \text{Step} & e \in \text{Exp} & c \in \text{Const} & x \in \text{Var} & \oplus, \otimes \in \text{Op} \end{array}$$

(d) Syntax domains of *SRL* and *RL*.

Fig. 12. Syntax of the two reversible flowchart languages.

4. Two reversible flowchart programming languages

We now demonstrate the usefulness of the diagrammatic considerations by presenting two concrete programming languages that form a natural core subset for reversible imperative programming languages. We present complete syntax and semantics for both languages. One is a simple low-level *assembler-style language* with unstructured jumps (e.g., for a reversible microprocessor), and the other is a high-level *block-structured reversible language* with structured statements and data structures. Later, a translator from the high-level language to the low-level language is given (Section 5), complementing the Structured Reversible Program Theorem of the previous section. We begin by choosing a textual representation of the reversible control flow and defining the atomic step operations for the languages (for simplicity these are shared by both languages). Even though the languages presented here are small, they are later shown to be r-Turing complete (Section 6). The reader is invited to extend our languages with further constructs; therefore, the description here can serve as a starting point for the design and formalization of more sophisticated reversible imperative programming languages at various abstraction levels of the computation stack.

The reversible language Janus [40,63,67] is an example of a structured reversible language. The reversible CFOs can be viewed as generalizations of the corresponding operators in Gries' invertible language [29] and Frank's language R [23]. Examples of low-level languages are PISA, an assembly language for the reversible microprocessor Pendulum [23,60], and Bob, a reversible instruction set architecture [57]. These languages can all be modeled by reversible flowcharts.

4.1. Representing structured and unstructured reversible control flow

A key difference from conventional languages is that the reversible control flow has to be represented textually in our structured and unstructured languages. The block-structured language is the simpler case, as there are no explicit jumps and the required assertions can be made part of the syntax of the language CFOs (conditional, loop). In the case of the unstructured language, which has a flat sequence of labeled blocks, the incoming control flow has to be made explicit at the entry of each block by adding a *come-from* construct (from, fi). The remaining control-flow constructs (goto, if) are familiar from conventional assembly languages. We briefly describe the syntax of the languages; their semantics will be given afterwards.

A program in the *structured reversible language* (*SRL*) consists of a block of recursively nested syntactic forms. A block is either a step operation, a sequence of blocks, a conditional (`if` e_1 `then` b_1 `else` b_2 `fi` e_2), or a loop (`from` e_1 `do` b_1 `loop` b_2 `until` e_2). The syntax is shown in Fig. 12(a). The blocks textually represent the reversible structured CFOs of Fig. 5. The step operations a and expressions e will be detailed below.

Remark *SRL* has minimal syntax and features, and provides a skeleton for high-level reversible programming languages. For example, Janus can be regarded as an extension to *SRL* with additional features that *SRL* does not have, such as parameterized procedures, local variables, and direct access to inverse semantics. We note that *SRL* programs always terminate

on bounded data, whereas Janus programs may consume unbounded memory for the call stack of procedures, even if the variable space is bounded.

A program in the *unstructured reversible language* (*RL*) is a nonempty sequence of basic (*RL*-)blocks. The syntax of *RL* is shown in Fig. 12(b). A block consists of four components: a unique label l, a come-from assertion k, a sequence of step operations a^*, and a jump j. A come-from assertion may be unconditional (`from` l), which means that control always comes from the block l; conditional (`fi` e `from` l_1 `else` l_2), which means that *control comes from* the block l_1 when the predicate e is true and from block l_2 otherwise; or the program entry (`entry`). As usual, a jump may be unconditional (`goto` l), conditional (`if` e `goto` l_1 `else` l_2), or the program exit (`exit`). Well-formed unstructured programs contain exactly one `entry` and one `exit`.

4.2. Designing reversible step operations

A *step operation* is the smallest unit of execution in a reversible flowchart, and a step, which changes stores, is either an assignment or a stack operation in our languages. For simplicity, both languages use the same step operations and thus share the corresponding syntax domains (cf. the bottom line of Fig. 12(d).) In general, any *reversible update* [7] can be added as a step operation. A step operation in our languages can update scalar variables, one-dimensional arrays of bounded size, and stacks of potentially unbounded size. Thus, the considerations in Section 2.4 regarding termination under bounded space do not apply to our languages in general. The syntax of step operations is shown in Fig. 12(c).

An *assignment* is an update of either a scalar variable or an array variable. We use C-like compound assignment operators (x `^=` e, x `+=` e, x `-=` e, x`[`e_l`]` `^=` e, x`[`e_l`]` `+=` e, and x`[`e_l`]` `-=` e), where the variable x on the left-hand side must not occur in the expression e on the right-hand side nor in the array index expression e_l. This simple syntactic constraint ensures that the assignments are reversible.

A *stack operation* is either `push` x_1 x_2 or `pop` x_1 x_2. The push operation moves the value of the first argument x_1 to the top of the stack x_2, and zero-clears variable x_1. The pop operation, the inverse of `push`, moves the top of the stack x_2 into the first argument x_1 which must be zero-cleared. This ensures the reversibility of `push` and `pop`. The top element of a stack x is accessed by `top` x. The predicate `empty` x tests whether the stack x is empty.

A *skip operation* `skip` does nothing. It can be regarded as an abbreviation of x `^= 0`.

4.3. Defining the reversible language semantics

Structural operational semantics for the two reversible flowchart languages is given below, making use of some of the properties of these languages on the semantics level. We shall see how reversibility is reflected at the formal semantics level and what it takes to claim the reversibility of a language.

The semantic domains used are shown in Fig. 15. A *store* σ is a mapping from left values to values. *Left values* are locations in memory, designated by variables or indexed array variables on the left-hand side of assignments. We write $\sigma[x \mapsto v]$ for the store obtained from σ by replacing its contents in x by v, and we write $\sigma \backslash x$ for the store obtained from σ by removing the binding for x. A *value* is either an integer or in a stack of integers, where we shall use $\mathbb{Z}_{32}$, the set of 32-bit integers for concreteness, i.e., $\{0, \ldots, 2^{32} - 1\}$. (This choice is only to ensure the finiteness of the base set of integer values.) Boolean values are represented by integers; zero is used for false and nonzero for true. A label in the semantic domain is a label from the syntax domain or one of the internal labels *entry* or *exit*. The latter are only used for pointing to and from the (imaginary) entry and exit *RL*-blocks of an unstructured program, respectively.

Generally, when designing a reversible language, we need to justify that the language is truly reversible. Usually, this happens by showing that the language satisfies a more precise claim of what reversibility means in the language. Although this depends on the concrete language, we can provide some intuition of how this can work semantically. The execution of a program construct t of the syntax domain X (down to a reasonable level of granularity[7]) is regarded as a transformation from the store σ to σ', formulated by the judgment

$$\sigma \vdash_X t \Rightarrow \sigma' . \tag{3}$$

The claim that the execution of t is reversible is formulated as both the forward and backward determinism of the transformation:

Definition 1 *(Execution of t is reversible).* A well-formed executable language construct t of the syntax domain X is called *reversible*, iff

$$\forall \sigma_1 . \forall \sigma_2 . \forall \sigma . \quad \sigma \vdash_X t \Rightarrow \sigma_1 \ \wedge \ \sigma \vdash_X t \Rightarrow \sigma_2 \ \implies \ \sigma_1 = \sigma_2 , \tag{4}$$

$$\forall \sigma_1 . \forall \sigma_2 . \forall \sigma . \quad \sigma_1 \vdash_X t \Rightarrow \sigma \ \wedge \ \sigma_2 \vdash_X t \Rightarrow \sigma \ \implies \ \sigma_1 = \sigma_2 . \tag{5}$$

[7] For example, executable constructs bottom out at the *step* level.

$$\frac{\sigma \backslash x \vdash_{exp} e \Rightarrow v}{\sigma \vdash_{step} x \oplus\!= e \Rightarrow \sigma[x \mapsto [\![\oplus]\!](\sigma(x), v)]} \textsc{AsnVar}$$

$$\frac{\sigma \backslash x[v_l] \vdash_{exp} e_l \Rightarrow v_l \quad \sigma \backslash x[v_l] \vdash_{exp} e \Rightarrow v}{\sigma \vdash_{step} x[e_l] \oplus\!= e \Rightarrow \sigma[x[v_l] \mapsto [\![\oplus]\!](\sigma(x[v_l]), v)]} \textsc{AsnArr}$$

$$\frac{}{\sigma \vdash_{step} \texttt{skip} \Rightarrow \sigma} \textsc{Skip}$$

$$\frac{}{\sigma \vdash_{step} \texttt{push}\, x_1\, x_2 \Rightarrow \sigma[x_2 \mapsto \sigma(x_1) :: \sigma(x_2)][x_1 \mapsto 0]} \textsc{Push} \qquad \frac{\sigma' \vdash_{step} \texttt{push}\, x_1\, x_2 \Rightarrow \sigma}{\sigma \vdash_{step} \texttt{pop}\, x_1\, x_2 \Rightarrow \sigma'} \textsc{Pop}$$

Fig. 13. Execution of step operations.

$$\frac{[\![c]\!] = v}{\sigma \vdash_{exp} c \Rightarrow v} \textsc{Con} \qquad \frac{}{\sigma \vdash_{exp} x \Rightarrow \sigma(x)} \textsc{Var} \qquad \frac{\sigma \vdash_{exp} e \Rightarrow v}{\sigma \vdash_{exp} x[e] \Rightarrow \sigma(x[v])} \textsc{Arr}$$

$$\frac{\sigma \vdash_{exp} e_1 \Rightarrow v_1 \quad \sigma \vdash_{exp} e_2 \Rightarrow v_2 \quad [\![\otimes]\!](v_1, v_2) = v}{\sigma \vdash_{exp} e_1 \otimes e_2 \Rightarrow v} \textsc{Bop} \qquad \frac{\sigma(x) = v :: s}{\sigma \vdash_{exp} \texttt{top}\, x \Rightarrow v} \textsc{Top}$$

$$\frac{\sigma(x) = nil}{\sigma \vdash_{exp} \texttt{empty}\, x \Rightarrow 1} \textsc{EmptyTrue} \qquad \frac{\sigma(x) = v :: s}{\sigma \vdash_{exp} \texttt{empty}\, x \Rightarrow 0} \textsc{EmptyFalse}$$

Fig. 14. Evaluation of expressions.

$$\begin{aligned} \sigma \in \text{Store} &= \text{Lvalue} \rightharpoonup \text{Value} \\ u \in \text{Lvalue} &= \text{Var} \cup \{x[v] \mid x \in \text{Var}, v \in \text{Value}\} \\ v \in \text{Value} &= \mathbb{Z}_{32} \cup \text{Svalue} \\ s \in \text{Svalue} &= \{nil\} \cup \{v :: s \mid v \in \mathbb{Z}_{32}, s \in \text{Svalue}\} \\ l \in \text{Label}' &= \text{Label} \cup \{entry, exit\} \end{aligned}$$

Fig. 15. Semantic values of the reversible flowchart languages.

Because the atomic constructs in reversible languages must satisfy these claims, they effectively serve as design requirements for reversible programming languages. Below, we give the semantics for both low-level and high-level language constructs that satisfy these claims.

The store transformation of a *step operation* is defined by a judgment

$$\sigma \vdash_{step} a \Rightarrow \sigma' \tag{6}$$

where σ and σ' are stores, and a is a step operation (Fig. 13). A step operation realizes an elementary executable operation (Fig. 2(a)) by an *injective* relation. We will show how, in a straightforward manner, we can deal with three non-standard issues that arise in this context: ensuring the reversibility of assignments, using inverse semantics, and coping with irreversible expression evaluation.

The syntactic requirement that a variable that occurs on the left-hand side of an assignment must not occur on its right-hand side is reflected in the semantics rules by *hiding* the variable from the evaluation of the right-hand side. In general, an assignment may not be reversible if the left-hand side variable occurs on the right-hand side (e.g., `x -= x`). In rules AsnVar and AsnArr, the variable x occurring on the left-hand side of an assignment is removed from the store ($\sigma \backslash x$) in which the expression e is evaluated. Hiding x from σ ensures that the execution of assignments in which x occurs in e is undefined.

The semantics of operations that are *inverses* of each other can be conveniently defined by swapping the stores σ and σ' in the relation:

$$\frac{\sigma' \vdash_X \mathcal{I}_X[\![t]\!] \Rightarrow \sigma}{\sigma \vdash_X t \Rightarrow \sigma'} \tag{7}$$

in which $\mathcal{I}_X$ is an inverter on the syntax domain X of t. This design principle is demonstrated in the *SRL* language; as we will see in Fig. 20, the stack operations `push` and `pop` are inverses of each other, and `pop` is simply defined in terms of `push` in rule Pop in Fig. 13.

The evaluation of an expression (e.g., on the right-hand side of an assignment) is defined as usual by a judgment $\sigma \vdash_{exp} e \Rightarrow v$ (Fig. 14), where σ is a store, e is an expression, and v is a value. We assume that the binary operator $\otimes$ stays within the semantic domain $\mathbb{Z}_{32}$, by performing *modular arithmetic* on $\mathbb{Z}_{32}$ (rule Bop). In contrast to step operations, the relation on expressions is not injective: different stores may induce the same result value, such as $\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\} \vdash_{exp} \mathtt{x + y} \Rightarrow 3$ and $\{\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 1\} \vdash_{exp} \mathtt{x + y} \Rightarrow 3$. Although σ uniquely determines v for a given e (forward deterministic), v does not uniquely define σ (backward nondeterministic). This backward nondeterminism does not harm the injectivity

$$\frac{\sigma \vdash_{step} a \Rightarrow \sigma'}{\sigma \vdash_{blk} a \Rightarrow \sigma'}\ \textsc{Step} \qquad \frac{\sigma \vdash_{blk} b_1 \Rightarrow \sigma'' \quad \sigma'' \vdash_{blk} b_2 \Rightarrow \sigma'}{\sigma \vdash_{blk} b_1\ b_2 \Rightarrow \sigma'}\ \textsc{Seq}$$

$$\frac{\sigma \vdash_{exp} e_1 \not\Rightarrow 0 \quad \sigma \vdash_{blk} b_1 \Rightarrow \sigma' \quad \sigma' \vdash_{exp} e_2 \not\Rightarrow 0}{\sigma \vdash_{blk} \texttt{if}\ e_1\ \texttt{then}\ b_1\ \texttt{else}\ b_2\ \texttt{fi}\ e_2 \Rightarrow \sigma'}\ \textsc{IfTrue}$$

$$\frac{\sigma \vdash_{exp} e_1 \Rightarrow 0 \quad \sigma \vdash_{blk} b_2 \Rightarrow \sigma' \quad \sigma' \vdash_{exp} e_2 \Rightarrow 0}{\sigma \vdash_{blk} \texttt{if}\ e_1\ \texttt{then}\ b_1\ \texttt{else}\ b_2\ \texttt{fi}\ e_2 \Rightarrow \sigma'}\ \textsc{IfFalse}$$

$$\frac{\sigma \vdash_{exp} e_1 \not\Rightarrow 0 \ \ \sigma \vdash_{blk} b_1 \Rightarrow \sigma' \ \ \sigma' \vdash_{loop} (e_1, b_1, b_2, e_2) \Rightarrow \sigma''}{\sigma \vdash_{blk} \texttt{from}\ e_1\ \texttt{do}\ b_1\ \texttt{loop}\ b_2\ \texttt{until}\ e_2 \Rightarrow \sigma''}\ \textsc{Loop}$$

$$\frac{\sigma \vdash_{exp} e_2 \not\Rightarrow 0}{\sigma \vdash_{loop} (e_1, b_1, b_2, e_2) \Rightarrow \sigma}\ \textsc{LoopEnd} \qquad \frac{\begin{array}{c}\sigma \vdash_{exp} e_2 \Rightarrow 0 \ \ \sigma \vdash_{blk} b_2 \Rightarrow \sigma' \\ \sigma' \vdash_{exp} e_1 \Rightarrow 0 \ \ \sigma' \vdash_{blk} b_1 \Rightarrow \sigma'' \\ \sigma'' \vdash_{loop} (e_1, b_1, b_2, e_2) \Rightarrow \sigma'''\end{array}}{\sigma \vdash_{loop} (e_1, b_1, b_2, e_2) \Rightarrow \sigma'''}\ \textsc{LoopRep}$$

$$\frac{\sigma \vdash_{blk} p \Rightarrow \sigma'}{\sigma \vdash_{SRL} p \Rightarrow \sigma'}\ \textsc{SRL-Program}$$

Fig. 16. Operational semantics of the structured reversible language *SRL*.

of the step operations, and thus, the reversibility of the language. This is because the rules AsnVar and AsnArr that use expression evaluation each define a reversible update [7] of the store with $[\![\oplus]\!]$ being injective in its first argument.

It is desirable to prove the reversibility of all atomic operations in reversible programming languages. Here, we briefly outline the proof for the reversibility of step operations.

Proof for $X = step$. We break down the different cases. A skip operation `skip` relates the input and output stores as an identity relation by Skip and is thus reversible. In the case where the step operation is a reversible assignment for variables, we assume $\sigma \vdash_{step} x \oplus\!= e \Rightarrow \sigma'$. By AsnVar, we uniquely have v such that $\sigma \backslash x \vdash_{exp} e \Rightarrow v$. The input and output stores are related by $\sigma' = \sigma[x \mapsto [\![\oplus]\!](\sigma(x), v)]$, and σ determines σ' because $[\![\oplus]\!]$ is a function. Conversely, we show that σ' determines σ. We uniquely have v' such that $\sigma' \backslash x \vdash_{exp} e \Rightarrow v'$. Because $\sigma \backslash x = \sigma' \backslash x$, we have $v = v'$. Because $[\![\oplus]\!]$ is injective in its first argument, there is a function g that is the inverse of $\lambda w.\ [\![\oplus]\!](w, v)$, and we can then uniquely determine σ from σ' by the equation $\sigma = \sigma'[x \mapsto g(\sigma'(x))]$. When the given step operation is a reversible assignment for indexed arrays, we need to take into consideration the forward determinism of the evaluation of index expressions, but the same argument as in the previous case essentially holds. The stores updated by `push` satisfying the judgment $\sigma \vdash_{step} \texttt{push}\ x_1\ x_2 \Rightarrow \sigma'$ are related by $\sigma = \sigma'[x_1 \mapsto v_1, x_2 \mapsto v_2]$, where $v_1 :: v_2 = \sigma'(x_2)$. Because pop operations are the inverse of push operations, the reversibility of pop operations directly follows from the fact that store updates by push operations are forward deterministic. □

4.4. SRL semantics

The complete semantic definition of the structured reversible language *SRL* consists of the evaluation of expressions (Fig. 14), the execution of step operations (Fig. 13), and the rules defining the control flow of a structured reversible program (Fig. 16).

Rules apply to blocks on the basis of their forms. The rule Step applies to (atomic) step operations. The rules Seq, IfTrue and IfFalse, and Loop apply to structured reversible CFOs, i.e., sequences, conditionals, and loops, respectively. These rules realize the behavior of the structured reversible flowchart operators in Fig. 5. Note that $\sigma \vdash_{exp} e \not\Rightarrow v$ does *not* mean that e does not evaluate to a value at all, but rather that e evaluates to some value that is not v.

For simplicity, the Loop, LoopEnd, and LoopRep rules are asymmetric, represented in right-recursive form. The right-recursive form is efficient for forward execution. One can define symmetric representations as well, and it is easy to check that the symmetric and asymmetric forms relate exactly the same input and output stores when they appear in the premise of the Loop rule in the proof tree. The right (left, resp.) recursive loop rules are locally forward (backward, resp.) deterministic but not locally backward (forward, resp.) deterministic. However, this nondeterminism does not harm the reversibility of block executions. This is because the assertion e_1 and the test e_2 have nonzero values only at the entry and exit of the loop. The rest of the semantic rules are reversible, e.g., locally forward and backward deterministic.

The claim that the semantics of *SRL* is reversible is formalized by Definition 1 (with *SRL* as the syntactic domain X and t ranging over the programs p). This can be proven by structural induction on derivations. For the induction, we need to prove the reversibility of the execution of steps, loops (in the particular form used in blocks), and blocks, which can also be formulated as instances of Definition 1. Examples of such formalization in reversible language design can be found in [63,67], where [67] adopts symmetric loop rules and [63] adopts asymmetric loop rules.

$$\frac{}{\sigma \vdash_{steps} \cdot \Rightarrow \sigma}\ \textsc{StepList}_1 \qquad \frac{\sigma \vdash_{step} a \Rightarrow \sigma' \quad \sigma' \vdash_{steps} al \Rightarrow \sigma''}{\sigma \vdash_{steps} a\ al \Rightarrow \sigma''}\ \textsc{StepList}_2$$

$$\frac{}{\sigma \vdash_{from} \texttt{entry} \Rightarrow entry}\ \textsc{Entry} \qquad \frac{}{\sigma \vdash_{jump} \texttt{exit} \Rightarrow exit}\ \textsc{Exit}$$

$$\frac{}{\sigma \vdash_{from} \texttt{from}\ l \Rightarrow l}\ \textsc{From} \qquad \frac{}{\sigma \vdash_{jump} \texttt{goto}\ l \Rightarrow l}\ \textsc{Jump}$$

$$\frac{\sigma \vdash_{exp} e \not\Rightarrow 0}{\sigma \vdash_{from} \texttt{fi}\ e\ \texttt{from}\ l_1\ \texttt{goto}\ l_2 \Rightarrow l_1}\ \textsc{CFrom}_1 \qquad \frac{\sigma \vdash_{exp} e \not\Rightarrow 0}{\sigma \vdash_{jump} \texttt{if}\ e\ \texttt{goto}\ l_1\ \texttt{else}\ l_2 \Rightarrow l_1}\ \textsc{CJump}_1$$

$$\frac{\sigma \vdash_{exp} e \Rightarrow 0}{\sigma \vdash_{from} \texttt{fi}\ e\ \texttt{from}\ l_1\ \texttt{goto}\ l_2 \Rightarrow l_2}\ \textsc{CFrom}_2 \qquad \frac{\sigma \vdash_{exp} e \Rightarrow 0}{\sigma \vdash_{jump} \texttt{if}\ e\ \texttt{goto}\ l_1\ \texttt{else}\ l_2 \Rightarrow l_2}\ \textsc{CJump}_2$$

$$\frac{\Gamma_q(l_2) = k\ al\ j \quad \sigma \vdash_{from} k \Rightarrow l_1 \quad \sigma \vdash_{steps} al \Rightarrow \sigma' \quad \sigma' \vdash_{jump} j \Rightarrow l_3}{(\sigma, (l_1, l_2)) \Rightarrow_q (\sigma', (l_2, l_3))}\ \textsc{Block}$$

$$\frac{\Gamma_q(l_1) = \texttt{entry}\ al_1\ j \qquad \Gamma_q(l_2) = k\ al_2\ \texttt{exit} \qquad (\sigma, (entry, l_1)) \Rightarrow_q^* (\sigma', (l_2, exit))}{\sigma \vdash_{RL} q \Rightarrow \sigma'}\ \textsc{RL-Program}$$

Fig. 17. Operational semantics of the unstructured reversible language *RL*.

4.5. RL semantics

The semantic definition of the unstructured reversible language *RL* shares the semantic rules for expression evaluation (Fig. 14) and step execution (Fig. 13) with the structured reversible language *SRL*. However, the formalization of the control flow in an *RL* program is quite different from an *SRL* program (Fig. 17).

In rules StepList$_1$ and StepList$_2$, a sequence of step operations is decomposed into single steps, and each step is executed by the step execution rules in Fig. 13. Similar to the Loop rules in *SRL* semantics, the right-recursive StepList$_2$ rule can be formulated in left-recursive and symmetric forms that are functionally equivalent. Because the rules in these three forms are all forward and backward deterministic, StepList$_2$ is reversible.

In a transition from one block to another block in the Block rule, a pair of labels in the configuration $(\sigma, (l_1, l_2))$ designates the target block l_2 ("goto") as well as the source block l_1 ("come from"). The function Γ_q : Label $\rightharpoonup$ From $\times$ Step* $\times$ Jump associates a label l in a program q with the body of the block labeled l.

In the RL-Program rule, program execution is defined by a small-step semantics relating the initial store and (internal) entry label *entry* with the final store and (internal) exit label *exit*. Because well-formed *RL* programs have exactly one `entry` and one `exit`, the use of *entry* and *exit* in the semantics is unambiguous. This makes the use of the transitive closure transition ($\Rightarrow_q^*$) deterministic.

The claim that *RL* semantics is reversible for well-formed *RL* programs is formalized by Definition 1 (with $X = RL$, $t = q$). This can be proven by induction on the structure of *RL* programs.

5. Inversion and translation of reversible flowchart languages

To complete the presentation of the two concrete flowchart languages, we here show how the diagrammatic inversion of flowcharts considered in Section 2.3 carries over to actual languages and how the languages can be translated to each other.

The *program inverters* given in the first part of this section provide a simple core around which one can build program inverters for more sophisticated languages. (This provides further evidence for the versatility of the generic flowchart model.) An example of program inversion will be given afterwards in Section 6.1.

In the second part, clean *translators* between the two flowchart languages are presented. The translator from *SRL* to *RL* can be viewed as a template for compilers that generate reversible machine code from a block-structured language. This translation complements the structured reversible program theorem in Section 3, which translates flowchart programs in the opposite direction. The translator from *RL* to *SRL* can be viewed as an instance of the translator used in the proof of Theorem 3. It shows that the structured reversible program theorem holds for the concrete programming languages *SRL* and *RL*.

A clean *self-interpreter* for a structured reversible language has been presented elsewhere [67]. Taken together, this demonstrates that the familiar tools frequently used and necessary for standard (irreversible) programming practice are also available in the reversible flowchart setting.

5.1. Flowchart program inverters

The inversion of program text is analogous to the inversion of flowchart diagrams (Section 2.3). Thus, the formalization of the program inverters for *SRL* and *RL* is straightforward. In particular, local inversion is sufficient to invert any block of any program written in these two languages. The two program inverters differ only in the inversion of the program structure owing to the different textual representation of the control flow. Both inverters preserve the well-formedness of flowcharts.

$$
\begin{array}{ll}
\mathcal{I}_{SRL}[\![b_1\ b_2]\!] & = \mathcal{I}_{SRL}[\![b_2]\!]\ \mathcal{I}_{SRL}[\![b_1]\!] \\
\mathcal{I}_{SRL}[\![\texttt{if}\ e_1\ \texttt{then}\ b_1\ \texttt{else}\ b_2\ \texttt{fi}\ e_2]\!] & = \texttt{if}\ e_2\ \texttt{then}\ \mathcal{I}_{SRL}[\![b_1]\!]\ \texttt{else}\ \mathcal{I}_{SRL}[\![b_2]\!]\ \texttt{fi}\ e_1 \\
\mathcal{I}_{SRL}[\![\texttt{from}\ e_1\ \texttt{do}\ b_1\ \texttt{loop}\ b_2\ \texttt{until}\ e_2]\!] & = \texttt{from}\ e_2\ \texttt{do}\ \mathcal{I}_{SRL}[\![b_1]\!]\ \texttt{loop}\ \mathcal{I}_{SRL}[\![b_2]\!]\ \texttt{until}\ e_1 \\
\mathcal{I}_{SRL}[\![a]\!] & = \mathcal{I}_{step}[\![a]\!]
\end{array}
$$

Fig. 18. Program inverter $\mathcal{I}_{SRL}$ for the structured reversible language *SRL*.

$$
\begin{array}{ll}
\mathcal{I}_{RL}[\![d^+]\!] & = \mathcal{I}_{RL}[\![d]\!]^+ \\
\mathcal{I}_{RL}[\![l:\ k\ a^*\ j]\!] & = l:\ \mathcal{I}_{jump}[\![j]\!]\ rev(\mathcal{I}_{step}[\![a]\!]^*)\ \mathcal{I}_{from}[\![k]\!] \\
 & \\
\mathcal{I}_{jump}[\![\texttt{goto}\ l]\!] & = \texttt{from}\ l \\
\mathcal{I}_{jump}[\![\texttt{if}\ e\ \texttt{goto}\ l_1\ \texttt{else}\ l_2]\!] & = \texttt{fi}\ e\ \texttt{from}\ l_1\ \texttt{else}\ l_2 \\
\mathcal{I}_{jump}[\![\texttt{exit}]\!] & = \texttt{entry} \\
 & \\
\mathcal{I}_{from}[\![k]\!] & = \mathcal{I}^{-1}_{jump}[\![k]\!]
\end{array}
$$

Fig. 19. Program inverter $\mathcal{I}_{RL}$ for the unstructured reversible language *RL*.

$$
\begin{array}{lll}
\mathcal{I}_{step}[\![x \oplus\!= e]\!] = x \ominus\!= e & \text{where } \ominus = \mathcal{I}_{op}[\![\oplus]\!] & \mathcal{I}_{op}[\![+]\!] = - \\
\mathcal{I}_{step}[\![x[e_1] \oplus\!= e_2]\!] = x[e_1] \ominus\!= e_2 & \text{where } \ominus = \mathcal{I}_{op}[\![\oplus]\!] & \mathcal{I}_{op}[\![-]\!] = + \\
\mathcal{I}_{step}[\![\texttt{push}\ x_1\ x_2]\!] = \texttt{pop}\ x_1\ x_2 & & \mathcal{I}_{op}[\![\hat{}]\!] = \hat{} \\
\mathcal{I}_{step}[\![\texttt{pop}\ x_1\ x_2]\!] = \texttt{push}\ x_1\ x_2 & & \\
\mathcal{I}_{step}[\![\texttt{skip}]\!] = \texttt{skip} & &
\end{array}
$$

Fig. 20. The step-operation inverter common to both program inverters.

The *program inverter* $\mathcal{I}_{SRL}$ for the structured flowchart language *SRL* performs a recursive descent over the block structure of a program (Fig. 18). A sequence of blocks is reversed, and each block is inverted individually. In selection blocks and loop blocks, the assertions and tests are exchanged, and the constituent blocks are inverted. The inversion of step operations is common to both inverters and described below.

The *program inverter* $\mathcal{I}_{RL}$ for the unstructured flowchart language *RL* inverts each block of a program individually (Fig. 19). A sequence of blocks d^+ is inverted by inverting each block individually. Recall that the order of the blocks in an unstructured program does not matter. Each block d is then inverted by changing the direction of the incoming and outgoing control flow and inverting the sequence of step operations it contains. The control flow is inverted by exchanging the two flow operators k and j in each block and turning all `from`-constructs into `goto`-constructs, `fi`-constructs into `if`-constructs, and the program entry into the program exit by $\mathcal{I}_{from}$, and vice versa by $\mathcal{I}_{jump}$. We note that $\mathcal{I}_{from}$ and $\mathcal{I}_{jump}$ are inverses of each other, $\mathcal{I}_{from} = \mathcal{I}^{-1}_{jump}$; thus, it is sufficient to define just one of them. This shows again how the properties of the object language are reflected on the meta-level. Each step operation is individually inverted by $\mathcal{I}_{step}$, and the sequence of the inverted step operations is reversed by *rev*.

The step-operation inverter $\mathcal{I}_{step}$ is common to both program inverters (Fig. 20). It uses the fact that the assignment operators `+=` and `-=` are inverses of each other and that `^=` is self-inverse. The stack operations `push` and `pop` are also inverses of each other. The `skip` operation, which does nothing, is self-inverse. The step operators are closed under inversion.

Both program inverters are total and correct. In contrast, these properties generally do not hold simultaneously for the inversion of irreversible programs [1], and the inversion of programs usually requires a global program analysis (e.g., [26,27,32,45,51]). *Local invertibility* is one of the desired properties of automatic program inversion [25], and it allows program inversion to be performed on-the-fly while a program is running, which means that, e.g., fast inversion of reversible machine code can be supported in hardware by a microprocessor [23,57].

The correctness of the program inverters can be formulated as follows:

Proposition 2 *(Correctness of inverter $\mathcal{I}_X$). For any well-formed executable program construct t of the syntax domain X, we have*

$$\forall\sigma\,.\,\forall\sigma'\,.\,\sigma \vdash_X t \Rightarrow \sigma' \iff \sigma' \vdash_X \mathcal{I}_X[\![t]\!] \Rightarrow \sigma\,. \tag{8}$$

The inverters $\mathcal{I}_{SRL}$ and $\mathcal{I}_{RL}$ both satisfy this claim, which can be proven by structural induction.

Proof for $\mathcal{I}_{SRL}$ and $\mathcal{I}_{RL}$. ($\Longrightarrow$) Most of the cases are straightforward. For *RL*, an induction hypothesis is only required in $\mathcal{I}_{RL}$ for the list of blocks (d^+) and step operations (a^*); all others are simple base cases. Because the syntax and semantics of step operations is the same in both languages, the cases of $\mathcal{I}_{step}$ also hold for *SRL*. Although, in general the conclusions of semantic rules do not necessarily uniquely determine their premises, those of *SRL* and *RL* do. Therefore, given a sequence of blocks $b_1\ b_2$ such that $\sigma \vdash_{blk} b_1\ b_2 \Rightarrow \sigma'$, we have $\sigma \vdash_{blk} b_1 \Rightarrow \sigma''$ and $\sigma'' \vdash_{blk} b_2 \Rightarrow \sigma$ by Seq in Fig. 16. By the induction hypothesis, we have $\sigma'' \vdash_{blk} \mathcal{I}_{RL}[\![b_1]\!] \Rightarrow \sigma$ and $\sigma' \vdash_{blk} \mathcal{I}_{RL}[\![b_2]\!] \Rightarrow \sigma''$. Reversing the order of the judgments and applying Seq, we have the intended judgment $\sigma' \vdash_{blk} \mathcal{I}_{RL}[\![b_1\ b_2]\!] \Rightarrow \sigma$ by the definition of $\mathcal{I}_{RL}$. The case of a selection block is similar but requires a case analysis according to the values of the test and the assertion.

$\mathcal{T}_{SRL}[\![b]\!] =$

l_0 : entry
  goto l_1
$\mathcal{T}[\![b]\!]\ (l_0, l_1, l_2, l_3)$
l_3 : from l_2
  exit

where l_0, l_1, l_2, l_3 are fresh

$\mathcal{T}[\![b_1\ b_2]\!]\ (l_0, l_1, l_4, l_5) =$

$\mathcal{T}[\![b_1]\!]\ (l_0, l_1, l_2, l_3)$
$\mathcal{T}[\![b_2]\!]\ (l_2, l_3, l_4, l_5)$

where l_2, l_3 are fresh

$\mathcal{T}[\![a]\!]\ (l_0, l_1, l_2, l_3) =$

l_1 : from l_0
  a
  goto l_2
l_2 : from l_1
  goto l_3

$\mathcal{T}[\![\ \texttt{if}\ e_1\ \texttt{then}\ b_1\ \texttt{else}\ b_2\ \texttt{fi}\ e_2\ ]\!]\ (l_0, l_1, l_6, l_7) =$

l_1 : from l_0
  if e_1 goto l_2 else l_4
$\mathcal{T}[\![b_1]\!]\ (l_1, l_2, l_3, l_6)$
$\mathcal{T}[\![b_2]\!]\ (l_1, l_4, l_5, l_6)$
l_6 : fi e_2 from l_3 else l_5
  goto l_7

where l_2, l_3, l_4, l_5 are fresh

$\mathcal{T}[\![\ \texttt{from}\ e_1\ \texttt{do}\ b_1\ \texttt{loop}\ b_2\ \texttt{until}\ e_2\ ]\!]\ (l_0, l_1, l_4, l_7) =$

l_1 : fi e_1 from l_0 else l_6
  goto l_2
$\mathcal{T}[\![b_1]\!]\ (l_1, l_2, l_3, l_4)$
$\mathcal{T}[\![b_2]\!]\ (l_4, l_5, l_6, l_1)$
l_4 : from l_3
  if e_2 goto l_7 else l_5

where l_2, l_3, l_5, l_6 are fresh

Fig. 21. Clean translator $\mathcal{T}_{SRL}$ from structured *SRL* to unstructured *RL*.

The case of a loop block is more involved. If $\sigma_1 \vdash_{blk} \texttt{from}\ e_1\ \texttt{do}\ b_1\ \texttt{loop}\ b_2\ \texttt{until}\ e_2 \Rightarrow \sigma'$ holds, the premises of its unique right-recursive proof tree constructed by Loop, LoopEnd, and LoopRep in Fig. 16 have in their leaves from left to right, the judgment for entry $\sigma_1 \vdash_{exp} e_1 \Rightarrow 0$; the judgments for the store update by the do-branch $\sigma_1 \vdash_{blk} b_1 \Rightarrow \sigma_1'$; the four judgments for each loop $\sigma_i' \vdash_{exp} e_2 \Rightarrow 0$, $\sigma_i' \vdash_{blk} b_2 \Rightarrow \sigma_{i+1}$, $\sigma_{i+1} \vdash_{exp} e_1 \Rightarrow 0$, and $\sigma_{i+1} \vdash_{blk} b_1 \Rightarrow \sigma_{i+1}'$ for $1 \leq i \leq n-1$; and the judgment for exit $\sigma_n' \vdash_{exp} e_2 \Rightarrow 0$ for some n. Note that there is no other way to construct a proof tree for the judgment of the loop. Because the store is σ_n' at the exit of the loop, we have $\sigma' = \sigma_n'$. By the induction hypothesis, we have the judgments for the inverted blocks $\sigma_i' \vdash_{blk} \mathcal{I}_{SRL}[\![b_1]\!] \Rightarrow \sigma_i$ for $1 \leq i \leq n$ and $\sigma_{i+1} \vdash_{blk} \mathcal{I}_{SRL}[\![b_2]\!] \Rightarrow \sigma_i'$ for $1 \leq i \leq n-1$. If we reverse the sequence of the judgments and replace the judgments for blocks with those for their inverted blocks, the right-recursive proof tree constructed by the Loop, LoopEnd, and LoopRep rules, has the root $\sigma' \vdash_{blk} \texttt{from}\ e_2\ \texttt{do}\ \ \mathcal{I}_{SRL}[\![b_1]\!]\ \ \texttt{loop}\ \ \mathcal{I}_{SRL}[\![b_2]\!]\ \ \texttt{until}\ e_1 \Rightarrow \sigma_1$. By the definition of $\mathcal{I}_{SRL}$, this leads to the conclusion $\sigma' \vdash_{blk} \mathcal{I}_{SRL}[\![\texttt{from}\ e_1\ \texttt{do}\ \ b_1\ \ \texttt{loop}\ \ b_2\ \ \texttt{until}\ e_2]\!] \Rightarrow \sigma_1$.

($\Longleftarrow$) The derivation steps in the arguments above also hold in the other direction. Hence, by Eq. (8) we have the intended implications for both $X = RL$ and $X = SRL$. □

5.2. Clean flowchart translators

We can regard the use of structured CFOs as syntactic sugar in an unstructured program, and with this approach an *SRL*-program is easily translated into an *RL*-program. The *clean translator* $\mathcal{T}_{SRL}$ in Fig. 21 converts a program from *SRL* into *RL*. The translator generates the entry and exit blocks of the *RL* target program, and translates the single *SRL* main block into a sequence of *RL*-blocks using the recursive-descent block translator $\mathcal{T}$. The block translator takes an *SRL*-block b and a quadruple of labels that specifies the four labels of the sequence of *RL*-blocks to be generated from block b as an input: a from-label that defines where the control flow should come from, entry and exit labels for the block sequence, and a goto-label where the control flow should go to.

In the case where the *SRL*-block b consists of two blocks $b_1\ b_2$, each block is individually translated with two fresh labels (l_2, l_3) that wire the control flow between the two block sequences generated for b_1 and b_2. A block that is just an atomic step operation a is translated into a block l_1 receiving control from l_0 and a block l_2 passing control to l_3, and a itself is placed into one of the two blocks. The syntax of atomic step operations is identical in both languages; therefore, a can be placed into the target program without further modification.

The translation of conditionals and loops resembles each other, except that the wiring of the control flow between their blocks b_1 and b_2 differs. The two blocks are individually translated by a recursive call to $\mathcal{T}$ with four fresh labels wiring the internal control flow (recall the control flow in Fig. 5(b, c)).

This small translator from a structured language to an unstructured one can be regarded as a skeleton translation from which one can design and implement an actual compiler for a high-level structured reversible language. One such instance is the sophisticated clean translator from Janus to PISA [3], which satisfies Proposition 3. Another is the work on a compiler with heap management for the reversible functional language RFUN [6,47]. In a translator to reversible assembler code, one will perform further transformations such as generating instruction sequences for expressions in step operations, conditionals, and loops.

A correct translator should preserve the semantics of the programs. The following proposition states that the translated code preserves the exact relationships between the interpretation of the input store σ and the output store σ'. This means that no garbage data is generated; thus, the translator is *clean*. Instantiating $X = SRL$ and $Y = RL$, Eq. (9) in the proposition states the correctness of $\mathcal{T}_{SRL}$.

Proposition 3 *(Correctness of translator $\mathcal{T}_X$ from X to Y). For any well-formed X-program p, we have*

$$\forall\sigma\,.\,\forall\sigma'\,.\,\sigma \vdash_X p \Rightarrow \sigma' \iff \sigma \vdash_Y \mathcal{T}_X[\![p]\!] \Rightarrow \sigma'\,. \tag{9}$$

Proof for $\mathcal{T}_{SRL}$. The correctness of Proposition 3, instantiated for $\mathcal{T}_{SRL}$, directly follows the correctness of the translator $\mathcal{T}$ above: For any *SRL*-block b, we have

$$\sigma \vdash_{blk} b \Rightarrow \sigma' \iff (\sigma, (l_0, l_1)) \Rightarrow^*_{\mathcal{T}[\![b]\!]\ (l_0,l_1,l_2,l_3)} (\sigma', (l_2, l_3))\,. \tag{10}$$

($\Longrightarrow$) We use structural induction on b. For each case, we assume $\sigma \vdash_{blk} b \Rightarrow \sigma'$. When b is a step operation a, we have $\sigma \vdash_{step} a \Rightarrow \sigma'$ by the Step rule. By the From, StepList$_1$, StepList$_2$, and Jump rules, this judgment is equal to $(\sigma, (l_0, l_1)) \Rightarrow_{l_1:\texttt{from}\ l_0\ a\ \texttt{goto}\ l_2} (\sigma', (l_1, l_2))$, and we obviously have $(\sigma', (l_1, l_2)) \Rightarrow_{l_2:\texttt{from}\ l_1\ \texttt{goto}\ l_3} (\sigma', (l_2, l_3))$. Because the blocks l_1 : `from` l_0 a `goto` l_2 and l_2 : `from` l_1 `goto` l_3 are included in $\mathcal{T}[\![a]\!]\ (l_0, l_1, l_2, l_3)$, simple rewriting leads to the right-hand side of Eq. (10). In what follows in the small step rules we replace a set of blocks with its superset without further explanation. When b is a sequence of blocks $b_1\ b_2$, let q be $\mathcal{T}[\![b]\!]\ (l_0, l_1, l_4, l_5)$, where the labels l_2 and l_3 are fixed, as shown in Fig. 21. The induction hypothesis gives us

$$\sigma \vdash_{blk} b_1 \Rightarrow \sigma'' \Longleftrightarrow (\sigma, (l_0, l_1)) \Rightarrow^*_{\mathcal{T}[\![b_1]\!]\ (l_0,l_1,l_2,l_3)} (\sigma'', (l_2, l_3)) \tag{11}$$

$$\sigma'' \vdash_{blk} b_2 \Rightarrow \sigma' \Longleftrightarrow (\sigma'', (l_2, l_3)) \Rightarrow^*_{\mathcal{T}[\![b_2]\!]\ (l_2,l_3,l_4,l_5)} (\sigma', (l_4, l_5)) \tag{12}$$

By Seq, the conclusion $\sigma \vdash_{blk} b_1\ b_2 \Rightarrow \sigma'$ is equivalent to the premises $\sigma \vdash_{blk} b_1 \Rightarrow \sigma''$ and $\sigma'' \vdash_{blk} b_2 \Rightarrow \sigma'$. Similarly, by reversibility the reflexive transitive closure $(\sigma, (l_0, l_1)) \Rightarrow^*_q (\sigma', (l_4, l_5))$ is uniquely divided into $(\sigma, (l_0, l_1)) \Rightarrow^*_q (\sigma'', (l_2, l_3))$ and $(\sigma'', (l_2, l_3)) \Rightarrow^*_q (\sigma', (l_4, l_5))$ for each σ''. These equivalences combined with the above equations (Eq. (11) and (12)) imply Eq. (10). When b is conditional `if` e_1 `then` b_1 `else` b_2 `fi` e_2, let q be $\mathcal{T}[\![b]\!]\ (l_0, l_1, l_6, l_7)$, where the labels l_2, l_3, l_4, and l_5 are fixed, as shown in Fig. 21, and we further divide the case into two. If the test on the entry is true, i.e., $\sigma \vdash_{exp} e_1 \not\Rightarrow 0$ holds, the assertion at the exit should be true, and the then branch should be executed, i.e., $\sigma \vdash_{exp} e_2 \not\Rightarrow 0$ and $\sigma \vdash_{blk} b_1 \Rightarrow \sigma'$ hold. The induction hypothesis leads the latter judgment to $(\sigma, (l_1, l_2)) \Rightarrow^*_{\mathcal{T}[\![b_1]\!]\ (l_1,l_2,l_3,l_6)} (\sigma', (l_3, l_6))$. Because both expressions e_1 and e_2 are evaluated to nonzero values, we have $(\sigma, (l_0, l_1)) \Rightarrow_q (\sigma, (l_1, l_2))$ and $(\sigma', (l_3, l_6)) \Rightarrow_q (\sigma', (l_6, l_7))$. By merging the small step rules, we have $(\sigma, (l_0, l_1)) \Rightarrow^*_q (\sigma', (l_6, l_7))$. Similarly, the case for $\sigma \vdash_{exp} e_1 \not\Rightarrow 0$ holds.

An interesting case is when b is the loop `from` e_1 `do` b_1 `loop` b_2 `until` e_2. Let q be $\mathcal{T}[\![b]\!]\ (l_0, l_1, l_4, l_7)$, where the labels l_2, l_3, l_5, and l_6 are fixed as shown in Fig. 21. We assert a local lemma:

$$\forall\sigma\,.\,\forall\sigma'\,.\,\sigma \vdash_{loop} (e_1, b_1, b_2, e_2) \Rightarrow \sigma' \iff (\sigma, (l_3, l_4)) \Rightarrow^*_q (\sigma', (l_4, l_7)) \tag{13}$$

We use rule induction on the proof trees, rooted to the judgments on both sides. When $\sigma \vdash_{exp} e_2 \not\Rightarrow 0$, Eq. (13) immediately holds. When $\sigma \vdash_{exp} e_2 \Rightarrow 0$, we have

$$\begin{aligned}
&\sigma \vdash_{loop} (e_1, b_1, b_2, e_2) \Rightarrow \sigma' \\
&\quad\Longleftrightarrow\quad \sigma \vdash_{exp} e_2 \Rightarrow 0 \ \wedge\ \sigma \vdash_{blk} b_2 \Rightarrow \sigma'' \ \wedge\ \sigma'' \vdash_{exp} e_1 \Rightarrow 0 \ \wedge \\
&\qquad\quad \sigma'' \vdash_{blk} b_1 \Rightarrow \sigma''' \ \wedge\ \sigma''' \vdash_{loop} (e_1, b_1, b_2, e_2) \Rightarrow \sigma' \qquad (\because \text{LoopRep rule}) \\
&\quad\Longleftrightarrow\quad (\sigma, (l_3, l_4)) \Rightarrow_q (\sigma, (l_4, l_5)) \ \wedge\ (\sigma, (l_4, l_5)) \Rightarrow_q (\sigma'', (l_6, l_1)) \ \wedge\ (\sigma'', (l_6, l_1)) \Rightarrow_q (\sigma'', (l_1, l_2)) \ \wedge \\
&\qquad\quad (\sigma'', (l_1, l_2)) \Rightarrow_q (\sigma''', (l_3, l_4)) \ \wedge\ (\sigma''', (l_3, l_4)) \Rightarrow^*_q (\sigma', (l_4, l_7)) \\
&\qquad\qquad (\because \text{Induction hypothesis on } b \text{ (Eq. (10)) and the proof trees (Eq. (13))}) \\
&\quad\Longleftrightarrow\quad (\sigma, (l_3, l_4)) \Rightarrow^*_q (\sigma', (l_4, l_7)) \qquad (\because \text{Simple rewriting})
\end{aligned}$$

Therefore, Eq. (13) holds. By using this equation, a derivation similar to the previous cases leads to Eq. (10).

($\Longleftarrow$) The derivation steps in the arguments above also hold in the other direction. □

Conversely, we define a *clean translator* $\mathcal{T}_{RL}$ (Fig. 22) from any well-formed unstructured *RL* program q to a well-formed structured *SRL* program that is functionally equivalent to q, i.e., $[\![\mathcal{T}_{RL}[\![q]\!]]\!]^{SRL} = [\![q]\!]^{RL}$, except for the auxiliary Boolean variables $x^k_{j,i}$ $(i = 1, \ldots, n+1,\ j = 0, \ldots, n,\ k = 0, 1, 2)$ that keep track of the edge states. The main idea of the translation is the same as in the proof of Theorem 3. Each iteration of the main loop simulates a come-from assertion, a sequence of steps, or a jump of a block in q. Without loss of generality, we assume that q has n atomic operations uniquely labeled $l_1, \ldots, l_n$, and

$$
\begin{aligned}
\mathcal{T}_{RL}[\![q]\!] &= \texttt{if } \textstyle\bigwedge_{i=0}^{n}\bigwedge_{j=1}^{n+1}\bigwedge_{k=0}^{2}\neg x_{i,j}^{k} \texttt{ fi true}\\
&\quad\ x_{0,1}^{0} \texttt{ ^= true}\\
&\quad\ \texttt{from } x_{0,1}^{0} \texttt{ do}\\
&\quad\quad \mathcal{T}_{blks}[\![q]\!]\\
&\quad\ \texttt{until } x_{n,n+1}^{0}\\
&\quad\ x_{n,n+1}^{0} \texttt{ ^= true}\\
&\quad\ \texttt{if } \textstyle\bigwedge_{i=1}^{n}\bigwedge_{j=0}^{n+1}\bigwedge_{k=0}^{2}\neg x_{i,j}^{k} \texttt{ fi true}\\
\mathcal{T}_{blks}[\![d_1\ d_2\ \cdots\ d_n]\!] &= \mathcal{T}_{blk}[\![d_1]\!](\mathcal{T}_{blk}[\![d_2]\!](\cdots(\mathcal{T}_{blk}[\![d_n]\!](E_{\texttt{false}}))\cdots))\\
\mathcal{T}_{blk}[\![l_i : k\ a^*\ j]\!](F) &= \mathcal{T}_{flow}[\![k]\!](i, \mathcal{T}_{steps}[\![a^*]\!](i, \mathcal{T}_{flow}[\![j]\!](i, F)))\\
\mathcal{T}_{flow}[\![\begin{smallmatrix}\texttt{fi } e \texttt{ from } l_j\\ \texttt{else } l_k\end{smallmatrix}]\!](i, F) &= \begin{array}{l}\texttt{if } x_{j,i}^{0} \vee x_{k,i}^{0} \texttt{ then}\\ \quad \texttt{if } x_{j,i}^{0} \texttt{ then } P_{j,i} \texttt{ else } P_{k,i} \texttt{ fi } e\\ \texttt{else } F \texttt{ fi } x_{i,i}^{1}\end{array}\\
\mathcal{T}_{flow}[\![\begin{smallmatrix}\texttt{if } e \texttt{ goto } l_j\\ \texttt{else } l_k\end{smallmatrix}]\!](i, F) &= \begin{array}{l}\texttt{if } x_{i,i}^{2} \texttt{ then}\\ \quad \texttt{if } e \texttt{ then } R_{i,j} \texttt{ else } R_{i,k} \texttt{ fi } x_{i,j}^{0}\\ \texttt{else } F \texttt{ fi } x_{i,j}^{0} \vee x_{i,k}^{0}\end{array}\\
\mathcal{T}_{flow}[\![\texttt{from } l_j]\!](i, F) &= \texttt{if } x_{j,i}^{0} \texttt{ then } P_{j,i} \texttt{ else } F \texttt{ fi } x_{i,i}^{1}\\
\mathcal{T}_{flow}[\![\texttt{goto } l_j]\!](i, F) &= \texttt{if } x_{i,i}^{2} \texttt{ then } R_{i,j} \texttt{ else } F \texttt{ fi } x_{i,j}^{0}\\
\mathcal{T}_{flow}[\![\texttt{entry}]\!](1, F) &= \texttt{if } x_{0,1}^{0} \texttt{ then } P_{0,1} \texttt{ else } F \texttt{ fi } x_{1,1}^{1}\\
\mathcal{T}_{flow}[\![\texttt{exit}]\!](n, F) &= \texttt{if } x_{n,n}^{2} \texttt{ then } R_{n,n+1} \texttt{ else } F \texttt{ fi } x_{n,n+1}^{0}\\
\mathcal{T}_{steps}[\![a^*]\!](i, F) &= \texttt{if } x_{i,i}^{1} \texttt{ then } a^*;\ P_{i,i} \texttt{ else } F \texttt{ fi } x_{i,i}^{2}
\end{aligned}
$$

Fig. 22. Clean translator $\mathcal{T}_{RL}$ from unstructured *RL* to structured *SRL*.

that the `entry` and `exit` blocks are labeled l_1 and l_n, respectively. Further, for convenience, we assume two (imaginary) blocks l_0 and l_{n+1}, one from which l_1 is reached and one to which l_n jumps, respectively.

The *edge state* of the program q is modeled by adding fresh Boolean variables $x_{j,i}^{k}$ to the translated program. The simulation is said to be in the edge state $x_{j,i}^{k}$ if it is **true**, and all others are **false**. The edge state $x_{j,i}^{k}$ tells us that control comes from the block j to the current block i, and that k is the internal state of simulating the current block. The internal state indicates which of the three components of the current block i is to be simulated next: $x_{j,i}^{0}$ indicates that it is the come-from assertion, and $x_{i,i}^{1}$ and $x_{i,i}^{2}$ indicate that it is the sequence and jump, respectively. (After simulating the come-from assertion of i, the come-from block is changed from j to i, which will be the next come-from block once it is successfully simulated.) An edge state $x_{j,i}^{k}$ is updated by two injective *SRL* macros, $P_{j,j'}$ and $R_{i,i'}$, where the former updates j to j' and the latter updates i to i', and both advance k to $k' = (k+1) \bmod 3$. Thus, the edge state after $P_{j,j'}$ is $x_{j',i}^{k'}$ and that after $R_{i,i'}$ is $x_{j,i'}^{k'}$, respectively.

The translator $\mathcal{T}_{RL}$ generates a program having a palindromic structure. The first and last conditionals are the assertion flowchart E_e (Fig. 10), where $e = \bigwedge_{i=0}^{n}\bigwedge_{j=1}^{n+1}\bigwedge_{k=0}^{2}\neg x_{i,j}^{k}$. This asserts that all auxiliary variables are **false** before and after simulation. The body of the main loop is a case dispatch on the edge state and has a "ladder structure" similar to the construction in the proof of Theorem 3 (cf. the structured flowchart in Fig. 11). At the bottom layer $E_{\texttt{false}}$, i.e., `if false fi true`, the computation always abnormally halts. Starting from the initial edge state $x_{0,1}^{0}$, the main loop iterates until the final edge state $x_{n,n+1}^{0}$.

Given the corresponding component of a block i, and the already generated code F of the case dispatch, the translators $\mathcal{T}_{flow}$ and $\mathcal{T}_{steps}$ generate an *SRL*-block that simulates the component and updates the edge state. The cases for the steps a^*, conditional jumps `fi` e `from` l_j `else` l_k, and conditional come-from assertions `if` e `goto` l_j `else` l_k have the structures similar to the corresponding translated flowcharts in Fig. 9. Each pair of a come-from assertion and a jump—a conditional come-from assertion and a conditional jump, an unconditional come-from assertion and an unconditional jump, and `entry` and `exit`—translated by $\mathcal{T}_{flow}$ is inverse to each other modulo the administration of the edge states $x_{j,i}^{k}$.

In the *SRL* program generated by $\mathcal{T}_{blks}$, the tests at the entries of all conditionals are orthogonal, and the assertions at the exits of all conditionals (except the bottom one) are also orthogonal. The control flow in the target program is dispatched according to the edge state. The previous node state be appropriate only when the come-from assertions are simulated. (Otherwise, the control flow goes down to the bottom layer, and the computation abnormally halts.)

The resulting flowchart $\mathcal{T}_{RL}[\![q]\!]$ has a structure similar to the structured reversible flowchart F' in Fig. 8, except for the representation and manipulation of the edge state simulated by the auxiliary Boolean variables $x_{j,i}^{k}$. The argument of the correctness proof of the translator $\mathcal{T}_{RL}$ is similar to the proof of Theorem 3 when adapted to the different representation of the edge states and is not repeated here. Thus, the construction may serve as another proof of the reversible structured program theorem.

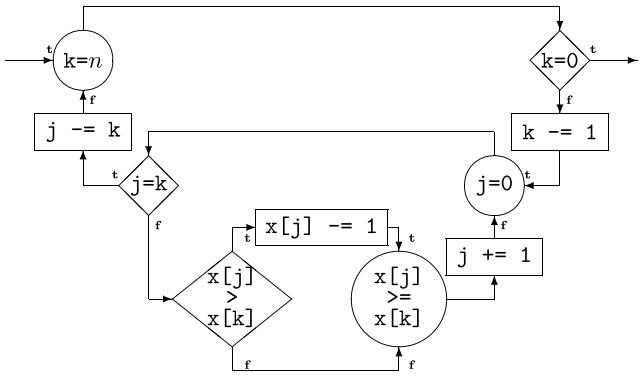


Fig. 23. Dijkstra's permutation-to-code problem as a structured reversible flowchart.

6. Two worked examples

In this section we show how to apply the ideas and methods presented in this paper to two non-trivial problems.

The first example of *encoding a permutation* [21] illustrates how an irreversible algorithm for an injective function can be made reversible through human insight, rather than mechanic reversibilization. The resulting reversible flowchart is concretized in both of our reversible languages. An interesting point is that a solution for the decoding problem (obtaining the permutation from its code) is usually regarded as harder than the encoding problem, but is easily obtained here by a simple inversion.

The second example is a general scheme for simulating RTMs [8,5] by converting them into reversible flowcharts, without generating garbage. This is a *direct proof* of the r-Turing completeness of reversible flowcharts, without relying on reversibilization techniques. This in turn requires more sophisticated programming techniques, as exemplified by a reversible implementation of the infinite tape of a Turing machine by way of two finite (reversible) stacks.

6.1. Dijkstra's permutation-to-code problem

The following example illustrates the reversible flowchart languages using the *permutation-to-code* problem [21]. The problem to be solved is to encode a permutation of n integers $(0, \ldots, n-1)$ contained in an array `x[0:n-1]` by an array of integers whose k-th element is the number of integers in `x[0:k-1]` that are smaller than `x[k]`. For example, given the permutation `[2,0,3,1,5,4]`, its code is `[0,0,2,1,4,4]`. The encoding algorithm was originally defined using Dijkstra's guarded command language with pre- and post-conditions, to derive the inverse algorithm [21,29].

Dijkstra's encoding algorithm, expressed as a reversible flowchart in Fig. 23, calculates the code of a permutation in the array `x[0:n-1]` *in situ* by iteration from right to left, i.e., from the last permutation element `x[n-1]` to the first `x[0]`. Use is made of the observation that for any permutation of k integers $(0, \ldots, k-1)$ contained in `x[0:k-1]`, the code of the element `x[k-1]` is equal to itself, i.e. the code is `x[k-1]`. Thus, in the main loop of the algorithm, what remains is to maintain the invariant for the next iteration, namely, that `x[0:k-2]` contains a permutation of the integers $0, \ldots, k-2$. This is done by decrementing every integer in `x[0:k-2]` that is greater than `x[k-1]` by 1. The code of the permutation is not changed by this modification done by the innermost loop. The reversible flowchart differs from the original formulation [21] in that both loops are guarded by an entry assertion (`k=`n, `j=0`), and the index variable `j` is not destructively overwritten by zero (`j := 0`), but reset by a reversible update (`j -= k`).

The reversible flowchart diagram in Fig. 23 can now be expressed as a reversible program text using our structured and unstructured languages.

The *SRL* program is shown in the left column of Fig. 24. The two nested loops in Fig. 23 are expressed using two loops (`from ... until`). A conditional forms the body of the innermost loop (`if ... fi`). As syntactic sugar, we omit the identity operations (`do skip` and `else skip`). The right column in Fig. 24 shows the inverse program produced by the program inverter in Fig. 18: a decoder that reconstructs the permutation from the given code. It is worthwhile to note that the encoder and decoder take the same number of computation steps on the corresponding input/output and consume the same space.

The *RL* program is shown in Fig. 25. This requires labeling each block (`l0`,...,`l8`) and connecting them using control-flow operators for the outgoing flow (`goto` and `if`) and incoming flow (`from` and `fi`). The entry and exit blocks of a program are marked with `entry` and `exit`. The *RL* program is an implementation of the encoder in Fig. 23. The program can be easily inverted using the flowchart inverter $\mathcal{I}_{RL}$ in Fig. 19 (the inverted program is omitted from this paper).

In all three formalizations, as a flowchart diagram or as an *SRL* or *RL* program, the problem of writing a corresponding decoding program is solved by program inversion, once we have defined the encoder in the above way. In general, defining a reversible encoder may be complicated, cf. Yokoyama *et al.* [65], but it is offset by the easy program inversion to some degree.

Permutation-to-code:

```
from  k=n
loop  k -= 1
      from  j=0
      loop  if x[j]>x[k]
            then x[j] -= 1
            fi x[j]>=x[k]
            j += 1
      until j=k
      j -= k
until k=0
```

$\Longleftrightarrow$ *program inversion* $\mathcal{I}_{SRL}$

Code-to-permutation:

```
from  k=0
loop  j += k
      from  j=k
      loop  j -= 1
            if x[j]>=x[k]
            then x[j] += 1
            fi x[j]>x[k]
      until j=0
      k += 1
until k=n
```

Fig. 24. Permutation encoder and decoder as a structured reversible *SRL* program.

Permutation-to-code:

```
l0: entry
    goto l1
l1: fi k=n from l0 else l7
    if k=0 goto l8 else l2
l2: from l1
    k -= 1
    goto l3
l3: fi j=0 from l2 else l6
    if j=k goto l7 else l4
l4: from l3
    if x[j]>x[k] goto l5 else l6
l5: from l4
    x[j] -= 1
    goto l6
l6: fi x[j]>=x[k] from l5 else l4
    j += 1
    goto l3
l7: from l3
    j -= k
    goto l1
l8: from l1
    exit
```

Fig. 25. Permutation encoder as an unstructured reversible *RL* program.

$$\begin{array}{ll} \langle\ 1,\ \langle \mathtt{b},\mathtt{b}\rangle,\ 2\ \rangle & \langle\ 3,\ \langle \mathtt{b},\mathtt{b}\rangle,\ 4\ \rangle \\ \langle\ 2,\ \ \rightarrow,\ \ 3\ \rangle & \langle\ 4,\ \ \leftarrow,\ \ 5\ \rangle \\ \langle\ 3,\ \langle 0,1\rangle,\ 4\ \rangle & \langle\ 5,\ \langle 0,0\rangle,\ 4\ \rangle \\ \langle\ 3,\ \langle 1,0\rangle,\ 2\ \rangle & \langle\ 5,\ \langle \mathtt{b},\mathtt{b}\rangle,\ 6\ \rangle \end{array}$$

Fig. 26. RTM for modular binary number incrementation (transition rules δ_{inc}).

6.2. Reversible Turing machine simulation

For our second example, we consider the problem of clean translation from RTMs to reversible flowcharts. That is, we show how to construct a reversible flowchart F that simulates an RTM T, without additional garbage. Thus, if an RTM computes the injective function f, then f is also computable without garbage data by a reversible flowchart F (and by Theorem 3, by a structured reversible flowchart). Because the RTMs can compute exactly the injective computable functions [5, 9], this construction can serve as a basis for an alternative proof of the r-Turing completeness of reversible flowcharts. We note that the asymptotic time and space complexity behavior of the RTM is preserved by the generated reversible flowchart. Without loss of generality, we consider RTMs with simplified *triple transition rules* [5].

Formally, a Turing machine T is defined by a tuple $(Q, \Sigma, \delta, b, q_s, q_f)$ of a finite set of states Q, a finite set of symbols Σ, starting and final states $q_s, q_f \in Q$, a blank symbol $b \in S$, and a finite set of transition rules δ, which are either symbol rules $\langle q_1, \langle s_1, s_2\rangle, q_2\rangle \in Q \times (\Sigma \times \Sigma) \times Q$ or shift rules $\langle q_1, d, q_2\rangle \in Q \times \{\leftarrow, \rightarrow\} \times Q$.

1. A *symbol rule* $\langle q_1, \langle s_1, s_2\rangle, q_2\rangle$ says that in state q_1 with the tape head reading symbol s_1, write s_2 and change to state q_2.
2. A *shift rule* $\langle q_1, d, q_2\rangle$ says that in state q_1, move the tape head in the direction $d \in \{\leftarrow, \rightarrow\}$ (left, right) and change to state q_2.

For a Turing machine to be reversible, it must be both forward deterministic (in the usual sense) and backward deterministic. A Turing machine is *forward deterministic* if and only if for any pair of distinct triples $\langle q_1, \tau, q_2\rangle$ and $\langle q_1', \tau', q_2'\rangle$, if $q_1 = q_1'$, then $\tau = \langle s_1, s_2\rangle$, $\tau' = \langle s_1', s_2'\rangle$, and $s_1 \neq s_1'$. That is, if two rules have the same source state, then they must be symbol rules with distinct read symbols. Similarly, a Turing machine is *backward deterministic* if and only if for any pair of distinct triples $\langle q_1, \tau, q_2\rangle$ and $\langle q_1', \tau', q_2'\rangle$, if $q_2 = q_2'$, then $\tau = \langle s_1, s_2\rangle$, $\tau' = \langle s_1', s_2'\rangle$, and $s_2 \neq s_2'$. That is, if two rules have the same target state, then they must be symbol rules with distinct write symbols.

Example Consider the RTM $(\{1, 2, 3, 4, 5, 6\}, \{\mathtt{b}, 0, 1\}, \delta_{inc}, \mathtt{b}, 1, 6)$ with the transition rules δ_{inc} defined in Fig. 26. It increments an n-bit binary number by one (modulo 2^n), represented with the least significant bit first. The tape head, initially to the left of the first bit, moves to the right, flipping bits until it flips a `0` to a `1` or encounters a blank, and then returns to the original position (e.g. `1101` $\rightarrow$ `0011`, `1111` $\rightarrow$ `0000`).

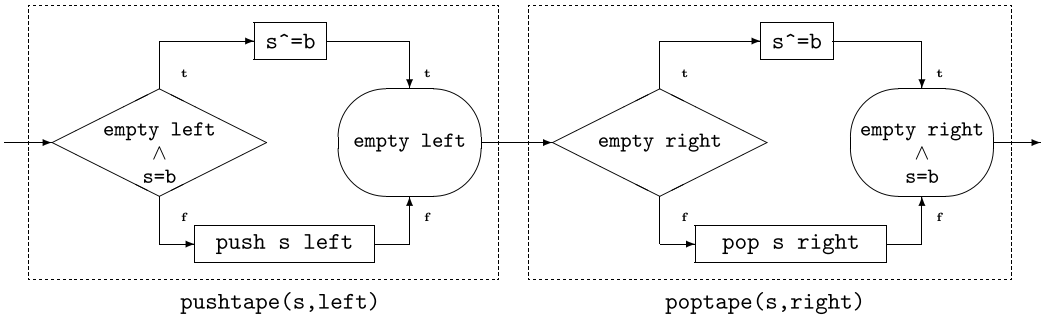


Fig. 27. Reversible flowchart implementation of $M_{\rightarrow}$ using a finite stack representation of the infinite tape. The flowchart of $M_{\leftarrow}$ can be obtained by inverting that of $M_{\rightarrow}$.

```
// SRL code for pushtape(s,left) ≡
   if empty left && s=b then
     s ^= b
   else
     push s left
   fi empty left
```

```
// SRL code for poptape(s,right) ≡
   if empty right then
     s ^= b
   else
     pop s right
   fi empty right && s=b
```

Fig. 28. *SRL* implementations of the macros `pushtape(s,left)` and `poptape(s,right)`.

6.2.1. Tape representation using finite stacks

Before detailing how RTMs can be mechanically translated into reversible flowcharts, we shall consider an important problem relating to which data structure should represent the tape. Previously [64], we used two reversible stacks to represent the (doubly infinite) tape, and for convenience, we assumed the stacks to be infinitely deep. However, the reversible stacks presented in this paper all have a finite depth.[8] Fortunately, finite stacks will work as well, although extra care must be taken to maintain reversibility. This requires a programming trick previously developed by the authors exactly for this problem [63] and that has proven to be useful in other contexts as well [6].

To simulate the infinite tape, we use two finite reversible stacks, `left` and `right`, that hold all non-blank symbols[9] on the left and right portions of the tape with respect to the tape head, respectively, and a variable, `s`, which contains the symbol under the tape head. Moving left (right, resp.) on the tape is essentially achieved by pushing `s` onto `right` (`left`, resp.) and popping an element from `left` (`right`, resp.), giving a blank symbol if the stack is empty.

A problem now arises with the uniqueness of the tape representation: an empty stack simulates a half-infinite blank tape, but naïvely, so does the stack with just a blank at the bottom. If both of these representations can occur, then popping a blank from either of these representations leads to the same representation of the tape, breaking reversibility.

This is resolved by forcing the representation of the tape (and tape-head position) to respect the following invariant: only *non-blank* symbols may be at the bottom of either stack. To conserve the invariant, we start with stacks being initialized with respect to the invariant, and never push a *blank* onto an empty stack: popping an empty stack produces a blank, but "pushing" a blank onto an empty stack produces just the empty stack (and a cleared push-variable).

We introduce the function-like macros `pushtape(s,stk)` and `poptape(s,stk)` for pushing and popping `s` to and from the stack `stk` (Fig. 27) with reversible flowcharts. We note that these are straightforwardly implemented in *SRL*, *cf.* Fig. 28. The three mutually exclusive cases that can occur when pushing `s` onto a stack `stk` are summarized in the following table (where b is the blank symbol, *nil* is the empty stack, and 0 is the value of a cleared variable).

Before `pushtape(s,stk)`	After `pushtape(s,stk)`
$\texttt{stk} = nil \quad \wedge\ \texttt{s} = b$	$\texttt{stk} = nil \quad \wedge\ \texttt{s} = 0$
$\texttt{stk} = nil \quad \wedge\ \texttt{s} = v \neq b$	$\texttt{stk} = v :: nil \ \wedge\ v \neq b \ \wedge\ \texttt{s} = 0$
$\texttt{stk} = s \neq nil \ \wedge\ \texttt{s} = w$	$\texttt{stk} = w :: s \quad \wedge\ \texttt{s} = 0$

Because reversible flowcharts are invertible, `pushtape` can be inverted to yield a flowchart implementing `poptape`. We denote a reversible flowchart block of moving the tape head in the direction d as M_d. For the RTM simulation, we let

$$M_{\rightarrow} \stackrel{\text{def}}{=} \texttt{pushtape(s,left) ; poptape(s,right)}$$

(analogously for $M_{\leftarrow}$). The flowchart implementation of $M_{\rightarrow}$ is shown in Fig. 27. We use this step operation M_d in the simulation in the next subsection. It is clear by simple consideration of the behavior of each M_d, that it computes the desired move simulation on the tape representation (`left`, `s`, `right`), i.e., that

[8] See Section 4 for the precise semantics for reversible stacks.

[9] The tape is assumed to contain only finitely many non-blank symbols at initialization; thus, the non-blank content will always be of finite extent.

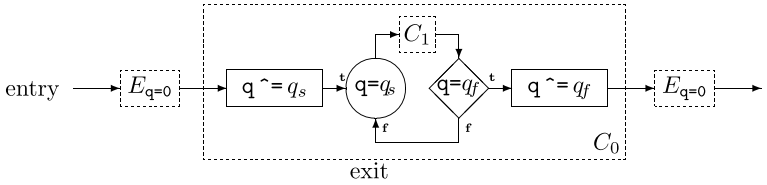


Fig. 29. RTM simulation by flowchart C_0 with main loop and domain restriction $E_{q=0}$.

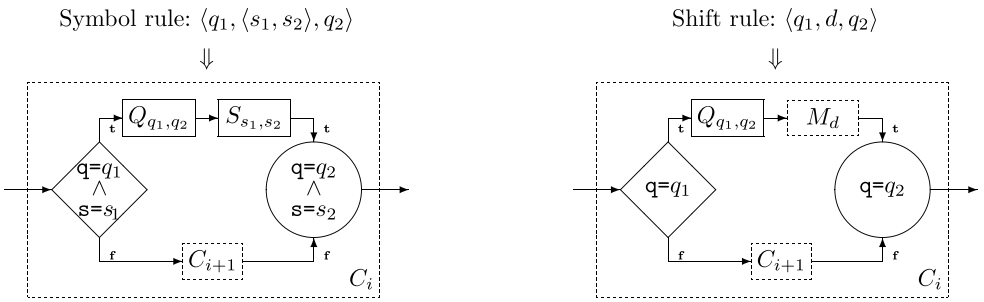


Fig. 30. Simulation of the transition rule R_i by the reversible flowchart C_i.

$$
\begin{array}{lll@{\qquad}lll}
[\![M_{\leftarrow}]\!](nil, b, nil) & = & (nil, b, nil) & [\![M_{\leftarrow}]\!](u :: t, b, nil) & = & (b :: u :: t, b, nil)\\
[\![M_{\leftarrow}]\!](nil, b, w :: s) & = & (nil, w, s) & [\![M_{\leftarrow}]\!](u :: t, b, w :: s) & = & (b :: u :: t, w, s)\\
[\![M_{\leftarrow}]\!](nil, v, nil) & = & (v :: nil, b, nil) & [\![M_{\leftarrow}]\!](u :: t, v, nil) & = & (v :: u :: t, b, nil)\\
[\![M_{\leftarrow}]\!](nil, v, w :: s) & = & (v :: nil, w, s) & [\![M_{\leftarrow}]\!](u :: t, v, w :: s) & = & (v :: u :: t, w, s)
\end{array}
$$

where $v \neq b$, and analogously for $M_{\rightarrow}$.

6.2.2. From RTM-transition rules to flowcharts

The main idea of the rule translation is to simulate the execution of an RTM by the application of one transition rule for each iteration of a reversible main loop, cf. Fig. 29, in much the same style as in Theorem 3. The configuration of an RTM is simulated as follows. `q` is a variable whose value is the (internal) current state, and `s`, `left`, and `right` simulate the tape as detailed above.

For a given RTM, assume that the transition rules are numbered R_1 to R_n. Each transition rule R_i is translated into a functionally equivalent reversible flowchart C_i according to the rules shown in Fig. 30. The helper function Q_{q_1,q_2} consists of the step `q ^= `q_1 ; `q ^= `q_2. This changes the value of `q` from q_1 to q_2 reversibly, which simulates changing the state of the RTM from q_1 to q_2. S_{s_1,s_2} is entirely analogous for the symbol variable `s`, i.e., `s ^= `s_1 ; `s ^= `s_2. The step M_d, defined in Fig. 27, simulates moving the head in direction d, as detailed above. In the translation of both rule types, if the transition rule R_i does not apply (enforced by the test predicate), the control flows into C_{i+1}, the translation of rule R_{i+1}. Upon returning from C_{i+1}, the backward determinism of the RTM ensures that the given assertions are sufficient to differentiate between the cases. In the translation of the final rule R_n, the divergence flowchart $E_{\texttt{false}}$ is inserted in place of C_{n+1}. The execution of C_1 thus simulates one application of the transition function on the (simulated) configuration of the RTM. Assuming the correctness of the S_{s_1,s_2}, Q_{q_1,q_2}, and M_d flowcharts, this can formally be proven by an easy induction over the number of rules.

Finally, C_1 is embedded in a reversible loop C_0 that executes C_1 repeatedly, starting in the state q_s until the final state q_f is reached (Fig. 29). C_0 thus simulates the RTM cleanly without the generation of garbage data. Analogous to the proof of Theorem 3, this can be formally proven by induction over the computation step sequence of the RTM, using the fact that each iteration of the loop simulates a single computation step. We omit the (tedious) formal proof here.

Similar to the domain restriction performed in flowchart F' in Fig. 8, `q` can be guaranteed to be zero-cleared at the entry and exit of C_0 by $E_{q=0}$, cf. Fig. 10. Thus, `q` does not contain garbage before or after RTM simulation.

The translation can be regarded as an alternative proof that reversible flowcharts are r-Turing-complete. However, unlike the (implied) proof via reversibilization in Section 2.6, this translation further shows that for any RTM, there is an equivalent structured reversible flowchart with asymptotically the same time and space complexity. By the tape representation used, the space usage is trivially the same. Every iteration of the main loop simulates the application of a transition rule (one time step of the RTM), and under reasonable assumptions about the time complexity of the reversible flowchart operations used, the time for a single loop iteration is bounded above by a constant dependent on the RTM but not on the input. Because our languages *SRL* and *RL explicitly* support the data structures used in the target flowchart, a further translation into concrete programming languages is quite simple.

Proposition 4 *(r-Turing-completeness of SRL and RL). The reversible flowchart languages SRL and RL are r-Turing-complete.*

```
// Assertion flowchart E_{q=0}
if q=0 fi true

// Flowchart C_0
q ^= 1

from  q=1
do    // Flowchart C_1
      if q=1 && s=b then
        q ^= 1; q ^= 2
        s ^= b; s ^= b
      else
        // Flowchart C_2
        if q=2 then
          q ^= 2; q ^= 3
          // Flowchart M_→
          pushtape(s,left)
          poptape(s,right)
        else
          // Flowchart C_3
          if q=3 && s=0 then
            q ^= 3; q ^= 4
            s ^= 0; s ^= 1
          else
            // Flowchart C_4
            if q=3 && s=1 then
              q ^= 3; q ^= 2
              s ^= 1; s ^= 0
            else
              // Flowchart C_5
              if q=3 && s=b then
                q ^= 3; q ^= 4
                s ^= b; s ^= b
              else
                // Flowchart C_6
                if q=4 then
                  q ^= 4; q ^= 5
                  // Flowchart M_←
                  pushtape(s,right)
                  poptape(s,left)
                else
                  // Flowchart C_7
                  if q=5 && s=0 then
                    q ^= 5; q ^= 4
                    s ^= 0; s ^= 0
                  else
                    // Flowchart C_8
                    if q=5 && s=b then
                      q ^= 5; q ^= 6
                      s ^= b; s ^= b
                    else
                      // Flowchart C_9 (E_false)
                      if false fi true
                    fi q=6 && s=b
                  fi q=4 && s=0
                fi q=5
              fi q=4 && s=b
            fi q=2 && s=0
          fi q=4 && s=1
        fi q=3
      fi q=2 && s=b
until q=6

q ^= 6

// Assertion flowchart E_{q=0}
if q=0 fi true
```

Fig. 31. RTM for modular binary number incrementation simulated by a reversible *SRL* program.

To the best of our knowledge, a reversible implementation of an RTM interpreter without garbage data was first reported in [63]. A translation from RTMs to a functional reversible language was reported in [66].

Example We demonstrate the translation into *SRL* of the RTM for modular binary number incrementation, as specified in Fig. 26. We follow the translation scheme above, inline the translations of Fig. 29 and Fig. 30, and obtain the code in Fig. 31. The domain restriction $E_{q=0}$, enclosing the main flowchart C_0, is the textual representation of Fig. 10. The i-th rule in Fig. 26 is inlined to the textual representation of C_i in Fig. 30 for i satisfying $1 \leq i \leq 8$. The *SRL* implementations of `pushtape` and `poptape` are shown in Fig. 28. In the translation of the final rule in Fig. 26, we use the divergence flowchart $E_{\texttt{false}}$. If no rule in the given Turing program is applicable and the final state q_f $(= 6)$ is not reached, the control flow reaches the divergence flowchart and the computation halts abnormally.

7. Related work

The usefulness of structured programming and goto statements in the form of exit/break/return statements from the middle of the loop were controversial subjects in the 1960s and 1970s [15,20,37]. However, from a theoretical point of view, the computational power of structured and unstructured programs was known to be equal very early in this debate. Böhm and Jacopini [11] showed that any flowchart can be transformed into a structured flowchart. In a letter to the editor in the same journal, Cooper suggested that only a single loop is necessary for this transformation [14], simplifying the transformation at the expense of the loss of the structure of the original flowchart. In this paper, we have proved the reversible version of the Structured Program Theorem suggested by Cooper. Interesting details and many references on the original theorem were collected in [30].

Many different (irreversible) general computation models have been proposed for investigating the mathematical properties of computation, all computationally equal in power. Similarly, the concept of reversibility has been studied using various computation models. Reversible computation models that can simulate an RTM without any garbage are called r-Turing-complete [5]. Many r-Turing-complete computation models are constructed by imposing restrictions on basic computation steps or on a subset of corresponding irreversible computation models. RTMs [8,49] are Turing machines with bijective deterministic transition functions. Reversible counter machines, which can be formalized by a multi-tape RTM

with read-only heads, are counter machines with bijective deterministic transition functions [48]. Reversible cellular automata [35,48] are equipped with bijective global or local functions. Reversible Boolean logic circuits [18,24,61] consist of reversible gates. The billiard ball model [24] is a reversible physical computation model that consists of perfectly elastically colliding balls and reflectors. A reversible process algebra based on Milner's CCS, allows processes to backtrack along any causally equivalent traces, on which loops and confluent paths are regarded as equivalent [16]. This is applied in a formal model using CCS-R for molecular biology [17]. Reversible cellular automata and reversible process algebras target massively parallel and concurrent computation models. In reversible combinatory logic [19], the information of which combinators are reduced and lost at each reduction step is added to the reduced terms, which guarantees deterministic backward reduction. A reversible random access machine [7] can be regarded as the simplest generic reversible architecture. Although each of these models can, in principle, simulate any computation, their representation can become very complex. From the viewpoint of software development, a computation model should be used for better understanding of programming. Flowcharts can naturally retain the structure of programs, and their modern extension, UML activity diagrams, are widely used in the software development process [52].

In this paper, we investigated the flowchart model on which the design of programming languages is based. Several reversible programming languages have been proposed. In particular, reversible languages that ensure the reversibility of programs by reversibly composing reversible primitives are as follows. To the best of our knowledge, the first reversible language is Janus [40], which has been formalized by the authors [63,67]. On the basis of Janus, the hardware language SyReC has been designed for reversible circuit synthesis [62]. Given R [23] source programs, an R compiler generates PISA (Pendulum instruction set architecture) code, which runs on the reversible processor Pendulum [7,60]. Related work includes the minimal reversible instruction set architecture Bob [57], for which a reversible arithmetic logic unit was developed [58]. A clean compiler from Janus to PISA has been also designed [3]. Gries' invertible language [29] is a meta-language designed for reasoning. It can also be regarded as a reversible language. The point-free functional language Inv [50] and Π [12,34], are also reversible in a sense. The simple imperative language (E)SRL [42] also belongs to the class of reversible languages. The control flow operators in high-level imperative reversible languages like Janus, SyRec, R, Gries' invertible language, and (E)SRL are all instances of the structured reversible CFOs in Fig. 5. The unstructured control flow of low-level imperative reversible languages with jump instructions, like PISA [23] and Bob [57], is modeled by the atomic operations in Fig. 2.

Computational universality is a desirable property for general programming languages. To be computationally universal in a reversible setting, reversible programming languages need to be r-Turing complete, as shown for the flowchart languages *RL* and *SRL* in this paper. Most of the above clean reversible languages have this property too. A notable exception is (E)SRL, which is used to characterize sub-universal classes; the class of the computation of SRL (ESRL, resp.) programs without nested loops is equal to the class of linear transformations $f(\vec{x}) = M\vec{x} + C$, where the matrix M has the determinant $+1$ (± 1, resp.). Though the original Janus [40,67] was not r-Turing complete, Janus with reversible stacks is [63].

The $\mathcal{MOQA}$ (MOdular Quantitative Analysis) language is used as a basis for analyzing program time complexity; reasoning in this language is made easier when the programs are reversible [55]. Recently, steps were made towards developing the reversible functional language RFUN [66], which also takes a clean approach. Methods for representing constructor terms on a reversible heap and translating RFUN into the reversible assembly language PISA without generating garbage have been proposed [6].

Reversibilization, the transformation of irreversible computations into reversible computations, has also been extensively studied in several languages. Reversibilization can be realized interpretively by reversible runtime systems. For example, the SEMCD machine [36] keeps track of sufficient information to reconstruct the backward computation. The reversible virtual machine (RVM) provides an efficient execution environment using a history stack for the stack-based language Forth [56]. Reversible process calculi embed a computational history in each thread and allow nondeterministic choices of causally equivalent paths when backtracking [16,53]. Despite the backward nondeterminism, reversibilized processes are guaranteed not to reach computation states inaccessible by the original processes. As another approach, such history information is embedded into the target language. The probabilistic guarded-command language (pGCL) was successfully embedded into a reversible programming language [68]. The point-free functional language Fun was also embedded into the injective language Inv [50]. The reversibilization technique has been successfully applied to several computation models such as reversible cellular automata [59], reversible logic gates [24], reversible combinatory logic [2,19], reversible flowcharts [64], and RTMs [5].

A subject closely related to reversible programming is inverse computation and program inversion, e.g., [1,21,26,27,29,32,43]. Generalized program inversion generates a semi-inverse program in the sense that, given some of the original inputs and outputs, it returns the remaining inputs and outputs [45,51].

Reversibilization and program inversion are used in the context of efficient optimistic parallel simulations [13]. The translations are all done in a subset of the irreversible imperative programming language C. Given a program, its reversibilized program and its (left-) inverse program are generated. The reversibilized program and the inverse program simulate the forward and backward computation of the original program, respectively.

Even in classical computation, undoing and redoing computation steps are facilities needed in many situations. Independently of a particular application, Leeman formalized general-purpose undo/redo operations [39], but those operations are not intended to provide a reversible language. In fact, one of his primitive operators realizing an undo operation deletes the recorded history until a desired previous program state. However, understanding the fundamentals of reversible com-

puting also provides a foundation for developing well-founded and efficient undo/redo capabilities in irreversible computing systems.

Finally, programming languages for bidirectional transformation, which also have notions of forward and backward semantics, have been extensively investigated to address the view updating problem [22,33]. However, a bidirectional language is not necessarily reversible or clean.

8. Conclusion

In this paper we introduced reversible flowcharts; showed that they have properties that are noteworthy in their own right, such as easy inversion and guaranteed termination on bounded state configuration spaces; and used the same formalism for a wide range of different purposes. We proved the r-Turing-completeness of reversible flowcharts and showed the reversibilization of classical flowcharts, which serve as proof of the computation universality of reversible flowcharts. We showed that structured and unstructured reversible flowcharts are equally expressive and that structured control-flow operators are "free" also in a reversible setting. We presented two concrete examples of structured and unstructured programming languages with complete syntax and semantics based on the reversible flowchart model. An interesting observation is that the object-language properties are reflected on the meta-level of formal semantics and program transformation, which makes them accessible to verification and analysis.

The foundations presented in this paper can be extended in a variety of directions including automata theory, computability and complexity, language semantics, and program transformation. The results can also serve as guidelines for designing new reversible programming languages independent of actual implementation. We believe that further investigation of the fundamental properties of reversible computing models can lead to interesting and important results and insights in other (irreversible) areas as well.

Acknowledgements

The authors thank Poul Clementsen, Kenji Moriyama, and Michael Kirkedal Thomsen for valuable comments and discussions. This work was partly supported by MEXT KAKENHI 25730049, the Danish Council for Strategic Research under the *MicroPower* project (0603-00222B), and the Danish Council for Independent Research | Natural Sciences under the *Foundations of Reversible Computing* project (12-126689).

References

[1] S.M. Abramov, R. Glück, Principles of inverse computation and the universal resolving algorithm, in: T.Æ. Mogensen, D.A. Schmidt, I.H. Sudborough (Eds.), The Essence of Computation: Complexity, Analysis, Transformation, in: Lecture Notes in Computer Science, vol. 2566, Springer-Verlag, 2002, pp. 269–295.
[2] S. Abramsky, A structural approach to reversible computation, Theoret. Comput. Sci. 347 (3) (2005) 441–464.
[3] H.B. Axelsen, Clean translation of an imperative reversible programming language, in: J. Knoop (Ed.), Compiler Construction, in: Lecture Notes in Computer Science, vol. 6601, Springer-Verlag, 2011, pp. 144–163.
[4] 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, in: Lecture Notes in Computer Science, vol. 6638, Springer-Verlag, 2011, pp. 117–128.
[5] H.B. Axelsen, R. Glück, What do reversible programs compute?, in: M. Hofmann (Ed.), Foundations of Software Science and Computation Structures. Proceedings, in: Lecture Notes in Computer Science, vol. 6604, Springer-Verlag, 2011, pp. 42–56.
[6] 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, in: Lecture Notes in Computer Science, vol. 7948, Springer-Verlag, 2013, pp. 96–109.
[7] H.B. Axelsen, R. Glück, T. Yokoyama, Reversible machine code and its abstract processor architecture, in: V. Diekert, M.V. Volkov, A. Voronkov (Eds.), Computer Science – Theory and Applications. Proceedings, in: Lecture Notes in Computer Science, vol. 4649, Springer-Verlag, 2007, pp. 56–69.
[8] C.H. Bennett, Logical reversibility of computation, IBM J. Res. Develop. 17 (6) (1973) 525–532.
[9] C.H. Bennett, Thermodynamics of computation—a review, Internat. J. Theoret. Phys. 21 (12) (1982) 905–940.
[10] C.H. Bennett, Time/space trade-offs for reversible computation, SIAM J. Comput. 18 (4) (1989) 766–776.
[11] C. Böhm, G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, Commun. ACM 9 (5) (1966) 366–371.
[12] W.J. Bowman, R.P. James, A. Sabry, Dagger traced symmetric monoidal categories and reversible programming, in: A. De Vos, R. Wille (Eds.), Reversible Computation. Proceedings, in: Lecture Notes in Computer Science, vol. 7165, Springer-Verlag, 2011, pp. 51–56.
[13] C.D. Carothers, K.S. Perumalla, R.M. Fujimoto, Efficient optimistic parallel simulations using reverse computation, ACM Trans. Model. Comput. Simul. 9 (3) (1999) 224–253.
[14] D.C. Cooper, Böhm and Jacopini's reduction of flow charts, Commun. ACM 10 (8) (1967) 463, 473.
[15] O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare (Eds.), Structured Programming, Academic Press, 1972.
[16] V. Danos, J. Krivine, Reversible communicating systems, in: P. Gardner, N. Yoshida (Eds.), Conference on Concurrency Theory, in: Lecture Notes in Computer Science, vol. 3170, Springer-Verlag, 2004, pp. 292–307.
[17] V. Danos, J. Krivine, Formal molecular biology done in CCS-R, Electron. Notes Theoret. Comput. Sci. 180 (3) (2007) 31–49.
[18] A. De Vos, Reversible Computing: Fundamentals, Quantum Computing, and Applications, Wiley-VCH, 2010.
[19] A. Di Pierro, C. Hankin, H. Wiklicky, Reversible Combinatory Logic, Math. Structures Comput. Sci. 16 (4) (2006) 621–637.
[20] E.W. Dijkstra, Letters to the editor: go to statement considered harmful, Commun. ACM 11 (3) (1968) 147–148.
[21] E.W. Dijkstra, Program inversion, in: F.L. Bauer, M. Broy (Eds.), Program Construction: International Summer School, in: Lecture Notes in Computer Science, vol. 69, Springer-Verlag, 1978, pp. 54–57.
[22] J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, A. Schmitt, Combinators for bi-directional tree transformations: a linguistic approach to the view update problem, ACM Trans. Program. Lang. Syst. 29 (3) (2007) 1–65.
[23] M.P. Frank, Reversibility for efficient computing, Ph.D. thesis, EECS Dept., Massachusetts Institute of Technology, 1999.

[24] E. Fredkin, T. Toffoli, Conservative logic, Internat. J. Theoret. Phys. 21 (1982) 219–253.
[25] 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: Lecture Notes in Computer Science, vol. 2895, Springer-Verlag, 2003, pp. 246–264.
[26] R. Glück, M. Kawabe, Derivation of deterministic inverse programs based on LR parsing, in: Y. Kameyama, P.J. Stuckey (Eds.), Functional and Logic Programming. Proceedings, in: Lecture Notes in Computer Science, vol. 2998, Springer-Verlag, 2004, pp. 291–306.
[27] R. Glück, M. Kawabe, Revisiting an automatic program inverter for Lisp, SIGPLAN Not. 40 (5) (2005) 8–17.
[28] C.K. Gomard, N.D. Jones, Compiler generation by partial evaluation: a case study, Struct. Program. 12 (1991) 123–144.
[29] D. Gries, The Science of Programming, Texts and Monographs in Computer Science, Springer-Verlag, 1981, pp. 265–274, Ch. 21 Inverting Programs.
[30] D. Harel, On folk theorems, SIGACT News 12 (3) (1980) 68–80.
[31] J. Hatcliff, An introduction to online and offline partial evaluation using a simple flowchart language, in: J. Hatcliff, T. Mogensen, P. Thiemann (Eds.), Partial Evaluation. Practice and Theory, in: Lecture Notes in Computer Science, vol. 1706, Springer-Verlag, 1999, pp. 20–82.
[32] C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, R. Vuduc, A new method for program inversion, in: M. O'Boyle (Ed.), Compiler Construction, in: Lecture Notes in Computer Science, vol. 7210, Springer-Verlag, 2012, pp. 81–100.
[33] Z. Hu, S.-C. Mu, M. Takeichi, A programmable editor for developing structured documents based on bidirectional transformations, High.-Order Symb. Comput. 21 (1–2) (2008) 89–118.
[34] R.P. James, A. Sabry, Information effects, in: Principles of Programming Languages. Proceedings, ACM Press, 2012, pp. 73–84.
[35] J. Kari, Theory of cellular automata: a survey, Theoret. Comput. Sci. 334 (1–3) (2005) 3–33.
[36] W.E. Kluge, A reversible SE(M)CD machine, in: P. Koopman, C. Clack (Eds.), Implementation of Functional Languages. Proceedings, Selected Papers, in: Lecture Notes in Computer Science, vol. 1868, Springer-Verlag, 2000, pp. 95–113.
[37] D.E. Knuth, Structured programming with go to statements, ACM Comput. Surv. 6 (4) (1974) 261–301.
[38] R. Landauer, Irreversibility and heat generation in the computing process, IBM J. Res. Develop. 5 (3) (1961) 183–191.
[39] G.B. Leeman Jr., A formal approach to undo operations in programming languages, ACM Trans. Program. Lang. Syst. 8 (1) (1986) 50–87.
[40] C. Lutz, Janus: a time-reversible language, Letter to R. Landauer, 1986.
[41] Z. Manna, Mathematical Theory of Computation, McGraw Hill, 1974.
[42] A.B. Matos, Linear programs in a simple reversible language, Theoret. Comput. Sci. 290 (3) (2003) 2063–2074.
[43] K. Matsuda, M. Wang, FliPpr: a prettier invertible printing system, in: M. Felleisen, P. Gardner (Eds.), Programming Languages and Systems. Proceedings, in: Lecture Notes in Computer Science, vol. 7792, Springer-Verlag, 2013, pp. 101–120.
[44] 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.
[45] T.Æ. Mogensen, Semi-inversion of guarded equations, in: R. Glück, M. Lowry (Eds.), Generative Programming and Component Engineering. Proceedings, in: Lecture Notes in Computer Science, vol. 3676, Springer-Verlag, 2005, pp. 189–204.
[46] T.Æ. Mogensen, Partial evaluation of the reversible language Janus, in: Partial Evaluation and Program Manipulation. Proceedings, ACM Press, 2011, pp. 23–32.
[47] T.Æ. Mogensen, Reference counting for reversible languages, in: S. Yamashita, S. Minato (Eds.), Reversible Computation. Proceedings, in: Lecture Notes in Computer Science, vol. 8507, Springer-Verlag, 2014, pp. 82–94.
[48] K. Morita, Reversible computing and cellular automata — a survey, Theoret. Comput. Sci. 395 (1) (2008) 101–131.
[49] K. Morita, Y. Yamaguchi, A universal reversible Turing machine, in: J. Durand-Lose, M. Margenstern (Eds.), Machines, Computations, and Universality. Proceedings, in: Lecture Notes in Computer Science, vol. 4664, Springer-Verlag, 2007, pp. 90–98.
[50] S.-C. Mu, Z. Hu, M. Takeichi, An injective language for reversible computation, in: D. Kozen (Ed.), Mathematics of Program Construction. Proceedings, in: Lecture Notes in Computer Science, vol. 3125, Springer-Verlag, 2004, pp. 289–313.
[51] N. Nishida, M. Sakai, T. Sakabe, Partial inversion of constructor term rewriting systems, in: J. Giesl (Ed.), Term Rewriting and Applications. Proceedings, in: Lecture Notes in Computer Science, vol. 3467, Springer-Verlag, 2005, pp. 264–278.
[52] Object Management Group, OMG unified modeling language UML (OMG UML): superstructure, version 2.4.1, Tech. rep. formal/2011-08-06, 2011.
[53] I. Phillips, I. Ulidowski, Reversing algebraic process calculi, J. Log. Algebr. Program. 73 (1–2) (2007) 70–96.
[54] M. Saeedi, I.L. Markov, Synthesis and optimization of reversible circuits — a survey, ACM Comput. Surv. 45 (2) (2013) 21:1–21:34.
[55] M.P. Schellekens, $\mathcal{MOQA}$; unlocking the potential of compositional static average-case analysis, J. Log. Algebr. Program. 79 (1) (2010) 61–83.
[56] B. Stoddart, R. Lynas, F. Zeyda, A virtual machine for supporting reversible probabilistic guarded command languages, in: Reversible Computation. Proceedings, Electron. Notes Theor. Comput. Sci. 253 (6) (2010) 33–56.
[57] 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, vol. 7165, Springer-Verlag, 2012, pp. 30–42.
[58] M.K. Thomsen, R. Glück, H.B. Axelsen, Reversible arithmetic logic unit for quantum arithmetic, J. Phys. A 43 (38) (2010) 382002.
[59] T. Toffoli, Computation and construction universality of reversible cellular automata, J. Comput. System Sci. 15 (2) (1977) 213–231.
[60] C.J. Vieri, Reversible computer engineering and architecture, Ph.D. thesis, EECS Dept., Massachusetts Institute of Technology, 1999.
[61] R. Wille, R. Drechsler, Towards a Design Flow for Reversible Logic, Springer-Verlag, 2010.
[62] R. Wille, S. Offermann, R. Drechsler, SyReC: a programming language for synthesis of reversible circuits, in: Forum on Specification and Design Languages, 2010, pp. 184–189.
[63] T. Yokoyama, H.B. Axelsen, R. Glück, Principles of a reversible programming language, in: Computing Frontiers. Proceedings, ACM Press, 2008, pp. 43–54.
[64] T. Yokoyama, H.B. Axelsen, R. Glück, Reversible flowchart languages and the structured reversible program theorem, in: L. Aceto, I. Damgård, L.A. Goldberg, M.M. Halldórsson, A. Ingólfsdóttir, I. Walukiewicz (Eds.), International Colloquium on Automata, Languages and Programming. Proceedings, in: Lecture Notes in Computer Science, vol. 5126, Springer-Verlag, 2008, pp. 258–270.
[65] T. Yokoyama, H.B. Axelsen, R. Glück, Optimizing clean reversible simulation of injective functions, J. Mult.-Valued Logic Soft Comput. 18 (1) (2012) 5–24.
[66] T. Yokoyama, H.B. Axelsen, R. Glück, Towards a reversible functional language, in: A. De Vos, R. Wille (Eds.), Reversible Computation. Proceedings, in: Lecture Notes in Computer Science, vol. 7165, Springer-Verlag, 2012, pp. 14–29.
[67] T. Yokoyama, R. Glück, A reversible programming language and its invertible self-interpreter, in: Partial Evaluation and Semantics-Based Program Manipulation. Proceedings, ACM Press, 2007, pp. 144–153.
[68] P. Zuliani, Logical reversibility, IBM J. Res. Develop. 45 (6) (2001) 807–818.