Formalizing Reversible Computations for Synchronous Dataflow Languages with Infinite Lists

Sosuke Moriguchi*, Satoshi Takimoto, Mizuki Shirai, and Takuo Watanabe

Department of Computer Science, Institute of Science Tokyo, Tokyo, Japan,
chiguri@acm.org, takimoto@psg.c.titech.ac.jp, shirai@psg.c.titech.ac.jp,
takuo@acm.org

Abstract. Computational systems that deal with discrete time, such as stream computations and synchronous data flow languages, can be modeled using lists. However, most list operations are on finite lists, and it is not easy to define them for infinite lists to express persistent behavior. In particular, when using theorem provers, intuitive definitions are unacceptable due to restrictions on the handling of infinities. When integrating computational failures into the system, existing research either directly expresses failures or utilizes a mechanism that continues to send invalid values indefinitely. Still, the latter cannot determine failures in terms of computation. In this study, we formalize a model of a reversible synchronous dataflow language using finite and infinite lists and show that the semantics of each correspond to each other using the Coq theorem prover. For infinite lists, the inconsistency that arises as a result of incorporating failures into the list is resolved using finite lookahead.

Keywords: synchronous dataflow language, theorem proving, reversible computation, coinductive definition

1 INTRODUCTION

Computational systems that use discrete time, such as stream computations and synchronous dataflow languages [1], are formalized using lists. For example, in [2], streams are treated as infinite lists. In our previous research, we employ lists as a representation of sequences of data in synchronous dataflow and functional reactive programming languages [12].

Although streams and lists have similar structures, they have different properties in several respects. In particular, streams often deal with infinite sequences of objects, while lists deal with finite ones. In general, definitions by lists use procedures such as append and zip. On the other hand, a definition by a stream provides restrictions such as the fact that an element can be computed from only its previous inputs.

In our formalization of the synchronous dataflow language, the system is represented as a partial function between lists to handle reversible computation. In general, reversible computation uses partial functions in the semantics to prevent

J. Caro et al. (eds.), Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2024) , Atlantis Highlights in Computer Sciences 23,
https://doi.org/10.2991/978-94-6463-684-0_2

the use of irreversible processing. For example, in RFun [14] and CoreFun [5], the right-hand sides of pattern matching are also considered to be patterns so that pattern matching can be traced back. However, the backward pattern is not necessarily exhaustive, and if there is no corresponding one, it gets stuck. In this formalization, the discrete-time progression is represented by the order in the list. Still, for a sequence containing uncomputable data (due to irreversibility), the system determines that the sequence is not computable. Thus, in the formalization, computability is determined only after all elements of the list have been processed.

Synchronous dataflow and functional reactive programming languages make it difficult to determine an upper bound on the length of an input sequence because of the continuous process. Therefore, extending this formalization to infinite lists makes the representation of the system more appropriate.

Naively, it does not seem too difficult to extend the definition by list to allow for infinite lengths. However, allowing infinite length restricts the following operations.

- To make some judgment on the entire list. For example, to determine the equivalence of two lists, to determine whether they are sublists, etc.
- To change the behavior of a function that produces a list, depending on the result for the rest. An example is the inversion of a list.

The validity of these restrictions is obvious when considering the actual execution on a computer since calculations for infinite steps are required. These restrictions may make the existing formalization itself difficult. For example, our formalization falls under the latter restriction because it determines computability based on the entire list.

In this study, we reformalize our formalization of reversible computation using the Coq theorem prover, prove its properties, and extend it to infinite lists. We represent a sequence of data as a finite list on Coq, and a partial function as a function whose return type is `option` (returning `None` when it is not computable and `Some` when it is computable). The definition is then modified based on the above restrictions to support infinite lists on Coq. We show that the modification is a natural extension to infinite lists by showing that the modified formalization keeps the results in finite lists with the original formalization. The contributions of this study are in providing mechanized verification of our formalization and in analyzing its extension to infinite lists as a case study. The formalization presented in this paper is available at https://github.com/psg-titech/IIST_proof.

The paper is organized as follows. In Section 2 we introduce our previous formalization of a reversible synchronous computation using the notation of Coq theorem prover. In Section 3 we re-formalize the computation with coinductive (possibly infinite) lists. Comparisons with related studies are made in Section 4 and the paper is summarized in Section 5.

2 REVERSIBLE SYNCHRONOUS DATAFLOW LANGUAGE

Here, we introduce the background of this research, synchronous dataflow language, and its reversible computations. Note that the notation of the formalization here is that of the theorem prover Coq.

2.1 Synchronous Dataflow Language

A synchronous dataflow language is a programming language that describes a system as a dataflow, taking as an input sequence of data generated by discrete time and obtaining a sequence of data as the output of the system. Typical synchronous dataflow languages include Lustre [3] and Signal [7]. A language that focuses more on synchronization is the functional reactive programming language. In particular, Emfrp [11], a functional reactive programming language for small-scale embedded systems, is strongly influenced by synchronous dataflow languages.

We formalized the dataflow in discrete time in these languages using lists [12]. In this formalization, discrete occurrences of data are represented as a list, with the later elements appearing later in the list. However, while this list implies a temporal order, it does not mean a specific time interval. Therefore, it cannot be expressed for time-dependent dataflows, such as a program that delays intermittent input for 5 seconds.

A dataflow program transforms a sequence of data on the input into a sequence of data on the output. We assume that program execution can fail (this is a requirement by reversible computation, which is described in Section 2.2). In this formalization, a dataflow program from data type `X` to data type `Y` is defined as a partial function that transfers a sequence of data in `X` to a sequence of data in `Y`. Specifically, the dataflow program is defined as follows.

```
1 list X -> option (list Y)
```

In case of success, the list is returned with `Some`, and in case of failure, `None`. For example, the type of a system that takes two natural numbers and returns a natural number and a boolean value would be `list (nat * nat) -> option (list (nat * bool))`. However, arbitrary functions are not allowed, and the following two conditions are required to represent a system that computes sequentially.

1. If the function fails (returns `None`) for a given input, then for the extended input (by appending backward list) it will fail.
2. If the function succeeds for a given input and outputs a list of `Y`, then for the extended input the output will also be extended, or it will fail.

These conditions indicate, for example, that for a given successful input, the prefix of the input (the former part of the list) will also succeed.

2.2 Reversible Computation on Synchronous Dataflow Language

Reversible computation is a computation in which the input can be derived from the output. In this computation, the procedural language [6] allows for reversion to the original state by traversing the statement in reverse, and the functional languages [5, 13] allow for the generation of arguments from the result in a function call. These inversions recover the data uniquely, and general programming languages do not satisfy the injectivity required by these inverse operations. On the other hand, because of injectivity, these reversible languages have restricted processing or functions are partial. Tracing backward to a state or argument with no backward destination is uncomputable.

Inverse traversal functions appear to be time retrogression, but when applied directly to discrete-time in a synchronous dataflow language, the following points become problematic.

- The process of obtaining input from output is in reverse order in time. Therefore, the retrogression requires a state at the end of the input. On the other hand, in many dataflow languages, the initial state is fixed. In other words, the reverse process, in which the initial state to be sought changes depending on the assumed input, cannot be described in a dataflow language.
- If we match the sequence of data in time, the process of obtaining one output datum from one input datum (the system is in the post-output state) and the process of obtaining one input datum from the one output datum (the system is in the pre-output state). The internal states do not match.

To solve these problems, we proposed a new definition of the systems as pairs of two functions. The pair of functions f and g satisfying the two conditions in Section 2.1 is defined as a reversible computation of synchronous dataflow language if for a sequence of data l in the domain of f, $f(l)$ is in the domain of g and $g(f(l))$ is a prefix of l.

However, since there is a function that always returns an empty sequence as such g, we would like to discuss a subclass in which g includes nontrivial ones. We defined such a subclass, called (d, d')-*FIRST* (Fixed-delay Incrementally Reversible Sequence Transformation), where the difference in length between the input sequence and the output sequence obtained by f is a natural number d and the difference in length between the output sequence and the input sequence obtained by g is a natural number d'. The d and d' are called the forward and backward *delay numbers*, respectively, Also, when the delay numbers of (d, d')-FIRST are less important, they are referred to as *FIRST*. Intuitively, it represents a system in which the first few inputs are used for initialization and do not return a response. Then, the system continues to output in response to the inputs.

2.3 Constructing Reversible Compuation

FIRST describes a system as a pair of functions that satisfy the above conditions. However, the definition still has problems in a system description.

- Although the formalization describes the system as a function of lists to lists, there is a large divergence from the actual system, which responds to inputs and produces outputs one after another.
- The relationship to general reversible computation (or reversible function) is not clear.

Therefore, it is desirable to describe FIRST in terms of components with simpler behavior. We attempted to solve these problems by following the form of reversible logic elements that simulate reversible cellular automata and reversible Turing machines [10].

We proposed a representation for constructing FIRST, called *FIRSD* (Fixed-delay Incrementally Reversible Synchronous Dataflow). This representation is a tree structure consisting of the following five types of nodes.

- mapfold(f, g, a) : f is a partial reversible function $A \to X \rightleftharpoons Y$, g is a function $A \to X \to A$, and a is an element of type A. The f takes the value of A and forms a partially reversible function between X and Y. The a is the initial value and the g is its update function as an accumulator. The next state of the accumulator is obtained by the application of the input and a to g. This accumulator is used as the first element of f to change the behavior.
- delay(x) : The flow is composed with x added at the beginning and the tail removed.
- hasten(x) : The flow is composed of a flow whose head is x and whose head is deleted. If the head is not x, the calculation fails. (In the inverse direction, delay(x) and the roles are swapped.)
- $E_1 \ggg E_2$: Configure a dataflow that applies FIRSD E_1 followed by FIRSD E_2.
- $E_1 \otimes E_2$: With the inputs as a sequence of data pairs, apply FIRSD E_1 and E_2 to each of the separations, and construct a dataflow that ties the resulting flows together. The length should be aligned to the shorter one.

For the FIRSD defined above, the interpretation functions are given in Listing 1. Each `fwd` and `bwd` is an interpretation that obtains an output from an input (forward) and an input from an output (backward), respectively. In this context, `<~~>` represents a reversible function, and `forward` yields a left-to-right partial function and `backward` a right-to-left partial function.

Each interpretation function is processed on the input (or output) sequence from the beginning, which is closer to the implementation of the system than FIRST, which is simply a partial function between lists. In addition, this interpretation function has the following properties.

1. The forward and backward interpretation functions yield a pair of functions such that it is a reversible computation of synchronous dataflow language.
2. The difference between the input and output lengths obtained from the interpretation function is a fixed value. Combined with the previous section, FIRST is obtained from the interpretation functions with these differences as the number of delays. Note that the delay numbers are calculated from the structure of FIRSD (Listing 2).

```
Definition option_bind {A B : Type} (a : option A)
                       (f : A -> option B) : option B :=
  match a with  | Some a => f a  | None => None  end.
Fixpoint zip {A B : Type} (l : list A) (r : list B) : list (A * B) :=
  match l, r with  | nil, _ | _, nil => nil
                   | a :: l, b :: r => (a, b) :: zip l r  end.
(* Forward/Backward Interpretation of mapfold *)
Fixpoint f_mapfold {A X Y : Type} (f : A -> X <~~> Y)
                   (g : A -> X -> A) (a : A) xs : option (list Y) :=
  match xs with
  | nil => Some nil
  | x :: xs' =>
     option_bind ((forward X Y (f a)) x)
                 (fun y => option_bind (f_mapfold f g (g a x) xs')
                                       (fun ys => Some (cons y ys)))
  end.
Fixpoint b_mapfold {A X Y : Type} (f : A -> X <~~> Y)
                   (g : A -> X -> A) (a : A) ys : option (list X) :=
  match ys with
  | nil => Some nil
  | y :: ys' =>
     option_bind ((backward X Y (f a)) y)
                 (fun x => option_bind (b_mapfold f g (g a x) ys')
                                       (fun xs => Some (cons x xs)))
  end.
(* Forward/Backward Interpretation of FIRSD *)
Fixpoint fwd {X Y : Type} (e : FIRSD X Y)
   : list X -> option (list Y) :=
  match e with
  | mapfold a f g => f_mapfold f g a
  | delay x => fun xs => Some (slide x xs)
  | hasten x => fun xs =>
     match xs with
     | nil => Some nil
     | x' :: xs' => if equiv_dec x x' then Some xs' else None
     end
  | xz ⋙ zy => fun xs => option_bind (fwd xz xs) (fwd zy)
  | xy1 ⊗ xy2 => fun x12s =>
     let (x1s, x2s) := split x12s in
     option_bind (fwd xy1 x1s)
            (fun y1s => option_bind (fwd xy2 x2s)
                                    (fun y2s => Some (zip y1s y2s)))
  end.
Fixpoint bwd {X Y : Type} (e : FIRSD X Y)
   : list Y -> option (list X) :=
  match e with
  | mapfold a f g => b_mapfold f g a
  | delay x => fun ys =>
     match ys with
     | nil => Some nil
     | y' :: ys' => if equiv_dec x y' then Some ys' else None
     end
  | hasten x => fun ys => Some (slide x ys)
  | xz ⋙ zy => fun ys => option_bind (bwd zy ys) (bwd xz)
  | xy1 ⊗ xy2 xy1 xy2 => fun y12s =>
     let (y1s, y2s) := split y12s in
     option_bind (bwd xy1 y1s)
            (fun x1s => option_bind (bwd xy2 y2s)
                                    (fun x2s => Some (zip x1s x2s)))
end.
```

Listing 1. Forward/Backward Interpretations of FIRSD.

```
Fixpoint delay_fwd {X Y : Type} (e : FIRSD X Y) : nat :=
  match e with
  | mapfold _ _ _ => 0
  | delay _ => 0
  | hasten _ => 1
  | xz ⋙ zy => delay_fwd xz + delay_fwd zy
  | xy1 ⊗ xy2 => max (delay_fwd xy1) (delay_fwd xy2)
  end.

Fixpoint delay_bwd {X Y : Type} (e : FIRSD X Y) : nat :=
  match e with
  | mapfold _ _ _ => 0
  | delay _ => 1
  | hasten _ => 0
  | xz ⋙ zy => delay_bwd xz + delay_bwd zy
  | xy1 ⊗ xy2 => max (delay_bwd xy1) (delay_bwd xy2)
  end.
```

Listing 2. Forward/Backward delay numbers of FIRSD.

3. For any (d, d')-FIRST, there exists a FIRSD that the interpretation functions yield the FIRST.

The proofs are performed using structural induction on FIRSD (and separately for mapfold, induction on lists).

From the above properties, it is shown that FIRSD is sufficient to construct any FIRST. Since the only parameter that affects the result of the computation is mapfold, and the function used here is a partially reversible function using the function for the accumulator and its values, the reversible computation of the synchronous dataflow language can be reduced to a partially reversible function between the elements.

3 DATAFLOW AS INFINITE LISTS

3.1 Infinite Lists with Failures

In this section, we modify the formalization in the previous section and formulate it again with possibly infinite lists. However, several essential problems arise in the formalization. A major one is the representation of the failure in the interpretation function. In the finite list, the method is to "return `None` if the remaining computation fails" (even if it has succeeded up to the present). As a result, it could be expressed by a combination of list and `option`. However, since it is impossible to "calculate all remaining elements" in an infinite list, calculation using the result is also impossible.

Therefore, we use a (possibly infinite) list including failures, as follows.

```
CoInductive colist (A : Type) : Type :=
| conil : colist A
| cocons : A -> colist A -> colist A
| cofail : colist A.
```

The command `CoInductive` defines the coinductive definition (the greatest fixed point of the recursive definition). The `colist` is a list that terminates with `cofail` if the computation fails and is otherwise intuitively similar to `list`, except that infinite sequences are also allowed.

Unlike the representation in the finite list, the elements of the successful steps remain after the failure. For example, the case of failure following the outputs of 1, 2, and 3 is represented by `None` in the finite list, but in `colist` it is represented by data combining 1, 2, 3 and `cofail` with `cocons`. In Coq, success is specified as follows.

```
CoInductive failless_colist {A : Type} : colist A -> Prop :=
| flcnil : failless_colist conil
| flccons : forall a l, failless_colist l
             -> failless_colist (cocons a l).
```

Due to the limitations of coinductive definitions in Coq, it is possible to describe the above property as being "connected except for failure" (i.e., not failing), but it is impossible to determine this property by computation.

3.2 Interpretation Functions

In this section, we discuss the function that interprets FIRSD defined in Section 2.3 by `colist`. The fundamental idea is to express error propagation by immediately returning a failure if one appears. The forward interpretations for mapfold and FIRSD are described as the following functions.

```
CoFixpoint cof_mapfold {A X Y : Type} (f : A -> X <~~> Y)
             (g : A -> X -> A) (a : A) (xs : colist X) : colist Y :=
 match xs with
 | conil => conil | cofail => cofail
 | cocons x xs =>
   match (forward X Y (f a)) x with
   | None => cofail | Some y => cocons y (cof_mapfold f g (g a x) xs)
   end
 end.
Fixpoint cofwd {X Y : Type} (e : FIRSD X Y) : colist X -> colist Y :=
 match e with
 | mapfold a f g => cof_mapfold f g a
 | delay x => coslide x
 | hasten x => fun xs =>
   match xs with
   | conil => conil | cofail => cofail
   | cocons x' xs => if equiv_dec x x' then xs else cofail
   end
 | xz ⋙ zy => fun xs => cofwd zy (cofwd xz xs)
 | xy1 ⊗ xy2 => fun x12s =>
   cozip (drop_tl_n (delay_fwd xy2 - delay_fwd xy1)
                    (cofwd xy1 (coleft  x12s)))
         (drop_tl_n (delay_fwd xy1 - delay_fwd xy2)
                    (cofwd xy2 (coright x12s)))
 end.
```

The function for mapfold is almost the same as the definition for `list` with only minor differences such as the keywords and the name of the function. On

the other hand, there is a major difference compared to `fwd` for finite lists in the case of ⊗. The `zip` function for `colist`, `cozip`, is defined as follows.

```
1 CoFixpoint cozip {A B : Type} (la : colist A) (lb : colist B) :
      colist (A*B) :=
2   match la, lb with
3   | cofail, _ | _, cofail => cofail
4   | conil, _ | _, conil => conil
5   | cocons a la, cocons b lb => cocons (a, b) (cozip la lb)
6   end.
```

The difference with `zip` is that if one fails, all failures take the form of failures from there. In the case of a finite list, the failures are handled in a different data type, but in this function, they are embedded.

Here, consider FIRSD, which connects the following two processes in parallel (by ⊗).

- If the value is less than 5, the process returns that value, otherwise, it is undefined.
- If the first value in the flow is 1 and the second is 2, the process returns the following flow, otherwise, it is undefined.

The former can be constructed with mapfold and the latter with hasten(1) ⋙ hasten(2). If a list from 1 to 5 is given to each process, the former will fail at the end of the list from 1 to 4. In the latter, a list from 3 to 5 appears, ending with success. When these are combined as they are, `(1,3)`, `(2, 4)`, and `(3, 5)` appear and are judged to be successful. However, since one of these flows contains a failure, the whole flow must contain a failure somewhere. The forward interpretation function for finite lists returns `None` for this. Thus, this behavior is completely different from that of a finite list. On the other hand, as already mentioned, it is computationally infeasible to determine if there is a failure anywhere in each other when merging.

We solve this problem by introducing a process to "align" their lengths with each other before joining them with `cozip`, and by representing failures upfront in this process. As a result, the interpretation for ⊗ calls `drop_tl_n`, which erases the tail, as in lines 21 and 23. This function looks ahead to the second argument as many times as the first argument, discards the remainder if there is a terminator there, assumes failure if there is a failure, and recurses to target one follow if it continues more.

The "alignment" we propose means that both sides are aligned with `conil` when they end successfully. In contrast, there are multiple behaviors for failure (`cofail`).

1. Both fail at the same time. In other words, if one is a `cofail`, the other is also a `cofail`.
2. Only one side can fail first.
3. Both can fail at any time.

As an example of 1, two `colist`s extracted the left and right elements from the `colist` of pairs by `coleft` and `coright`. If the original `colist` contains any

failures, then both the left and right elements will fail at the corresponding points.

An example of the second case is the relationship between the input `colist` whose last elements are removed by `drop_tl_n` and the output of the `cofwd` . Intuitively, the difference in length on success is the number of delays, and failure in the input means failure in the output. Since the failure in the output may occur due to computation, the failure in the output does not immediately mean failure in the input, so this is not an example of 1.

3 is a looser property than 1 and 2, and there are no restrictions on failure. It is intended to express that `cozip` does not truncate `cofail`. By combining 1 and 2, it can be shown that the interpretation of $\otimes$ does not truncate `cofail`.

Using them, the proof of the (intuitively obvious but nontrivial) proposition that if there is no failure in the output, then there is no failure in the input requires about 1200 lines. This enables us to show that failures are not hidden by `cozip` and that if there is no failure in an output, there is also no failure in the respective output.

3.3 Correspondence between Two Formalizations

To correspond to the interpretation functions defined for each of the finite and infinite lists, a predicate is defined to represent the correspondence between the lists, including failures.

```
Inductive CorrListColist {A : Type}
  : option (list A) -> colist A -> Prop :=
| clcsnil : CorrListColist (Some nil) conil
| clcscons : forall a l cl, CorrListColist (Some l) cl
                  -> CorrListColist (Some (cons a l)) (cocons a cl)
| clcnfail : CorrListColist None cofail
| clcncons : forall a cl, CorrListColist None cl
                  -> CorrListColist None (cocons a cl).
```

This is a predicate that corresponds a finite list to a finite `colist` (with the same elements in the same order), or a failure by `None` to a `colist` containing `cofail`. Note that the correspondence cannot be described by a function because a function can be defined from a finite list to `colist`, but not vice versa.

Using the correspondence, the following properties can be described and verified.

```
Theorem fwd_cofwd_correspond :
 forall {X Y : Type} (l : list X) (cl : colist X),
   CorrListColist (Some l) cl
    -> forall (e : FIRSD X Y), CorrListColist (fwd e l) (cofwd e cl).
```

This means that for each corresponding list, the interpretation function yields the corresponding list (including failures). The proof requires multiple lemmas, and the entire formalization and proof of the correspondence is about 650 lines.

3.4 Hard-to-Prove Properties

So far, we have shown that the formulation with `colist` is feasible and that it is an extension of that of the finite list. However, not all properties that are verifiable with a finite list are also verifiable. For example, the following property is hard to prove.

```
Theorem cofwd_failless_inverse :
 forall {X Y : Type} (e : FIRSD X Y) xs ys,
  failless_coprefix ys (cofwd e xs)
  -> exists xs', cofwd e xs' = ys /\ failless_coprefix xs' xs.
```

This property is intuitively obvious: for a sequence obtained by forward interpretation, if we take out a prefix that does not contain a failure, there exists an input prefix that returns it. The formalization with finite lists easily proves the corresponding property (as mentioned in Section 2.1). The property requires the contents of `xs` and `ys` to construct `xs'` which is a prefix of `xs`. However, a direct proof is difficult because recursion to `colist` (coinductively defined types) is not allowed for existentially-quantified propositions (inductively defined types). A possible way to prove this property is to define the construction method of `xs'` as a function, but the actual proof of the property is left for future work.

4 RELATED WORKS

There are two major ways to indicate failure in verifying against an infinite stream.

- Introduce a representation of the failure as the end of streams [4].
- Use a stream that returns a value that implies an infinite number of failures [2].

The former study axiomatizes formulations for a wide range of partial computations and treats streams as an example. In this study, failure is treated explicitly as a value, and in this respect, it is identical to our study. Our study uses Coq's built-in coinductive definition and does not need to consider a new system. We have also confirmed that a similar definition can be used in Agda, another theorem prover. In this respect, the fact that the analysis is performed using naive definitions is the contribution of this paper to this study. Note that the failure concealment in the case of finite length mentioned in Section 3 does not occur in this study because streams are considered. Whether concepts such as length adjustment can be handled as in our study is a subject for future research.

The latter method represents a partial computation by continuously outputting invalid values. The position of failure in this study is not merely a temporary problem, but a situation that cannot be answered computationally. If it is possible to recover from failure by computation, it should be returned as the result of each computation. Our position is that such an expression should be represented by an `option` type or a similar type in computation but not by

an invalid value in a framework for handling infinity. This allows the system to continue the computation and is distinguished from the system stopping the computation due to an error. On the other hand, this approach has the problem that it is difficult to semantically separate a temporary failure from an inability to compute since it is impossible to observe an infinite number of streams in a computation.

As a formalization for partial computation, there is a formalization for Sparcl, the language for partially reversible computation [9], in Agda [8]. The mapfold in our study utilizes a partially reversible computation and can be related to this formulation to discuss the descriptiveness of reversible computation of synchronous dataflow language.

5 CONCLUSIONS

In this paper, we described and verified the formalization of reversible computation for the synchronous dataflow language proposed in our previous study [12] in Coq, and further modified the formalization to allow for infinite lists. It differs significantly from the original formalization for finite lists, especially in the treatment of failures. In this modification, by making the targets infinite lists, including finite lists, we found and solved a problem that does not occur when streams are targeted.

In the future, we would like to discuss propositions that are difficult to prove directly (as described in Section 3.4), and compare the results with other formalization methods mentioned in the previous section. In addition, we would like to formalize not only FIRST, but also subclasses in which the number of delays changes within an upper bound at runtime, and extend the formalization with an infinite list.

Acknowledgements

This work was supported in part by JSPS KAKENHI Grant Numbers JP19K20245, JP21K11822, JP22K11967, and JP24K14892.

References

1. Bainomugisha, E., Carreton, A.L., Van Cutsem, T., Mostinckx, S., De Meuter, W.: A survey on reactive programming. ACM Computing Surveys 45(4), 52:1–52:34 (2013)
2. Capretta, V.: General recursion via coinductive types. Logical Methods in Computer Science Volume 1, Issue 2 (Jul 2005)
3. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: A declarative language for real-time programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 178–188. POPL '87, ACM, New York, NY, USA (1987)

4. Cheval, H., Nowak, D., Rusu, V.: Formal definitions and proofs for partial (co)recursive functions. Journal of Logical and Algebraic Methods in Programming 141, 100999 (2024)
5. Jacobsen, P.A.H., Kaarsgaard, R., Thomsen, M.K.: CoreFun: A typed functional reversible core language. In: Kari, J., Ulidowski, I. (eds.) Reversible Computation (RC2018). Lecture Notes in Computer Science, vol. 11106, pp. 304–321. Springer, Cham (2018)
6. James, R.P.: Theseus : A high level language for reversible computing (2014), wIP paper at RC 2014. https://legacy.cs.indiana.edu/~sabry/papers/theseus.pdf
7. LeGuernic, P., Gautier, T., Le Borgne, M., Le Maire, C.: Programming real-time applications with Signal. Proceedings of the IEEE 79(9), 1321–1336 (1991)
8. Matsuda, K.: A proof-of-concept implementation of sparcl in agda, in an intricately-typed style (2024), https://github.com/kztk-m/sparcl-agda
9. Matsuda, K., Wang, M.: Sparcl: A language for partially invertible computation. Journal of Functional Programming 34, e2 (2024)
10. Morita, K.: Reversible computing and cellular automata—a survey. Theoretical Computer Science 395(1), 101–131 (2008)
11. Sawada, K., Watanabe, T.: Emfrp: A functional reactive programming language for small-scale embedded systems. In: Companion Proceedings of the 15th International Conference on Modularity. pp. 36–44. MODULARITY Companion 2016, ACM, New York, NY, USA (2016)
12. Shirai, M., Moriguchi, S., Watanabe, T.: Construction of inverse computation in synchronous dataflow programming. Computer Software 41(3), 3_34–3_40 (2024)
13. Thomsen, M.K., Axelsen, H.B.: Interpretation and programming of the reversible functional language RFUN. In: Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages. IFL '15, ACM, New York, NY, USA (2015)
14. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) Reversible Computation (RC2011). Lecture Notes in Computer Science, vol. 7165, pp. 14–29. Springer, Berlin, Heidelberg (2012)