Reversible Flowchart Languages and the Structured Reversible Program Theorem

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

[1] NCES, Graduate School of Information Science, Nagoya University
[2] DIKU, Department of Computer Science, University of Copenhagen
yokoyama@nagoya-u.jp, funkstar@diku.dk, glueck@acm.org

Abstract. Many irreversible computation models have reversible counterparts, but these are poorly understood at present. We introduce reversible flowcharts with an assertion operator and show that any reversible flowchart can be simulated by a structured reversible flowchart using only three control flow operators. Reversible flowcharts are *r-Turing-complete*, meaning that they can simuluate reversible Turing machines without garbage data. We also demonstrate the *injectivization* of classical flowcharts into reversible flowcharts. The reversible flowchart computation model provides a theoretical justification for low-level machine code for reversible microprocessors as well as high-level block-structured reversible languages. We give examples for both such languages and illustrate them with a lossless encoder for permutations given by Dijkstra.

1 Introduction

In the microprocessor industry, the *circuit model*, based on well-known logical connectives such as OR and AND, reigns supreme. In recent years, however, energy efficiency has become an increasing concern, since standard desktop processors dissipate on the order of 100W of power, which must be removed as heat. Lowering power consumption while increasing computing power is a non-trivial obstacle for the microprocessor industry, and efforts to do this have involved computer science, physics and engineering.

Non-standard models of computing have therefore received increased attention [17]. One such model is *reversible computing*, which is the only approach known to date that can circumvent the hard, physical barrier to the energy efficiency of irreversible computations (such as the ubiquitous NAND-gate). This physical barrier, the *von Neumann-Landauer* limit, provides a strict lower boundary to the energy dissipated as heat with every bit of information destroyed, whence *irreversibility*. Reversible computing, as well as *reversible programming*, are poorly understood at present. This is unfortunate, since a good understanding of reversible computing is also essential for *quantum computing*, in that every operation on a quantum state must be *unitary*, and therefore invertible and reversible. Low-power CMOS and quantum computing are two of the possible applications for the reversible computing model.

L. Aceto et al. (Eds.): ICALP 2008, Part II, LNCS 5126, pp. 258–270, 2008.

A reversible computing model allows deterministic time-invertible computations, in which not only the next computation state, but also the previous computation state is determined uniquely by the current state. All computations are forward and backward deterministic. Store updates are non-destructive. Although there are several reversible computation models, such as reversible Turing machines [2] and invertible cellular automata [18], they are not sufficiently program-oriented to relate theoretical considerations and recent practical developments [7,9].

Most modern programming languages are *imperative*, with block-structured control flow operators (CFOs) such as `if` and `while`. Structured programs are more readable and maintainable [6]. The theoretical foundation for structured programming is the classic *Structured Program Theorem* from the 1960s [4,5], which guarantees that any unstructured program can be written using only three structured CFOs: *sequence*, *selection* and *loop*. The same property is desirable for reversible programming languages, but it is not obvious that it should carry over from classical computing models.

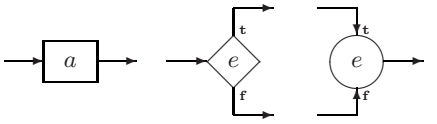
The main goal of the present paper is to provide the theoretical justification for the design, translation and computational strength of high-level imperative reversible languages, such as Janus [14,19,20], and low-level machine code for reversible architectures, such as the Pendulum microprocessor [9,1]. The flowchart model is well suited for this purpose, as it accommodates both low-level aspects such as jumps and high-level aspects such as structured control flow operators.

We identify three reversible CFOs that are sufficient for the definition of a structured reversible flowchart language. We show that reversible flowcharts are *r-Turing-complete*, in that they can simuluate *reversible Turing machines* without garbage data. We show the *injectivization* of classical flowcharts into reversible flowcharts, indicating that the latter are Turing-complete, if garbage data necessary for the injectivity of the computed function are disregared. We present examples of how programming languages based on reversible flowcharts can be designed, along with two code examples.

2 Reversible Flowcharts

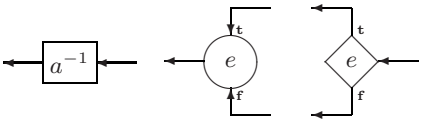
Flowcharts have been used extensively in the study of programming languages. Most programming languages used today have a control flow, which can be easily modeled by flowcharts, making the latter important analytical tools in programming language theory (*e.g.*, [4,5,10,12,15]).

Reversible flowcharts. A *reversible flowchart* F is a finite directed graph with three kinds of nodes, each representing an *atomic operation* (Fig. 1): a *step* performs an elementary operation on the store specified by a transition function a; a *test* dispatches the control flow depending on the value of predicate e; and an *assertion* is a join point that passes incoming control flow through, depending on the value of predicate e. Computation in a flowchart proceeds sequentially along the directed graph of F. A *well-formed* flowchart has exactly one entry and one exit. An interpretation of a flowchart F consists of a domain X (*e.g.*, a store)



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

Fig. 1. Atomic operations of reversible flowcharts



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

Fig. 2. Inverted atomic operations of reversible flowcharts

and an appropriate association with the partial transition functions ($a : X \rightharpoonup X$) and the predicates ($e : X \to \mathit{Bool}$).

The transition function a of each step must be *locally invertible*, defined as having an inverse transition function a^{-1} that can be determined without referring to a's context or location in a flowchart.

The assertion operator is also new (Fig. 1(c)): 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. The operator is represented by a circle.

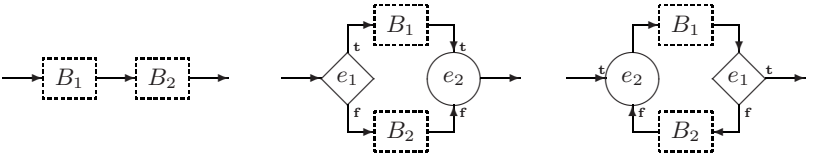
In classical flowcharts, the join points are not associated with a predicate and there is no information about the incoming control flow. Classical join points are sources of *backward non–determinism*, which break reversibility, as do non-invertible transition functions. Reversible flowcharts remove these sources.

Structured reversible flowcharts. Similar to classical flowcharts, reversible flowcharts allow unstructured control flow. Needless to say, it is easy to construct incomprehensible "spaghetti code" with unstructured reversible flowcharts.

A *structured control flow operator* (structured CFO) has exactly one entry and one exit. We define three structured reversible CFOs (Fig. 3): *sequence*, *selection*, and *loop*. A block B_i is either a locally invertible step (as above) or one of the three structured reversible CFOs. The latter can be nested any number of times. The constructs are all symmetric. A *structured reversible flowchart* is one constructed from locally invertible steps and structured reversible CFOs. Structured control flow makes a program modular and easier to verify.

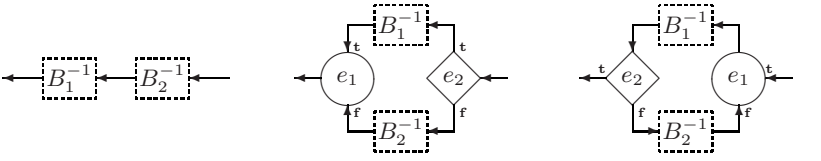
The selection corresponds to an irreversible **if**-statement but has an exit assertion e_2. The loop is repeated as long as test e_1 and assertion e_2 are false. The loop corresponds to an irreversible **while** loop if B_1 is empty and to an irreversible **do-while** loop if B_2 is empty. In either case, the assertion at the loop entry and the test at the loop exit make the loop reversible.

Inverse flowcharts. Starting with a reversible flowchart, structured or unstructured, the following method can be used to generate an *inverse flowchart*: (1) change the direction of each arrow, (2) replace each transition 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).



(a) Sequence (b) Selection (c) Loop

Fig. 3. Structured reversible CFOs



(a) Sequence (b) Selection (c) Loop

Fig. 4. Inverted structured reversible CFOs

Fig. 2 shows the inverse of each atomic operator in Fig. 1, where a^{-1} is the inverse transition function. Similarly, Fig. 4 shows the inverse of each CFO in Fig. 3, where B^{-1} is the inverse flowchart of B.

The flowchart resulting from inversion is also reversible, whether structured or unstructured. Inversion does not add or delete atomic operations or CFOs. Repeating the inversion once more restores the original reversible flowchart. The transformation is purely local and does not require global analyses or changes in control flow beyond changing the direction of the arrows. The ease with which reversible flowcharts are inverted is a unique property of this computation model and makes it an attractive analytical tool for program complexity [13]. In general, it is difficult to construct an inverse flowchart mechanically from a classical flowchart. Clearly, programming reversible flowcharts is quite different from programming classical flowcharts.

3 The Structured Reversible Program Theorem

Nowadays, it is easy to forget that the uses and benefits of structure in high-level programming languages were controversial. From a computational viewpoint, this debate was effectively closed by the *Structured Program Theorem* [4], which showed that structured and unstructured flowcharts have the same expressive power. Thus, the useful ancillary benefits of structured high-level languages, including their increased readability and being much easier to reason about, had no computational weaknesses.

The same question is relevant to the reversible computation paradigm. Reversible computing is sufficiently different from standard computational models

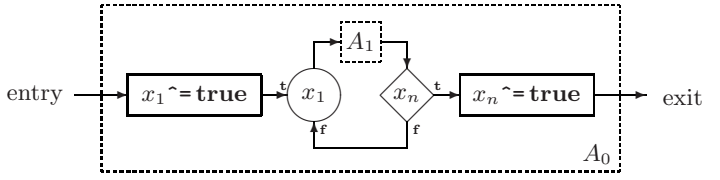


Fig. 5. Flowchart A_0 with main loop

that it is unclear whether results from classical (backward non-deterministic) computing carry over to the reversible paradigm.[1] Indeed, none of the classic constructions (and therefore proofs) apply because they lead to classical irreversible flowcharts. While it may be intuitively obvious that structure is also "free" in reversible programming, this must be proven.

Theorem 1 (Structured Reversible Program Theorem). *For any well-formed reversible flowchart F, a functionally equivalent structured reversible flowchart A_0, 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 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. For this, we label every edge in F uniquely by l_i $(1 \leq i \leq n)$. Without loss of generality, 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 is translated to a structured equivalent. A main loop simulates the control flow of F one node at a time, by keeping track of which edge 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 domain X. The initial and final value of each x_i is **false**. The edge state is called i if x_i is **true** and all other x_j's are **false**. Thus, if in F the control flow is at edge l_i, then the edge state in A_0 should be i. The edge state is changed from i to j by using an injective transition function $P_{i,j}$ that executes x_i **^= true; ** x_j **^= true**.[2] In F this corresponds to moving from edge l_i to edge l_j.

First, generate the main loop in Fig. 5 for entry edge l_1 and exit edge l_n. Flowchart A_0 is a reversible loop between an initial and a final step. The initial step sets x_1 to **true**. If test x_n is **true** the loop ends; otherwise, the loop continues. The path back to assertion x_1 is a skip operation.

Then build the structured reversible flowcharts A_i $(0 < i < n)$ by the rules in Fig. 6, where the atomic operation with incoming edge l_i (or l_i and l_{i+1}) in

[1] As an example, given a bounded store (*i.e.* a finite number of possible configurations), computations cannot be guaranteed to terminate for classical flowcharts. This is *not* true in reversible computing, where a bounded store *is* sufficient to obtain termination for well-formed reversible flowcharts.

[2] x_i **^= true** is shorthand for x_i **:=** $x_i \oplus$ **true**, where $\oplus$ is logical *exclusive-or*. This is an injective (reversible) step.

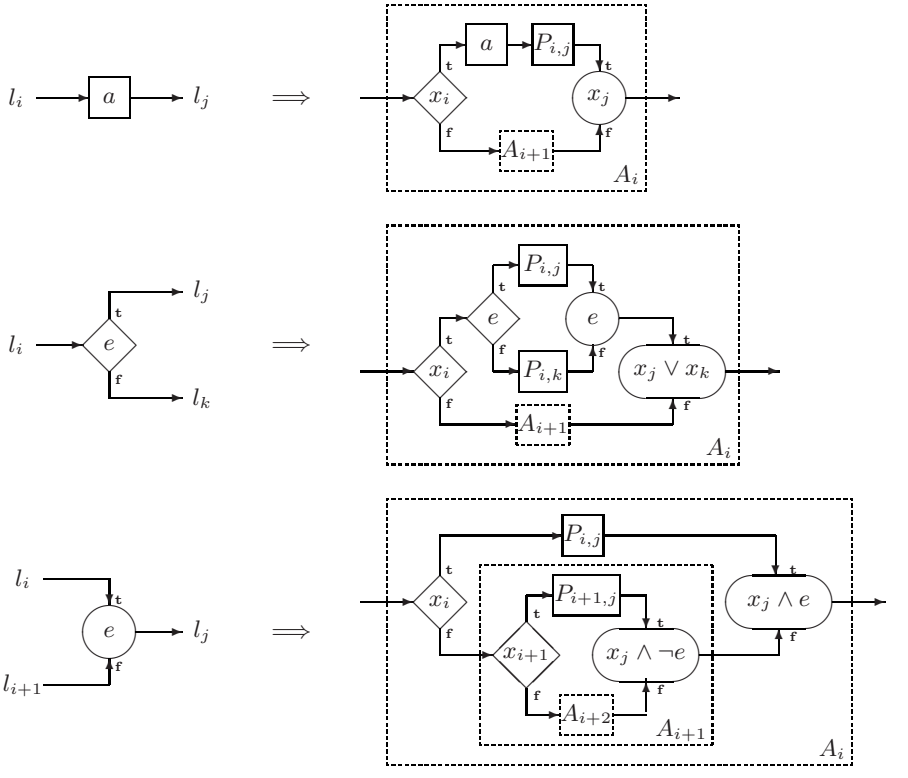


Fig. 6. Unstructured operations transformed into structured reversible flowcharts A_i

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 simulating the execution of control flow along an edge l_j over exactly one node.

(1) A *step* with transition function a is executed in the translated flowchart only if the 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 edges, after executing A_{i+1} the state cannot be j, so an assertion of x_j is sufficient to distinguish between the two possibilities.

(2) A *test* is similar to a step. $P_{i,j}$ and $P_{i,k}$ change variables x_i, x_j and x_k. Thus, the value of predicate e in the test and the assertion must be the same, and depending on e the edge state is set to either j or k. By an argument analogous to the *step* case, the assertion $x_j \vee x_k$ is sufficient to distinguish this from A_{i+1}.

(3) Edges l_i and l_{i+1} of an *assertion* are translated simultaneously, so A_i contains A_{i+1} and A_{i+2}. Predicate e differentiates between the two possibilities.

Finally, for well-formedness, we insert a dummy step A_n (*e.g.*, an identity step) although it is never reached in a computation. The structured reversible flowchart A_0 generated by the rules in Fig. 5 and 6 thus simulates the execution of every step, test and assertion in the flowchart F. □

The proof is constructive, and shows that a structured reversible flowchart can be constructed from an arbitrary reversible flowchart with exactly the same functionality. 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 [5], but was more involved.

4 *r*-Turing Completeness of Reversible Flowcharts

At first glance, reversible flowcharts may not seem as powerful as their classical counterparts, which do not require assertions and allow any transformation on the store in steps.

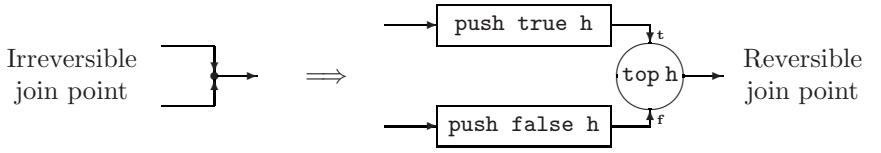
First, we shall demonstrate that reversible flowcharts with unbounded space are *Turing-complete*, provided that the generation of *garbage data*, extraneous data needed for reversibility, is ignored. This can be accomplished by *injectivizing* classical flowcharts (*i.e.*, with irreversible join points and non-injective steps), which are Turing-complete, into reversible flowcharts with the same functionality. Such an injectivization effectively changes the *type* of the computed function: if the classical flowchart F computes function $f : X \to Y$, then the injectivized flowchart will compute a function $f_g : X \to Y \times G$, where G is some domain of garbage data, necessary to guarantee that f_g is an injective function. While injectivization works for any computable function, it is *not* necessary for the large and important class of *injective, computable functions*.[3]

Second, we show that reversible flowcharts are *r-Turing-complete*, meaning that they can compute the same functions as *reversible* Turing machines *cleanly*, *i.e.* without the generation of garbage data. In other words, if a reversible Turing machine (RTM) computes the injective function $f : X \to Y$, then f is computable without garbage data in reversible flowcharts (and by Thm. 1, in structured flowcharts). Since RTMs can cleanly compute *any* injective, computable function [3,13], so can reversible structured and unstructured flowcharts.

Theorem 2 (Injectivization of Classical Flowcharts). *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. As shown above, the irreversibility of classical flowchart is due to irreversible steps (non-injective transformations) and join points without assertions. To translate a classical flowchart into a reversible flowchart, the store will be extended with a *history stack* `h` to record the information required to reconstruct the previous computation state. The stack is associated with the two standard operations `push` and `pop`, which are inverse to each other. The operation `top` is used to check the top element of the stack. A join point is injectivized as follows.

[3] A classical computation example is *lossless* audio codecs. Every operation on a quantum state in a quantum computer must be *unitary*, and therefore injective.



The injectivization of steps is similar: assume that every step is an assignment x `:=` e, which overwrites x with the value of expression e. Replace every step with one computing `push` x `h;` x `^=` e, which saves the original value of x on `h`. The resulting reversible flowchart F_h is an injectivized version of F. □

Corollary 1 (Input Embedding). *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 injectivized reversible flowchart F_h computing $f_h : x \mapsto (f(x), h)$, where h is the garbage (history) induced by Thm. 2. (2) Invert F_h to obtain F_h^{-1} which computes $f_h^{-1} : (f(x), h) \mapsto x$. (3) Construct F_i, which executes F_h, copies the values of all output variables into 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 variables used by F_h. Flowchart F_i returns both the original input and the output of executing flowchart F, and therefore computes f_i. □

RTMs are usually defined using quadruple rules [2,16], instead of the more common quintuple rules. A quadruple TM is defined by a finite set of states Q, a finite set of symbols S, and a finite set of symbol rules $\langle q_1, s_1, s_2, q_2\rangle$ and shift rules $\langle q_1, /, d, q_2\rangle$. A symbol rules says that in state q_1 with the tape head reading symbol s_1, write s_2 and change into state q_2. A shift rule says that in state q_1, move the tape head in the direction $d \in \{-, 0, +\}$ (left, stay, right) and change into state q_2. For a TM to be *reversible* there must be both forward determinism (in the usual sense) and backward determinism. A quadruple TM is *backward deterministic* iff for any pair of distinct quadruples $\langle q_1, t_1, t_2, q_2\rangle$ and $\langle q_1', t_1', t_2', q_2'\rangle$, if $q_2 = q_2'$ then $t_1 \neq /$, $t_1' \neq /$ and $t_2 \neq t_2'$.

Theorem 3 (*r*-Turing completeness). *Any reversible Turing machine can be simulated* cleanly *(without added garbage) by reversible flowcharts.*

Proof. The configuration of a Turing machine can be simulated as follows. `q` is a variable whose value is the current state, `s` holds the symbol under the tape head and `l` and `r` are stacks holding the left and right portions of the tape relative to the tape head, respectively.[4]

For a given RTM, assume that q_s and q_f are the start and finish states, respectively, and 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. 8. The helper function Q_{q_1,q_2} consist of the

[4] For convenience we assume the stacks are infinitely deep. Finite stacks will work as well, although care must be taken to maintain reversibility.

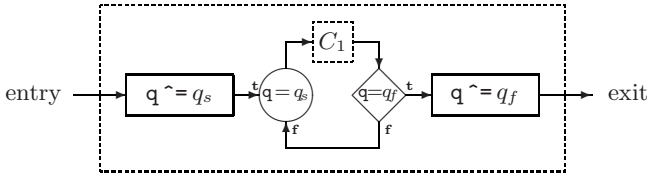


Fig. 7. Flowchart C_0 with main loop for RTM simulation

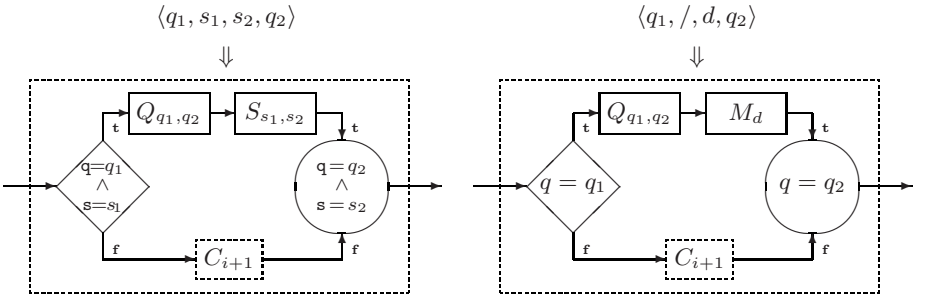


Fig. 8. Translation of RTM transition rules R_i into reversible flowcharts C_i

step `q ^= `q_1 `; q ^= `q_2. This changes `q`'s value from q_1 to q_2 reversibly, simulating 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`. The step M_d simulates moving the head in direction d. For example, M_+ is defined (reversibly) as `push s l; pop s r`. In the translation of both rule types, if rule R_i did not apply (enforced by the test predicate), control flows into C_{i+1}, the translation of rule R_{i+1}. Upon return from C_{i+1}, backward determinism of the RTM ensures that the given assertions are sufficient to differentiate between the two cases. In the translation of the final rule R_n, a dummy step is inserted in place of C_{n+1}. Execution of C_1 thus simulates the application of exactly the rule implied by the (simulated) configuration of the RTM.

C_1 can be embedded in a reversible loop C_0 that executes C_1 repeatedly, starting in state q_s until the final state q_f is reached (Fig. 7). C_0 thus computes the same function as the RTM, without the generation of garbage data. □

5 Reversible Flowchart Programming Languages

The reversible flowchart computation model provides a theoretical justification for low-level unstructured *machine code* (*e.g.*, for a reversible microprocessor) as well as for high-level *block-structured reversible languages*. We give two programming languages as examples for both types and illustrate them with a garbage-free implementation of a lossless encoder for permutations given by Dijkstra [8].

Grammar of reversible language *RL*

$p ::= b^+$ $\quad k ::=$ `from` l; $\quad j ::=$ `goto` l;
$b ::= l$: $k\ a^*\ j$ $\quad \mid$ `if` e `from` l `else` l; $\quad \mid$ `if` e `goto` l `else` l;
$a ::= x$ `^=` e; $\quad \mid$ `entry`; $\quad \mid$ `exit`;

Grammar of structured reversible language *SRL* **Expressions**

$p ::= b$ $\quad b ::= a$ $\quad e ::= c \mid x \mid o\ e \cdots e$
$a ::= x$ `^=` e; $\quad \mid\ b\ b$ $\quad o ::=$ `+` $\mid$ `*` $\mid \cdots$
$\mid$ `if` e `then` b `else` b `fi` e
$\mid$ `from` e `do` b `loop` b `until` e

Syntax Domains

$p \in$ Prog $\quad a \in$ Assign $\quad j \in$ Jump $\quad k \in$ From $\quad l \in$ Label
$b \in$ BasicBlock $\quad e \in$ Expr $\quad c \in$ Const $\quad x \in$ Var $\quad o \in$ Op

Fig. 9. A family of reversible flowchart languages

The encoder implements an injective function. The decoder can be obtained from the encoder using the straightforward inversion of Sec. 2, and vice versa.

Unstructured Reversible Language. A program written in the unstructured reversible language *RL* is a sequence of basic blocks. A block consists of a label, an unconventional *from* construct, a sequence of assignments, and a jump. A jump may be unconditional (`goto` l), conditional (`if` e `goto` l_1 `else` l_2), or the exit from the program (`exit`). The values of all variables are initially zero. The syntax is shown in Fig. 9.

An assignment is a C-like exclusive-or assignment (x `^=` e), where variable x must not occur in expression e. This syntactic constraint makes the assignment self-inverse. In general, any reversible update can be used as assignment operator to the language (*e.g.*, the C-like assignment operators `+=` and `-=`; see Sec. 2).

A from construct is an *unconditional assertion* (`from` l) that the control flow always comes from block l, a *conditional assertion* (`if` e `from` l_1 `else` l_2) that the control flow comes from block l_1 when predicate e is true and from block l_2 otherwise, or the entry of the program (`entry`). This construct makes the control flow of programs backward deterministic. Well-formed programs contain exactly one `entry` and one `exit`.

Structured Language. A program written in the structured reversible language *SRL* consists of one, possibly nested block. A block is an assignment, 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). They textually represent the reversible structured CFOs of Fig. 3. The syntax is shown in Fig. 9, with operational semantics rules omitted for space reasons.

Example: permutation-to-code. Consider the problem of translating an array `x[]` of length n, containing a permutation of the numbers $0, \ldots, n-1$, into an array where each index entry i counts the number of elements in `x[]` *smaller* than `x[`i`]` *preceding* the occurrence `x[`i`]` in `x[]`. For example, given the input permutation `x[]={2,0,3,1,5,4}`, we obtain the encoded array `x[]={0,0,2,1,4,4}`.

This is a fine program inversion example described by Dijkstra, who used an irreversible guarded commands language to write this program [8,11].

The structured reversible program in SRL is shown below in the left column. The right column shows the inverted program: a decoder that reconstructs the original permutation. Note that the encoder and decoder take the same number of steps on the corresponding input/output and that their space consumption is identical (due to the assumption that atomic step `+=` and its inverse `-=` consume equal execution time and space). For simplicity, we use the reversible update operators `+=` and `-=`, which can be simulated by `^=` and auxiliary variables.

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

The same program can be expressed in the unstructured reversible language RL. The program can easily be inverted (omitted due to lack of space.)

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

Admittedly, both RL and SRL are small reversible programming languages. Their purpose is theoretical: to model unstructured control flow of low-level reversible machine code with jumps and register updates, high-level reversible languages with structured control flow and assignments and the clean translation and interpretation of these languages within the reversible computing paradigm (*e.g.*, garbage-free reversible self-interpretation [20]).

6 Conclusion

We introduced the concept of reversible flowcharts, and showed that structured and unstructured reversible flowcharts are equally expressive. We demonstrated an injection of classical flowcharts, and proved the r-Turing completeness of

reversible flowcharts. The work presented here is part of a larger effort on the development of reversible programming systems, *e.g.* [1,7,9,19,20]. The results of this paper can be guidelines in designing new structured and unstructured reversible programming languages, independent of actual implementation.

Acknowledgments. An abstract of this paper was presented at the informal, unrefereed 18th Nordic Workshop on Programming Theory, 2006. Part of this work was supported by CREST, JST; and the FIRST research school.

References

1. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Computer Science - Theory and Applications. Proceedings. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007)
2. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. 17(6), 525–532 (1973)
3. Bennett, C.H.: Time/space trade-offs for reversible computation. SIAM J. Comput. 18(4), 766–776 (1989)
4. Böhm, C., Jacopini, G.: Flow diagrams, Turing machines and languages with only two formation rules. Communications of the ACM 9(5), 366–371 (1966)
5. Cooper, D.C.: Böhm and Jacopini's reduction of flow charts. Communications of the ACM 10(8), 463–473 (1967)
6. Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R. (eds.): Structured Programming. Academic Press, London (1972)
7. De Vos, A., Van Rentergem, Y.: Reversible computing: from mathematical group theory to electronical circuit experiment. In: 2nd Conf. on Computing Frontiers, pp. 35–44. ACM Press, New York (2005)
8. Dijkstra, E.W.: Program inversion. In: Bauer, F.L., Broy, M. (eds.) Program Construction: Intl. Summer School. LNCS, vol. 69, pp. 54–57. Springer, Heidelberg (1978)
9. Frank, M.P.: Reversibility for Efficient Computing. PhD thesis. MIT, Cambridge (1999)
10. Gomard, C.K., Jones, N.D.: Compiler generation by partial evaluation: a case study. Structured Programming 12, 123–144 (1991)
11. Gries, D.: The Science of Programming, ch.21: Inverting Programs, Texts and Monographs in Computer Science. Springer, Heidelberg (1981)
12. Hatcliff, J.: An introduction to online and offline partial evaluation using a simple flowchart language. In: Hatcliff, J., Mogensen, T., Thiemann, P. (eds.) Partial Evaluation. Practice and Theory. LNCS, vol. 1706, pp. 20–82. Springer, Heidelberg (1999)
13. Jacopini, G., Mentrasti, P., Sontacchi, G.: Reversible Turing machines and polynomial time reversibly computable functions. SIAM Journal on Discrete Mathematics 3(2), 241–254 (1990)
14. Lutz, C.: Janus: a time-reversible language. Letter written to Landauer, R. (1986), `http://www.cise.ufl.edu/~mpf/rc/janus.html`
15. Manna, Z.: Mathematical Theory of Computation. McGraw-Hill, New York (1974)
16. Morita, K., Yamaguchi, Y.: A universal reversible Turing machine. In: Durand-Lose, J., Margenstern, M. (eds.) Machines, Computations, and Universality. Proceedings. LNCS, vol. 4664, pp. 90–98. Springer, Heidelberg (2007)

17. Munakata, T.: Beyond silicon: New computing paradigms. Special issue. Communications of the ACM 50(9), 30–72 (2007)
18. Toffoli, T.: Computation and construction universality of reversible cellular automata. J. Comput. Sys. Sci. 15, 213–231 (1977)
19. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: 5th Conf. on Computing Frontiers, pp. 43–54. ACM Press, New York (2008)
20. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation. Proceedings, pp. 144–153. ACM Press, New York (2007)