Abstract

Invertibility is a fundamental concept in computer science, with various manifestations in software development (serializer/deserializer, parser/printer, redo/undo, compressor/decompressor, and so on). Full invertibility necessarily requires bijectivity, but the direct approach of composing bijective functions to develop invertible programs is too restrictive to be useful. In this paper, we take a different approach by focusing on partially invertible functions—functions that become invertible if some of their arguments are fixed. The simplest example of such is addition, which becomes invertible when fixing one of the operands. More involved examples include entropy-based compression methods (e.g., Huffman coding), which carry the occurrence frequency of input symbols (in certain formats such as Huffman tree), and fixing this frequency information makes the compression methods invertible.

We develop a language SPARCL for programming such functions in a natural way, where partial invertibility is the norm and bijectivity is a special case, hence gaining significant expressiveness without compromising correctness. The challenge in designing such a language is to allow ordinary programming (the “partially” part) to interact with the invertible part freely, and yet guarantee invertibility by construction. The language SPARCL is linear-typed and has a type constructor to distinguish data that are subject to invertible computation and those that are not. We present the syntax, type system, and semantics of the language and prove that SPARCL correctly guarantees invertibility for its programs. We demonstrate the expressiveness of SPARCL with examples including tree rebuilding from preorder and inorder traversals, Huffman coding, arithmetic coding, and LZ77 compression.

1 Introduction

Invertible computation, also known as reversible computation in physics and more hardware-oriented contexts, is a fundamental concept in computing. It involves computations that run both forward and backward so that the forward/backward semantics form a bijection. (In this paper, we do not concern ourselves with the totality of functions. We call a function a bijection if it is bijective on its actual domain and range, instead of its formal domain and codomain.) Early studies of invertible computation arise
from the effort to reduce heat dissipation caused by information-loss in the traditional (unidirectional) computation model (Landauer, 1961). More modern interpretations of the problem include software concerns that are not necessarily connected to the physical realization. Examples of such include developing pairs of programs that are each other’s inverses: serializer and deserializer (Kennedy & Vytiniotis, 2012), parser and printer (Rendel & Ostermann, 2010; Matsuda & Wang, 2013, 2018b), compressor and decompressor (Srivastava et al., 2011), and also auxiliary processes in other program transformations such as bidirectionalization (Matsuda et al., 2007).

Invertible (reversible) programming languages are languages that offer primitive support to invertible computations. Examples include Janus (Lutz, 1986; Yokoyama et al., 2008), Frank’s R (Frank, 1997), Ψ-Lisp (Baker, 1992), RFun (Yokoyama et al., 2011), Π/Πᵦ (James & Sabry, 2012), and Inv (Mu et al., 2004b). The basic idea of these programming languages is to support deterministic forward and backward computation by local inversion: if a forward execution issues (invertible) commands \(c_1\), \(c_2\), and \(c_3\) in this order, a backward execution issues corresponding inverse commands in the reverse order, as \(c_3^{-1}\), \(c_2^{-1}\), and \(c_1^{-1}\). This design has a strong connection to the underlying physical reversibility and is known to be able to achieve reversible Turing completeness (Bennett, 1973); i.e., all computable bijections can be defined.

However, this requirement of local invertibility does not always match how high-level programs are naturally expressed. As a concrete example, let us see the following toy program that computes the difference of two adjacent elements in a list, where the first element in the input list is kept in the output. For example, we have \(\text{subs} [1, 2, 5, 2, 3] = [1, 1, 3, -3, 1]\).

\[
\text{subs} :: [\text{Int}] \rightarrow [\text{Int}]
\text{subs} \; xs = \text{goSubs} \; 0 \; xs
\]

\[
\text{goSubs} :: \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}]
\text{goSubs} \; n \; (x : xs) = (x - n) : \text{goSubs} \; x \; xs
\]

Despite being simple, these kind of transformations are nevertheless useful. For example, a function similar to \(\text{subs}\) can be found in the preprocessing step of image compression algorithms such as those used for PNG.\(^1\) Another example is the encoding of bags (multisets) of integers, where \(\text{subs}\) can be used to convert sorted lists to lists of integers without any constraints (Kennedy & Vytiniotis, 2012).

The function \(\text{subs}\) is invertible. We can define its inverse as below.

\[
\text{subs}^{-1} :: [\text{Int}] \rightarrow [\text{Int}]
\text{subs}^{-1} \; ys = \text{goSubs}' \; 0 \; ys
\]

\[
\text{goSubs}' :: \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}]
\text{goSubs}' \; n \; (y : ys) = \text{let} \; x = y + n \; \text{in} \; x : \text{goSubs}' \; x \; ys
\]

However, \(\text{subs}\) cannot be expressed directly in existing reversible programming languages. The problem is that, though \(\text{subs}\) is perfectly invertible, its subcomponent \(\text{goSubs}\) is not

\(^1\) https://www.w3.org/TR/2003/REC-PNG-20031110/.
SPARCL: A language for partially invertible computation

(its first argument is discarded in the empty-list case, and thus the function is not injective). Similar problems are also common in adaptive compression algorithms, where the model (such as a Huffman tree or a dictionary) grows in the same way in both compression and decompression, and the encoding process itself is only invertible after fixing the model at the point.

In the neighboring research area of program inversion, which studies program transformation techniques that derive $f^{-1}$ from $f$’s definition, functions like goSubs are identified as partially invertible. Note that this notion of partiality is inspired by partial evaluation, and partial inversion (Romanenko, 1991; Nishida et al., 2005; Almendros-Jiménez & Vidal, 2006) allows static (or fixed) parameters whose values are known in inversion and therefore not required to be invertible (for example the first argument of goSubs). (To avoid potential confusion, in this paper, we avoid the use of “partial” when referring to totality, and use the phrase “not-necessarily-total” instead.) However, program inversion by itself does not readily give rise to a design of invertible programming language. Like most program transformations, program inversion may fail, and often for reasons that are not obvious to users. Indeed, the method by Nishida et al. (2005) fails for subs, and for some other methods (Almendros-Jiménez & Vidal, 2006; Kirkeby & Glück, 2019, 2020), success depends on the (heuristic) processing order of the expressions.

In this paper, we propose a novel programming language SPARCL that for the first time addresses the practical needs of partially invertible programs. The core idea of our proposal is based on a language design that allows invertible and conventional unidirectional computations, which are distinguished by types, to coexist and interact in a single definition. Specifically, inspired by Matsuda & Wang (2018c), our type system contains a special type constructor $(-)^*$ (pronounced as “invertible”), where $A^*$ represents $A$-typed values that are subject to invertible computation. However, having invertible types like $A^*$ only solves half of the problem. For the applications we consider, exemplified by subs, the unidirectional parts (the first argument of goSubs) may depend on the invertible part (the second argument of goSubs), which complicates the design. (This is the very reason why Nishida et al. (2005)’s partial inversion fails for subs. In other words, a binding-time analysis (Gomard & Jones, 1991) is not enough (Almendros-Jiménez & Vidal, 2006).) This interaction demands conversion from invertible values of type $A^*$ to ordinary ones of type $A$, which only becomes feasible when we leverage the fact that such values can be seen as static (in the sense of partial inversion (Almendros-Jiménez & Vidal, 2006)) if the values are already known in both forward and backward directions. The nature of reversibility suggests linearity or relevance (Walker, 2004), as discarding of inputs is intrinsically irreversible. In fact, reversible functional programming languages (Baker, 1992; Mu et al., 2004b; Yokoyama et al., 2011; James & Sabry, 2012; Matsuda & Wang, 2013) commonly assume a form of linearity or relevance, and in SPARCL this assumption is made explicit by a linear type system based on $\lambda^q$ (the core system of Linear Haskell (Bernardy et al., 2018)).

As a teaser, an invertible version of subs in SPARCL is shown in Figure 1. In SPARCL, invertible functions from $A$ to $B$ are represented as functions of type $A^* \rightarrow B^*$, where $\rightarrow$ is

---

2 The name stands for “a system for partially reversible computation with linear types”.

3 We use a Haskell-like syntax in this paper for readability, although our prototype implementation (https://github.com/kztk-m/sparcl) uses simple non-indentation-sensitive syntax that requires more keywords for parsing.
\[ \text{subs} : (\text{List}\ \text{Int})^* \rightarrow (\text{List}\ \text{Int})^* \]
\[ \text{subs}\ \text{xs} = \text{goSubs}\ 0\ \text{xs} \]

\[ \text{goSubs} : \text{Int} \rightarrow (\text{List}\ \text{Int})^* \rightarrow (\text{List}\ \text{Int})^* \]
\[ \text{goSubs}\ \text{Nil}^* = \text{Nil}^* \ \text{with}\ \text{null} \]
\[ \text{goSubs}\ n\ (\text{Cons}\ \text{xs})^* = \]
\[ \begin{aligned} &\text{let } (x, r) = \text{pin}\ x (\lambda x'. \text{goSubs}\ x'\ \text{xs}) \ 	ext{in} \quad \quad \quad \ -- x' : \text{Int} \text{ is a static version of } x : \text{Int}^*. \\
&\text{Cons}^* (\text{sub}\ n\ x)\ r\ \text{with}\ \text{not}\ \circ\ \text{null} \end{aligned} \]

\[ \text{sub} : \text{Int} \rightarrow \text{Int}^* \rightarrow \text{Int}^* \]
\[ \text{sub}\ n = \text{lift}\ (\lambda x. x - n) (\lambda x. x + n) \]

Fig. 1. Invertible \text{subs} in \text{SPARCL}.

the constructor for linear functions. Partial invertibility is conveniently expressed by taking additional parameters as in \(\text{Int} \rightarrow \text{Int}^* \rightarrow \text{Int}^*\) and \(\text{Int} \rightarrow (\text{List}\ \text{Int})^* \rightarrow (\text{List}\ \text{Int})^*\). The \text{pin} : \text{A}^* \rightarrow \text{(A} \rightarrow \text{B}^*)^* \rightarrow (\text{A} \otimes \text{B})^*\ operator converts invertible objects into unidirectional ones. It captures a snapshot of its invertible argument and uses the snapshot as a static value in the body to create a safe local scope for the recursive call. Both the invertible argument and evaluation result of the body are returned as the output to preserve invertibility. The \text{with} conditions associated with the branches can be seen as postconditions which will be used for invertible case branching. We leave the detailed discussion of the language constructs to later sections, but would like to highlight the fact that looking beyond the surface syntax, the definition is identical in structure to how \text{subs} is defined in a conventional language: \text{goSubs} has the same recursive pattern with two cases for empty and nonempty lists. This close resemblance to the conventional programming style is what we strive for in the design of \text{SPARCL}.

What \text{SPARCL} brings to the table is bijectivity guaranteed by construction (potentially with partially invertible functions as auxiliary functions). We can run \text{SPARCL} programs in both directions, for example as below, and it is guaranteed that \text{fwd} \ e \ v results in \(v'\) if and only if \(\text{bwd} \ e \ v'\) results in \(v\) (\text{fwd} and \text{bwd} are primitives for forward and backward executions).

\[ > \text{fwd}\ \text{subs}\ [1, 2, 5, 2, 3] \]
\[ [1, 1, 3, -3, 1] \]
\[ > \text{bwd}\ \text{subs}\ [1, 1, 3, -3, 1] \]
\[ [1, 2, 5, 2, 3] \]

This guarantee of bijectivity is clearly different from the case of (functional) logic programming languages such as Prolog and Curry. Those languages rely on (lazy) generate-and-test (Antoy et al., 2000) to find inputs corresponding to a given output, a technique that may be adopted in the context of inverse computation (Abramov et al., 2006). However, the generate-and-test strategy has the undesirable consequence of making reversible programming less apparent: it is unclear to programmers whether they are writing bijective programs that may be executed deterministically. Moreover, lazy generation of inputs may cause unpredictable overhead, whereas in reversible languages (Lutz, 1986; Baker, 1992; Frank, 1997; Mu et al., 2004b; Yokoyama et al., 2008, 2011; James & Sabry, 2012) including \text{SPARCL}, the forward and backward executions of a program take the same steps.
One might notice from the type of \texttt{pin} that \textsc{sparcl} is a higher-order language, in the sense that it contains the simply-typed $\lambda$-calculus (more precisely, the simple multiplicity fragment of $\lambda^q_\rightarrow$ (Bernardy \textit{et al.}, 2018)) as a subsystem. Thus, we can, for example, write an invertible map function in \textsc{sparcl} as below.

\[
\mapR : (a^\rightarrow b^*) \rightarrow \text{(List } a^\text{)}^* \rightarrow \text{(List } b^\text{)}^*
\]

\[
\mapR f \text{ Nil}^* = \text{Nil}^* \quad \text{with null}
\]

\[
\mapR f \text{ (Cons } x \text{ xs)}^* = \text{Cons}^* (f \text{ } x) \text{ (mapR } f \text{ } xs)^* \quad \text{with not } \circ \text{ null}
\]

Ideally, we want to program invertible functions by using higher-order functions. But it is not possible. It is known that there is no higher-order invertible languages where $\rightarrow$ always denotes (not-necessarily-total) bijections. In contrast, there is no issue on having first-order invertible languages as demonstrated by existing reversible languages (see, e.g., RFun (Yokoyama \textit{et al.}, 2011)). Thus, the challenge of designing a higher-order invertible languages lies in finding a sweet spot such that a certain class of functions denote (not-necessarily-total) bijections and programmers can use higher-order functions to abstract computation patterns. Partial invertibility plays an important role here, as functions can be used as static inputs or outputs without violating invertibility. Though this idea has already been considered in the literature (Almendros-Jiménez & Vidal, 2006; Mogensen, 2008; Jacobsen \textit{et al.}, 2018) while with restrictions (specifically, no closures), and the advantage is inherited from Matsuda & Wang (2018c) from which \textsc{sparcl} is inspired, we claim that \textsc{sparcl} is the first invertible programming language that achieved a proper design for higher-order programming.

In summary, our main contributions are as follows:

- We design \textsc{sparcl}, a novel higher-order invertible programming language that captures the notion of partial invertibility. It is the first language that handles both clear separation and close integration of unidirectional and invertible computations, enabling new ways of structuring invertible programs. We formally specify the syntax, type system, and semantics of its core system named $\lambda^{PI}_\rightarrow$ (Section 3).
- We state and prove several properties about $\lambda^{PI}_\rightarrow$ (Section 3.6), including subject reduction, bijectivity, and reversible Turing completeness (Bennett, 1973). We do not state the progress property directly, which is implied by our definitional (Reynolds, 1998) interpreter written in Agda\footnote{Available from \url{https://github.com/kztk-m/sparcl-agda}.} (Section 4).
- We demonstrate the utility of \textsc{sparcl} with nontrivial examples: tree rebuilding from inorder and preorder traversals (Mu & Bird, 2003) and simplified versions of compression algorithms including Huffman coding, arithmetic coding, and LZ77 (Ziv & Lempel, 1977) (Section 5).

In addition, a prototype implementation of \textsc{sparcl} is available from \url{https://github.com/kztk-m/sparcl}, which also contains more examples. All the artifacts are linked from the \textsc{sparcl} web page (\url{https://bx-lang.github.io/EXHIBIT/sparcl.html}).\footnote{The code is archived on Software Heritage: \url{https://archive.softwareheritage.org/swh:1:rev:c3ed8ceb553472de673e5e3804a01ef0bd51a050} and \url{https://archive.softwareheritage.org/swh:1:rev:9750a5aa7626b7bf122bf82c8f57a0af469be91e}.}
A preliminary version of this paper appeared in ICFP20 (Matsuda & Wang, 2020) with the same title. The major changes include a description of our Agda implementation in Section 4 and the arithmetic coding and LZ77 examples in Sections 5.3 and 5.4. Moreover, the related work section (Section 6) is updated to include work published after the preliminary version (Matsuda & Wang, 2020).

2 Overview

In this section, we informally introduce the essential constructs of SPARCL and demonstrate their use with small examples.

2.1 Linear-typed programming

Linearity (or weaker relevance) is commonly adopted in reversible functional languages (Baker, 1992; Mu et al., 2004b; Yokoyama et al., 2011; James & Sabry, 2012; Matsuda & Wang, 2013) to exclude noninjective functions such as constant functions. SPARCL is no exception (we will revisit its importance in Section 2.3) and adopts a linear type system based on \( \lambda q \rightarrow \) (the core system of Linear Haskell (Bernardy et al., 2018)). A feature of the type system is its function type \( A \rightarrow p B \), where the arrow is annotated by the argument’s multiplicity (1 or \( \omega \)). Here, \( A \rightarrow 1 B \) denotes linear functions that use the input exactly once, while \( A \rightarrow \omega B \) denotes unrestricted functions that have no restriction on the use of its input. The following are some examples of linear and unrestricted functions.

\[
\begin{align*}
  id &: a \rightarrow_1 a \\
  id x &= x \\
  double &: \text{Int} \rightarrow_{\omega} \text{Int} \\
  double x &= x + x \\
  const &: a \rightarrow_1 b \rightarrow_{\omega} a \\
  const x y &= x \\
  app &: (p \leq q) \Rightarrow (a \rightarrow p b) \rightarrow, a \rightarrow q b \\
  app f x &= f x
\end{align*}
\]

Observe that the \( double \) used \( x \) twice and \( const \) discards \( y \); hence, the corresponding arrows must be annotated by \( \omega \). The purpose of the type system is to ensure bijectivity. But having linearity alone is not sufficient. We will come back to this point after showing invertible programming in SPARCL. Readers who are familiar with linear-type systems that have the exponential operator \( ! \) (Wadler, 1993) may view \( A \rightarrow \omega B \) as \( !A \rightarrow B \).

A small deviation from the (simply-typed fragment of) \( \lambda q \rightarrow \) is that SPARCL is equipped with rank-1 polymorphism with qualified typing (Jones, 1995) and type inference (Matsuda, 2020). For example, the system infers the following types for the following functions.

\[
\begin{align*}
  id &: a \rightarrow_p a \\
  id x &= x \\
  const &: a \rightarrow_p b \rightarrow a \\
  const x y &= x \\
  app &: (p \leq q) \Rightarrow (a \rightarrow p b) \rightarrow, a \rightarrow q b \\
  app f x &= f x
\end{align*}
\]

In first two examples, \( p \) is arbitrary (i.e., 1 or \( \omega \)); in the last example, the predicate \( p \leq q \) states an ordering of multiplicity, where \( 1 \leq \omega \).\(^6\) This predicate states that if an argument is linear then it cannot be passed to an unrestricted function, as an unrestricted function may use its argument arbitrary many times. A more in-depth discussion of the surface type system is beyond the scope of this paper, but note that unlike the implementation of Linear

\(^6\) For curious readers, we note that the inequality predicate is sufficient for typing our core system (Section 3) where constructors have linear types (Matsuda, 2020).
Haskell as of GHC 9.0.X\(^7\) which checks linearity only when type signatures are given explicitly, \textsc{sparcl} can infer linear types thanks to the use of qualified typing.

For simplicity, we sometimes write \(\rightarrow\) for \(\rightarrow_1\) and simply \(\rightarrow\) for \(\rightarrow_\omega\) when showing programming examples in \textsc{sparcl}.

### 2.2 Multiplication

One of the simplest examples of partially invertible programs is multiplication (Nishida \textit{et al.}, 2005). Suppose that we have a datatype of natural numbers defined as below.

```haskell
data Nat = Z | S Nat
```

In \textsc{sparcl}, constructors have linear types: \(Z : \text{Nat}\) and \(S : \text{Nat} \rightarrow \text{Nat}\).

We define multiplication in term of addition, which is also partially invertible.\(^8\)

\[
\texttt{add} : \text{Nat} \rightarrow \text{Nat}^* \rightarrow \text{Nat}^*
\]

\[
\begin{align*}
\texttt{add } Z & \quad y = y \\
\texttt{add } (S x) & \quad y = S^* (add x y)
\end{align*}
\]

As mentioned in the introduction, we use the type constructor \((-)^*\) to distinguish data that are subject to invertible computation (such as \(\text{Nat}^*\)) and those that are not (such as \(\text{Nat}\)): when the latter is fixed, a partially invertible function is turned into a (not-necessarily-total) bijection, for example, \(\texttt{add } n : \text{Nat}^* \rightarrow \text{Nat}^*\). (For those who read the paper with colors, the arguments of \((-)^*\) are highlighted in dark red.) Values of \((-)^*\)-types are constructed by \textit{lifted constructors} such as \(S^* : \text{Nat}^* \rightarrow \text{Nat}^*\). In the forward direction, \(S^*\) applies \(S\) to the input, and in the backward direction, it strips one \(S\) (and the evaluation gets stuck if \(Z\) is found). In general, since constructors by nature are always bijective (though not necessarily total in the backward direction), every constructor \(C : \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \tau\) automatically give rise to a corresponding lifted version \(C^* : \sigma_1^* \rightarrow \ldots \rightarrow \sigma_n^* \rightarrow \tau^*\).

A partially invertible multiplication function can be defined by using \texttt{add} as below.\(^9\)

\[
\texttt{mul} : \text{Nat} \rightarrow \text{Nat}^* \rightarrow \text{Nat}^*
\]

\[
\begin{align*}
\texttt{mul } z & \quad Z^* = Z^* \quad \text{with } \texttt{isZ} \\
\texttt{mul } z (S x)^* & \quad = add z (\texttt{mul } z x) \quad \text{with } \texttt{not } \circ \texttt{isZ}
\end{align*}
\]

An interesting feature in the \texttt{mul} program is the \textit{invertible pattern matching} (Yokoyama \textit{et al.}, 2008) indicated by patterns \(Z^*\) and \((S x)^*\) (here, we annotate patterns instead of constructors). Invertible pattern matching is a branching mechanism that executes bidirectionally: the forward direction basically performs the standard pattern matching, the backward direction employs an additional \texttt{with} clause to determine the branch to be taken.

For example, \(\texttt{mul } n : \text{Nat}^* \rightarrow \text{Nat}^*\), in the forward direction, values are matched against the forms \(Z\) and \(S x\); in the backward direction, the \texttt{with} conditions are checked upon an

\(^7\) The GHC 9.6.2 user manual: “Linear and multiplicity-polymorphic arrows are always declared, never inferred.” (https://downloads.haskell.org/ghc/9.6.2/docs/users_guide/exts/linear_types.html#linear-types-references).

\(^8\) This type is an instance of the most general type \(\text{Nat} \rightarrow_p \text{Nat}^* \rightarrow_q \text{Nat}^*\) of \texttt{add}; recall that there is no problem in using unrestricted inputs only once. We want to avoid overly polymorphic functions for simplicity of presentation.

\(^9\) Nishida \textit{et al.} (2005) discusses a slightly more complicated but efficient version.
output of the function \textit{mul} \textit{n}: if \textit{isZ} : \textit{Nat} \rightarrow \textit{Bool} returns \textit{True}, the first branch is chosen, otherwise the second branch is chosen. When the second branch is taken, the backward computation of \textit{add} \textit{n} is performed, which essentially subtracts \textit{n}, followed by recursively applying the backward computation of \textit{mul} \textit{n} to the result. As the last step, the final result is enclosed with \textit{S} and returned. In other words, the backward behavior of \textit{mul} \textit{n} recursively tries to subtract \textit{n} and returns the count of successful subtractions.

In Sparcl, \textbf{with} conditions are provided by programmers and expected to be exclusive; the conditions are enforced at run-time: the \textbf{with} conditions are asserted to be postconditions on the branches’ values. Specifically, the branch’s \textbf{with} condition is a positive assertion while all the other branches’ ones are negative assertions. Thus, the forward computation fails when the branch’s \textbf{with} condition is not satisfied, or any of the other \textbf{with} conditions is also satisfied. This exclusiveness enables the backward computation to uniquely identify the branch (Lutz, 1986; Yokoyama et al., 2008). Sometimes we may omit the \textbf{with} condition of the last branch, as it can be inferred as the negation of the conjunction of all the others. For example, in the definition of \textit{goSubs} the second branch’s \textbf{with} condition is not \circ \textit{null}. One could use sophisticated types such as refinement types to infer \textbf{with}-conditions and even statically enforce exclusiveness instead of assertion checking. However, we stick to simple types in this paper as our primal goal is to establish the basic design of Sparcl.

An astute reader may wonder what bijection \textit{mul} \textit{Z} defines, as zero times \textit{n} is zero for any \textit{n}. In fact, it defines a non-total bijection that in the forward direction the domain of the function contains only \textit{Z}, i.e., the trivial bijection between \{\textit{Z}\} and \{\textit{Z}\}.

\textbf{2.3 Why linearity itself is insufficient but still matters}

The primal role of linearity is to prohibit values from being discarded or copied, and Sparcl is no exception. However, linearity itself is insufficient for partially invertible programming.

To start with, it is clear that \textit{−o} is not equivalent to not-necessarily-total bijections. For example, the function \(\lambda x.x \ (\lambda y.y) \ (\lambda z.z) : ((\sigma \rightarrow o \sigma) \circ (\sigma \rightarrow o \sigma) \circ (\sigma \rightarrow o \sigma)) \circ o \sigma \circ o \sigma\) returns \(\lambda y.y\) for both \(\lambda f.\lambda g.\lambda x.f (g x)\) and \(\lambda f.\lambda g.\lambda x.g (f x)\). Theoretically, this comes from the fact that the category of (not-necessarily-total) bijections is not (monoidal) closed. Thus, as discussed above, the challenge is to find a sweet spot where a certain class of functions denote (not-necessarily-total) bijections.

It is known that a linear calculus concerning tensor products (⊗) and linear functions (→) (even with exponentials (!)) can be modeled in the Int-construction (Joyal et al., 1996) of the category of not-necessarily-total bijections (Abramsky et al., 2002; Abramsky, 2005). Here, roughly speaking, first-order functions on base types can be understood as not-necessarily-total bijections. However, it is also known that such a system cannot be easily extended to include sum-types nor invertible pattern matching (Abramsky, 2005, Section 7).

Moreover, linearity does not express partiality as in partially invertible computations. For example, without the \((−)^*\) types, function \textit{add} can have type \textit{Nat} → \textit{Nat} → \textit{Nat} (note the linear use of the first argument), which does not specify which parameter is a fixed one. It even has type \textit{Nat} ⊗ \textit{Nat} → \textit{Nat} after uncurrying though addition is clearly not
fully invertible. These are the reasons why we separate the invertible world and the uni-directional world by using \((-)^\bullet\), inspired by staged languages (Nielson & Nielson, 1992; Moggi, 1998; Davies & Pfenning, 2001). Readers familiar with staged languages may see \(A^\bullet\) as residual code of type \(A\), which will be executed forward or backward at the second stage to output or input \(A\)-typed values.

On the other hand, \((-)^\bullet\) does not replace the need for linearity either. Without linearity, \((\_\_\_)^\bullet\)-typed values may be discarded or duplicated, which may lead to non-bijectivity. Unlike discarding, the exclusion of duplication is debatable as the inverse of duplication can be given as equality check (Glück and Kawabe, 2003). So it is our design choice to exclude duplication (contraction) in addition to discarding (weakening) to avoid unpredictable failures that may be caused by the equality checks. Without contraction, users are still able to implement duplication for datatypes with decidable equality (see Section 5.1.3). However, this design requires duplication (and the potential of failing) to be made explicit, which improves the predictability of the system. Having explicit duplication is not uncommon in this context (Mu et al., 2004b; Yokoyama et al., 2011).

Another design choice we made is to admit types like \((A \rightarrow B)^\bullet\) and \((A^\bullet)^\bullet\) to simplify the formalization; otherwise, kinds will be needed to distinguish types that can be used in \((-)^\bullet\) from general types, and subkinding to allow running and importing bijections (Sections 2.4 and 2.5). Such types are not very useful though, as function- or invertible-typed values cannot be inspected during invertible computations.

### 2.4 Running reversible computation

SPARCL provides primitive functions to execute invertible functions in either directions: \(\text{fwd} : (A^\bullet \rightarrow B^\bullet) \rightarrow A \rightarrow B\) and \(\text{bwd} : (A^\bullet \rightarrow B^\bullet) \rightarrow B \rightarrow A\). For example, we have:

\[
\begin{align*}
> & \text{fwd} (\text{add} (S Z)) (S Z) & \rightarrow (1 +) 1 \\
& S (S Z) & \rightarrow = 2 \\
> & \text{bwd} (\text{add} (S Z)) (S (S Z)) & \rightarrow (1 +)^{-1} 2 \\
& S Z & \rightarrow = 1 \\
> & \text{fwd} (\text{mul} (S (S Z))) (S (S (S Z))) & \rightarrow (2 \times) 3 \\
& S (S (S (S (S Z)))) & \rightarrow = 6 \\
> & \text{bwd} (\text{mul} (S (S Z))) (S (S (S (S Z)))) & \rightarrow (2 \times)^{-1} 6 \\
& S (S (S Z)) & \rightarrow = 3 \\
\end{align*}
\]

Of course, the forward and backward computations may not be total. For example, the following expression (legitimately) fails.

\[
\begin{align*}
> & \text{bwd} (\text{mul} (S (S Z))) (S (S (S Z))) & \rightarrow (2 \times)^{-1} 3 \\
\text{Runtime Error:...}
\end{align*}
\]

The guarantee SPARCL offers is that derived bijections are total with respect to the functions’ actual domains and ranges; i.e., \(\text{fwd} e v\) results in \(u\), then \(\text{bwd} e u\) results in \(v\), and vice versa (Section 3.6.2).

Linearity plays a role here. Linear calculi are considered resource-aware in the sense that linear variables will be lost once used. In our case, resources are \(A^\bullet\)-typed values, as \(A^\bullet\) represents (a part of) an input or (a part of) an output of a bijection being constructed,
which must be retained throughout the computation. This is why the first argument of 
\text{fwd}/\text{bwd} is unrestricted rather than linear. Very roughly speaking, an expression that can 
be passed to an unrestricted function cannot contain linear variables, or “resources”. Thus, 
a function of type \(A^\star \rightarrow B^\star\) passed to \text{fwd}/\text{bwd} cannot use any resources other than \(\text{one}\) 
value of type \(A^\star\) to produce \(\text{one}\) value of type \(B^\star\). In other words, all and only information 
from \(A^\star\) is retained in \(B^\star\), guaranteeing bijectivity. As a result, \text{S PARCL}'s type system 
effectively rejects code like \(\text{bwd} (\lambda x.\text{Z}^\star)\) and \(\text{bwd} (\lambda x.\text{if}\ \text{fwd} (\lambda()^\star x) ()\ \text{then}\ \text{Z}^\star\ \text{else}\ \text{Z}^\star)\) 
as \(x\)'s multiplicity is \(\omega\) in both cases. In the former case, \(x\) is discarded and multiplicity in 
our system is either 1 or \(\omega\). In the latter case, \(x\) appears in the first argument of \text{fwd}, which 
is unrestricted.

### 2.5 Importing existing invertible functions

Bijectivity is not uncommon in computer science or mathematics, and there already exist 
many established algorithms that are bijective. Examples include nontrivial results in 
number theory or category theory, and manipulation of primitive or sophisticated data 
structures such as Burrows-Wheeler transformations on suffix arrays.

Instead of (re)writing them in \text{S PARCL}, the language provides a mechanism to directly 
import existing bijections (as a pair of functions) to construct valid \text{S PARCL} programs: 
\text{lift}: (A \rightarrow B) \rightarrow (B \rightarrow A) \rightarrow A^\star \rightarrow B^\star \text{ converts a pair of functions into a function on } (-)^\star 
typed values, expecting that the pair of functions form mutual inverses. For example, by 
\text{lift}, we can define \text{addInt} as below

\[
\text{addInt} : \text{Int} \rightarrow \text{Int}^\star \rightarrow \text{Int}^\star \\
\text{addInt} n = \text{lift} (\lambda x.x + n)(\lambda x.x - n)
\]

The use of \text{lift} allows one to create primitive bijections to be composed by the vari-
ous constructs in \text{S PARCL}. Another interesting use of \text{lift} is to implement in-language 
inversion.

\[
\text{invert} : (A^\star \rightarrow B^\star) \rightarrow (B^\star \rightarrow A^\star) \\
\text{invert} h = \text{lift}(\text{bwd}\ h)(\text{fwd}\ h)
\]

### 2.6 Composing partially invertible functions

Partially invertible functions in \text{S PARCL} expect arguments of both \((-)^\star\) and non-(\(-)^\star\) 
types, which sometimes makes the calling of such functions interesting. This phenomenon 
is particularly noticeable in recursive calls where values of type \(A^\star\) may need to be fed 
into function calls expecting values of type \(A\). In this case, it becomes necessary to convert 
\(A^\star\)-typed values to \(A\)-typed one. To avoid the risk of violating invertibility, such conver-
sions are carefully managed in \text{S PARCL} through a special function \text{pin}: A^\star \rightarrow (A \rightarrow B^\star) \rightarrow (A \otimes B)^\star, 
inspired by the \text{depGame} function in Kennedy & Vytiniotis (2012) and reversible 
updates (Axelsen et al., 2007) in reversible imperative languages (Lutz, 1986; Frank, 1997; 
Yokoyama et al., 2008; Glück & Yokoyama, 2016). The function \text{pin} creates a static snap-
shot of its first argument (\(A^\star\)) and uses the snapshot (\(A\)) in its second argument. Bijectivity
of a function involving \( \text{pin} \) is guaranteed as the original \( A^* \) value is retained in the output \( (A \otimes B)^* \) together with the evaluation result of the second argument \( (B^*) \). For example, \( \lambda(x, y) \cdot \text{pin} \ x \ (\lambda x'. \text{add} \ x' \ y) \), which defines the mapping between \((n, m)\) and \((n, n + m)\), is bijective. We will define the function \( \text{pin} \) and formally state the correctness property in Section 3.

Let us revisit the example in Section 1. The partially invertible version of \( \text{goSubs} \) can be implemented via \( \text{pin} \) as below.

\[
\text{goSubs} : \text{Int} \rightarrow (\text{List} \ \text{Int})^* \rightarrow (\text{List} \ \text{Int})^*
\]

\[
\text{goSubs} \ _\text{Nil}^* = \text{Nil}^* \text{ with null}
\]

\[
\text{goSubs} \ n \ (\text{Cons} \ x \ xs)^* = (\text{case} \ \text{pin} \ x \ (\lambda x'. \text{goSubs} \ x' xs) \ of
\]

\[
(x, r)^* \rightarrow \text{Cons}^* \ (\text{sub} \ n \ x \ r \ \text{with} \ \lambda_{\_ \text{True}}) \text{ with not o null}
\]

Here, we used \( \text{pin} \) to convert \( x : \text{Int}^* \) to \( x' : \text{Int} \) in order to pass it to the recursive call of \( \text{goSubs} \). In the backward direction, \( \text{goSubs} \ n \) executes as follows.\(^{10}\)

\[
\text{bwd} \ (\text{goSubs} \ 0)^{[1, 1, 3, -3, 1]}
\]

\[
= \{ \text{Cons branch is taken; Cons (sub 0 x) r = [1, 1, 3, -3, 1] \implies x = 1, r = [1, 3, -3, 1].} \}
\]

\[
\text{Cons 1 (bwd (goSubs 1)^{[1, 3, -3, 1]})}
\]

\[
= \{ \text{Cons branch is taken; Cons (sub 1 x) r = [1, 3, -3, 1] \implies x = 2, r = [3, -3, 1].} \}
\]

\[
\text{Cons 1 (Cons 2 (bwd (goSubs 2)^{[3, -3, 1]})}
\]

\[
= \{ \text{Cons branch is taken; Cons (sub 2 x) r = [3, -3, 1] \implies x = 5, r = [-3, 1].} \}
\]

\[
\text{Cons 1 (Cons 2 (Cons 5 (bwd (goSubs 5)^{[-3, 1]})])}
\]

\[= \ldots \]

\[
= \text{Cons 1 (Cons 2 (Cons 5 (Cons 2 (Cons 3 (bwd (goSubs 3)^{[]})))))}
\]

\[
= \{ \text{Nil branch is taken.} \}
\]

\[
\text{Cons 1 (Cons 2 (Cons 5 (Cons 2 (Cons 3 Nil)))))} = [1, 2, 5, 2, 3]
\]

Note that the first arguments of (recursive) calls of \( \text{goSubs} \) (which are static) have the same values \((1, 2, 5, 2, 3)\) in both forward/backward executions, distinguishing their uses from those of the invertible arguments. As one can see, \( \text{goSubs} \ n \) behaves exactly like the hand-written \( \text{goSubs}' \) in \( \text{sub}^{-1} \) which is reproduced below.

\[
\text{goSubs'} \ _\text{Nil} = [ ]
\]

\[
\text{goSubs'} \ n \ (y : ys) = \text{let} \ x = y + n \ \text{in} \ x : \text{goSubs'} \ x \ ys
\]

The use of \( \text{pin} \) commonly results in an invertible \text{case} with a single branch, as we see in \( \text{goSubs} \) above. We capture this pattern with an invertible \text{let} as a shorthand notation, which enables us to write \text{let} \ \( p^* = e_1 \ \text{in} \ e_2 \) for \text{case} \( e_1 \) of \( [p^* \rightarrow e_2 \ \text{with} \ \lambda_{\_ \text{True}}] \). The definition of \( \text{goSubs} \) shown in Section 1 uses this shorthand notation, which is reproduced in Figure 2(a).

We would like to emphasize that partial invertibility, as supported in \( \text{SPARCL} \), is key to concise function definitions. In Figure 2, we show side-by-side two versions of the same program written in the same language: the one on the left allows partial invertibility whereas the one on the right requires all functions (include the intermediate ones) to be fully invertible (note the different types in the two versions of \( \text{goSubs} \) and \( \text{sub} \)). As a result,

\(^{10}\) This execution trace is (overly) simplified for illustration purpose. See Section 3.5 for the actual operational semantics.
goSubsF is much harder to define and the code becomes fragile and error-prone. This advantage of SPARCL, which is already evident in this small example, becomes decisive when dealing with larger programs, especially those requiring complex manipulation of static values (for example, the Huffman coding in Section 5.2).

We end this section with a theoretical remark. One might wonder why \((-\cdot)^*\) is not a monad. This intuitively comes from the fact that the first and second stages are in different languages (the standard one and an invertible one, respectively) with different semantics. More formally, \((-\cdot)^*\), which brings a type in the second stage into the first stage, forms a functor, but the functor is not endo. Recall that \(A^*\) represents residual code in an invertible system of type \(A\); that is, \(A^*\) and its component \(A\) belong to different categories (though we have not formally described them).\(^{11}\) One then might wonder whether \((-\cdot)^*\) is a relative monad (Altenkirch et al., 2010). To form a relative monad, one needs to find a functor that has the same domain and codomain as (the functor corresponding to) \((-\cdot)^*\). It is unclear whether there exists such a functor other than \((-\cdot)^*\) itself; in this case, the relative monad operations do not provide any additional expressive power.

### 2.7 Implementations

We have implemented a proof-of-concept interpreter for SPARCL including the linear type system, which is available from https://github.com/kztk-m/sparcl. The implementation adds two small but useful extensions to what is presented in this paper. First, the implementation allows nonlinear constructors, such as \(\text{MkUn} : a \to \text{Un} a\) which serves as \(!\) and helps us to write a function that returns both linear and unrestricted results. Misusing such constructors in invertible pattern matching is guarded against by the type system (otherwise it may lead to discarding or copying of invertible values). Second, the implementation uses the first-match principle for both forward and backward computations. That is, both patterns and \(\text{with}\) conditions are examined from top to bottom. Recall also that

\(^{11}\) For curious readers, we note our conjecture that \((-\cdot)^*\) corresponds to the Yoneda embedding for the CPO-enriched category of (strict) bijections, analogous to Moggi (1998), although denotational semantics is outside the scope of this paper.
the implementation uses a non-indentation-sensitive syntax for simplicity as mentioned in Section 1.

It is worth noting that the implementation uses Matsuda (2020)’s type inference to infer linear types effectively without requiring any annotations. Hence, the type annotations in this paper are more for documentation purposes.

As part of our effort to prove type safety (subject reduction and progress), we also produced a parallel implementation in Agda to serve as proofs (Section 3.6), available from https://github.com/kztk-m/sparcl-agda.

3 Core system: $\lambda_{PI}$

This section introduces $\lambda_{PI}$, the core system that SPARCL is built on. Our design mixes ideas of linear-typed programming and meta-programming. As mentioned in Section 2.1, the language is based on (the simple multiplicity fragment of) $\lambda_q$ (Bernardy et al., 2018), and, as mentioned in Section 2.3, it is also two-staged (Nielson & Nielson, 1992; Moggi, 1998) with different meta and object languages. Specifically, the meta stage is a usual call-by-value language (i.e., unidirectional), and the object stage is an invertible language. By having the two stages, partial invertibility is made explicit in this formalization.

In what follows, we use a number of notational conventions. A vector notation $t$ denotes a sequence such as $t_1, \ldots, t_n$ or $t_1; \ldots; t_n$, where each $t_i$ can be of any syntactic category and the delimiter (such as “,” and “;”) can differ depending on the context; we also refer to the length of the sequence by $|t|$. In addition, we may refer to an element in the sequence $t$ as $t_i$. A simultaneous substitution of $x_1, \ldots, x_n$ in $t$ with $s_1, \ldots, s_n$ is denoted as $t[s_1/x_1, \ldots, s_n/x_n]$, which may also be written as $t[s/x]$.

3.1 Central concept: Bijections at the heart

The surface language of SPARCL is designed for programming partially invertible functions, which are turned into bijections (by fixing the static arguments) for execution. This fact is highlighted in the core system $\lambda_{PI}$ where we have a primitive bijection type $A \leftrightarrow B$, which is inhabited by bijections constructed from functions of type $A^* \rightarrow B^*$. Technically, having a dedicated bijection type facilitates reasoning. For example, we may now straightforwardly state that “values of a bijection type $A \equiv B$ are bijections between $A$ and $B$” (Corollary 3.4).

Accordingly, the $\text{fwd}$ and $\text{bwd}$ functions for execution in SPARCL are divided into application operators $\triangleright$ and $\triangleleft$ that apply bijection-typed values and an $\text{unlift}$ operator for constructing bijections from functions of type $A^* \rightarrow B^*$. For example, we have $\text{unlift} (\text{add} (S Z)) : \text{Nat} \rightleftharpoons \text{Nat}$ (where $\text{add} : \text{Nat} \rightarrow \text{Nat}^* \rightarrow \text{Nat}^*$ is defined in Section 2), and the bijection can be executed as $\text{unlift} (\text{add} (S Z)) \triangleright S \ Z$ resulting in $S \ Z$. In fact, the operators $\text{fwd}$ and $\text{bwd}$ are now derived in $\lambda_{PI}$, as $\text{fwd} = \lambda_{\omega}.h.\lambda_{\omega}.x.\text{unlift} h \triangleright x$ and $\text{bwd} = \lambda_{\omega}.h.\lambda_{\omega}.x.\text{unlift} h \triangleleft x$.

Here, $\omega$ of $\lambda_{\omega}$ indicates that the bound variable can be used arbitrary many. In contrast, $\lambda_1$ indicates that the bound variable must be used linearly. Hence, for example, $\lambda_1 x.Z$ and $\lambda_1 x.(x, x)$ are ill-typed, while $\lambda_1 x.x$, $\lambda_0 x.Z$ and $\lambda_0 x.(x, x)$ are well-typed. Similarly, we also annotate (unidirectional) cases with the multiplicity of the variables bound by pattern
matching. Thus, for example, \texttt{case}_1 \ S \ Z \ of \ \{ \ S \ x \rightarrow (x,x) \} \ and \ \lambda_1.x.\texttt{case}_\omega \ x \ of \ \{ \ S \ y \rightarrow Z \} \ are \ ill-typed.

3.2 Syntax

The syntax of $\lambda PI \rightarrow$ is given as below.

Expressions: $e ::= x \mid \lambda_\pi x.e \mid e_1 \ e_2 \mid C \ e \mid \texttt{case}_\pi \ e_0 \ of \ \{ \ p \rightarrow e \} \mid C^* \ e \mid \texttt{case} \ e_0 \ of \ \{ \ p^* \rightarrow e \ \text{with} \ e' \} \mid \texttt{pin} \ e_1 \ e_2 \mid \texttt{unlift} \ e \mid e_1 \triangleright e_2 \mid e_1 \triangleleft e_2$

Patterns: $p ::= \mathcal{C} \pi$

Multiplicities: $\pi ::= 1 \mid \omega$

There are three lines for the various constructs of expressions. The ones in the first line are standard except the annotations in $\lambda$ and \texttt{case} that determine the multiplicity of the variables introduced by the binders: $\pi = 1$ means that the bound variable is linear, and $\pi = \omega$ means there is no restriction. These annotations are omitted in the surface language as they are inferred. The second and third lines consist of constructs that deal with invertibility. As mentioned above, \texttt{unlift} $e$, $e_1 \triangleright e_2$, and $e_1 \triangleleft e_2$ handles bijections which can be used to encode \texttt{fwd} and \texttt{bwd} in SPARCL. We have already seen lifted constructors, invertible \texttt{case}, and \texttt{pin} in Section 2. For simplicity, we assume that \texttt{pin}, $C$ and $C^*$ are fully applied. Lifted constructor expressions $C^* \ e$ and invertible \texttt{cases} are basic invertible primitives in $\lambda PI \rightarrow$. They are enough to make our system reversible Turing complete (Bennett, 1973) (Theorem 3.5); i.e., all bijections can be implemented in the language. For simplicity, we assume that patterns are nonoverlapping both for unidirectional and invertible \texttt{cases}. We do not include \texttt{lift}, which imports external code into SPARCL, as it is by definition unsafe. Instead, we will discuss it separately in Section 3.7.

Different from conventional reversible/invertible programming languages, the constructs \texttt{unlift} (together with $\triangleright$ and $\triangleleft$) and \texttt{pin} support communication between the unidirectional world and the invertible world. The \texttt{unlift} construct together with $\triangleright$ and $\triangleleft$ runs invertible computation in the unidirectional world. The \texttt{pin} operator is the key to partiality; it enables us to temporarily convert a value in the invertible world into a value in the unidirectional world.

3.3 Types

Types in $\lambda PI \rightarrow$ are defined as below.

$A, B ::= \alpha \mid T \ A \mid A \rightarrow_\pi B \mid A^* \mid A \rightleftharpoons B$

Here, $\alpha$ denotes a type variable, $T$ denotes a type constructor, $A \rightarrow_\pi B$ is a function type annotated with the argument’s multiplicity $\pi$, $(\rightarrow)^*$ marks invertibility, and $A \rightleftharpoons B$ is a bijection type.

Each type constructor $T$ comes with a set of constructors $C$ of type

$C : A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow T \ \alpha$
with \( \text{fv}(A_i) \in \{\sigma\} \) for any \( i \).\(^{12}\) Type variables \( \alpha \) are only used for types of constructors in the language. For example, the standard multiplicative product \( \otimes \) and additive sum \( \oplus \) (Wadler, 1993) are represented by the following constructors.

\[
(-, -) : \alpha_1 \rightarrow \alpha_2 \rightarrow \alpha_1 \otimes \alpha_2 \quad \text{lnL} : \alpha_1 \rightarrow (\alpha_1 \oplus \alpha_2) \quad \text{lnR} : \alpha_2 \rightarrow (\alpha_1 \oplus \alpha_2)
\]

We assume that the set of type constructors at least include \( \otimes \) and \( \text{Bool} \), where \( \text{Bool} \) has the constructors \( \text{True} : \text{Bool} \) and \( \text{False} : \text{Bool} \). Types can be recursive via constructors; for example, we can have a list type \( \text{List} \alpha \) with the following constructors.

\[
\text{Nil} : \text{List} \alpha \quad \text{Cons} : \alpha \rightarrow \text{List} \alpha \rightarrow \text{List} \alpha
\]

We may write \( \overline{A} \rightarrow B \) for \( A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow B \) (when \( n \) is zero, \( \overline{A} \rightarrow B \) is \( B \)). We shall also instantiate constructors implicitly and write \( C : \overline{A'} \rightarrow T \overline{B} \) when there is a constructor \( C : \overline{A} \rightarrow T \overline{a} \) and \( A'_i = A_i[\overline{B}/\alpha] \) for each \( i \). Thus, we assume all types in our discussions are closed.

Negative recursive types are allowed in our system, which, for example, enables us to define general recursions without primitive fixpoint operators. Specifically, via \( F \) with the constructor \( \text{MkF} : (F \alpha \rightarrow \alpha) \rightarrow F \alpha \), we have a fixpoint operator as below.

\[
\text{fix}_\pi \triangleq \lambda_\alpha f. \lambda_\pi a. (\lambda_\alpha x. \lambda_\pi a f \ (\text{out} \ x \ x) \ a) \ (\text{MkF} \ (\lambda_\alpha x. \lambda_\pi a f \ (\text{out} \ x \ x) \ a)) \ a
\]

where \( \text{out} \) has type \( F C \rightarrow \alpha F C \rightarrow C \) for any \( C \) (in this case \( C = A \rightarrow B \)).

The most special type in the language is \( A^* \), which is the invertible version of \( A \). More specifically, the invertible type \( A^* \) represents residual code in an invertible system that are executed forward and backward at the second stage to output and input \( A \)-typed values.

Values of type \( A^* \) must be treated linearly and can only be manipulated by invertible operations, such as lifted constructors, invertible pattern matching, and \( \text{pin} \). To keep our type system simple, or more specifically single-kindred, we allow types like \( (A \rightarrow B)^* \) and \( (A^*)^* \), while the category of (not-necessarily-total) bijections are not closed and \( \lambda_{\text{PI}} \) has no third stage. These types do not pose any problem, as such components cannot be inspected in invertible computation by any means (except in \( \text{with} \) conditions, which are unidirectional, i.e., run at the first stage).

Note that we consider the primitive bijection types \( A \rightleftharpoons B \) as separate from \( (A \rightarrow B) \otimes (B \rightarrow A) \). This separation is purely for reasoning; in our theoretical development, we will show that \( A \rightleftharpoons B \) denotes pairs of functions that are guaranteed to form (not-necessarily-total) bijections (Corollary 3.4).

### 3.4 Typing relation

A \emph{typing environment} is a mapping from variables \( x \) to pairs of type \( A \) and its multiplicity \( \pi \), meaning that \( x \) has type \( A \) and can be used \( \pi \)-many times. We write \( x_1 : \pi_1 \ A_1, \ldots, x_n : \pi_n \ B_n \) instead of \( \{x_1 \mapsto (A_1, \pi_1), \ldots, x_n \mapsto (B_n, \pi_n)\} \) for readability and write \( \epsilon \) for the empty

---

\(^{12}\) For simplicity, we assume a constructor can only have linear fields. Extending our discussions to constructors with unrestricted field is straightforward for the unidirectional part of the language. Such constructors cannot appear as lifted constructors and patterns in invertible cases.
environment. Reflecting the two stages, we adopt a dual context system (Davies & Pfennig, 2001), which has unidirectional and invertible environments, denoted by $\Gamma$ and $\Theta$ respectively. This separation of the two is purely theoretical, for the purpose of facilitating reasoning when we interpret $A^*$-typed expressions that are closed in unidirectional variables but may have free variables in $\Theta$ as bijections. In fact, our prototype implementation does not distinguish the two environments. For all invertible environments $\Theta$, without the loss of generality we assume that the associated multiplicities must be 1, i.e., $\Theta(x) = (A_x, 1)$ for any $x \in \text{dom}(\Theta)$. Thus, we shall sometimes omit multiplicities for $\Theta$. This assumption is actually an invariant in our system since any variables introduced in $\Theta$ must have multiplicity 1. We make this explicit in order to simplify the theoretical discussions. Moreover, we assume that the domains of $\Gamma$ and $\Theta$ are disjoint.

Given two unidirectional typing environments $\Gamma_1$ and $\Gamma_2$, we define the addition $\Gamma_1 + \Gamma_2$ as below.

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} (A, \omega) & \text{if } \Gamma_1(x) = (A, \omega) \text{ and } \Gamma_2(x) = (A, \omega) \\ (A, \pi) & \text{if } \Gamma_j(x) = (A, \pi) \text{ and } x \notin \text{dom}(\Gamma_j) \text{ for some } i \neq j \in \{1, 2\} \end{cases}$$

If $\text{dom}(\Gamma_1)$ and $\text{dom}(\Gamma_2)$ are disjoint, we sometimes write $\Gamma_1, \Gamma_2$ instead of $\Gamma_1 + \Gamma_2$ to emphasize the disjointness. A similar addition applies to invertible environments. But as only multiplicity 1 is allowed in $\Theta$, $\Theta_1 + \Theta_2 = \Theta$ implicitly implies $\text{dom}(\Theta_1) \cap \text{dom}(\Theta_2) = \emptyset$.

We define multiplication of multiplicities as below.

$$1\pi = \pi 1 = \pi \quad \omega\pi = \pi \omega = \omega$$

Given $\Gamma = x_1 : \pi_1 A_1, \ldots, x_n : \pi_n A_n$, we write $\pi \Gamma$ for the environment $x_1 : \pi_1 A_1, \ldots, x_n : \pi_n A_n$. A similar notation applies to invertible environments. Again, $\omega\Theta' = \Theta$ means that $\Theta' = \varepsilon$. Notice that it can hold that $\Gamma = \Gamma + \Gamma$ and $\Gamma = \omega\Gamma = 1\Gamma$ if $\Gamma(x) = (\varepsilon, \omega)$ for all $x \in \text{dom}(\Gamma)$.

The typing relation $\Gamma; \Theta \vdash e : A$ reads that under environments $\Gamma$ and $\Theta$, expression $e$ has type $A$ (Figure 3). The definition basically follows $\lambda^\Theta_\varepsilon$ (Bernardy et al., 2018) except having two environments for the two stages. Although multiplicities in $\Theta$ are always 1, some of the typing rules refers to $\omega\Theta$ (which implies $\Theta = \varepsilon$) in the conclusion parts, to emphasize that $\Gamma$ and $\Theta$ are treated similarly by the rules. The idea underlying this type system is that, together with the operational semantics in Section 3.5, a term-in-context $e; \Theta \vdash e : A^*$ defines a piece of code representing a bijection between $\Theta$ and $A$, and hence $e; \varepsilon \vdash e' : A \equiv B$ defines a bijection between $A$ and $B$ (see Section 3.6). Our Agda implementation explained in Section 4, which we mentioned in Sections 1 and 2.7, follows this idea with some generalization. The typing rules in Figure 3 are divided into three groups: the standard ones inherited from $\lambda^\Theta_\varepsilon$ (T-VAR, T-ABS, T-APP, T-CON, and T-CASE), the ones for the invertible part (T-RVAR, T-RCON, and T-RCASE), and the ones for the interaction between the two (T-PIN, T-UNLIFT, T-FAPP, and T-BAPP).

Intuitively, the multiplicity of a variable represents the usage of a resource to be associated with the variable. Hence, multiplicities in $\Gamma$ and $\Theta$ are synthesized rather than checked in typing. This viewpoint is useful for understanding T-APP and T-CASE; it is natural that, if an expression $e$ is used $\pi$ times, the multiplicities of variables in $e$ are multiplied by $\pi$. Discarding variables, or weakening, is performed in the rules T-VAR, T-RVAR, T-CON,
Typing Rules for Expressions \(\Gamma; \Theta \vdash e : A\) and Patterns \(p : A \mapsto \pi\)

\[
\frac{\omega\Gamma + x : \pi A; \omega\Theta \vdash e : B}{\Gamma; \Theta \vdash \lambda_x e : A \rightarrow_\pi B} \quad \text{T-VAR}
\]

\[
\frac{\Gamma; \Theta \vdash \lambda_x e : A \rightarrow_\pi B}{\Gamma, x : \pi A; \Theta \vdash e : B} \quad \text{T-ABS}
\]

\[
\frac{\Gamma_1; \Theta_1 \vdash e_1 : A \rightarrow_\pi B \quad \Gamma_2; \Theta_2 \vdash e_2 : A}{\Gamma_1 + \pi\Gamma_2; \Theta_1 + \pi\Theta_2 \vdash e_1 e_2 : B} \quad \text{T-APP}
\]

\[
\frac{n = |e| = |\overline{A}| \quad C : \overline{A} \rightarrow_\pi \overline{B} \quad (\Gamma_1; \Theta_1 \vdash e_i : A_i)_i}{\omega\Gamma_0 + \Gamma_1 + \ldots + \Gamma_n; \Theta_1 + \ldots + \Theta_n \vdash C \overline{e} : \overline{B}} \quad \text{T-CON}
\]

\[
\frac{\Gamma_0; \Theta_0 \vdash e_0 : A \quad \{p_i : A \mapsto \pi \Gamma_i \quad \Gamma_i; \Theta_i \vdash e_i : B\}_i}{\pi\Gamma_0 + \Gamma_1 + \pi\Theta_0 + \Theta \vdash \text{case}_\pi e_0 \ of \ (p \rightarrow \overline{e}) : B} \quad \text{T-CASE}
\]

\[
\frac{\omega\Gamma; x : A \vdash x : A'}\quad n = |e| = |\overline{A}| \quad C : \overline{A} \rightarrow_\pi \overline{B} \quad (\Gamma_i; \Theta_i \vdash e_i : A_i^*)_i}{\omega\Gamma_0 + \Gamma_1 + \ldots + \Gamma_n; \Theta_1 + \ldots + \Theta_n \vdash C \overline{e} : (\overline{B}^{\pi_1})} \quad \text{T-RCON}
\]

\[
\frac{\Gamma_0; \Theta_0 \vdash e_0 : A \quad \{p_i : A \mapsto \pi \Theta_i \quad \Gamma_i; \Theta_i \vdash e_i : B^*\} \quad \Gamma'; \Theta' \vdash e' : B \mapsto_{\omega} \text{Bool}}_{\pi_1 + \Gamma_0 + \pi\Theta_0 + \Theta \vdash \text{case}_\pi e_0 \ of \ (p^* \rightarrow e with \ \overline{e'}) : B^*} \quad \text{T-RCASE}
\]

\[
\frac{\Gamma_1; \Theta_1 \vdash e_1 : A \quad \Gamma_2; \Theta_2 \vdash e_2 : A \rightarrow_{\omega} B^*}{\Gamma_1 + \Gamma_2; \Theta_1 + \Theta_2 \vdash \text{pin} e_1 e_2 : (A \otimes B)^*} \quad \text{T-PIN}
\]

\[
\frac{\Gamma_1; \Theta_1 \vdash e_1 : A \mapsto_{\omega} B \quad \Gamma_2; \Theta_2 \vdash e_2 : A\!\!\!\!\!\!\rightarrow_{\omega} B}{\omega\Gamma_1 + \omega\Gamma_2; \omega\Theta_1 + \Theta_2 \vdash \text{unlift} e : A \mapsto_{\omega} B} \quad \text{T-UNLIFT}
\]

\[
\frac{\Gamma_1; \Theta_1 \vdash e_1 : A \mapsto_{\omega} B \quad \Gamma_2; \Theta_2 \vdash e_2 : A\!\!\!\!\!\!\rightarrow_{\omega} B}{\Gamma_1 + \omega\Gamma_2; \Theta_1 + \omega\Theta_2 \vdash e_1 \triangleright e_2 : B} \quad \text{T-FAPP}
\]

\[
\frac{\Gamma_1; \Theta_1 \vdash e_1 : A \mapsto_{\omega} B \quad \Gamma_2; \Theta_2 \vdash e_2 : B\!\!\!\!\!\!\rightarrow_{\omega} B}{\Gamma_1 + \omega\Gamma_2; \Theta_1 + \omega\Theta_2 \vdash e_1 \triangleleft e_2 : A} \quad \text{T-BAPP}
\]

\[
\frac{C : \overline{A} \rightarrow_\pi \overline{B}}{\overline{C} : \overline{B} \mapsto_{\pi} x : \pi \overline{A}} \quad \text{T-UNLESS}
\]

Fig. 3. Typing rules for expressions and patterns.

and T-RCON which can be leaves in a derivation tree. Note that weakening is not allowed for \(\Theta\)-variables as they are linear.

The typing rules for the invertible part would need additional explanation. In T-RVAR, \(x\) has type \(A^*\) if the invertible typing environment is the singleton mapping \(x : A\). One explanation for this is that \(\Theta\) represents the typing environment for the object (i.e., invertible) system. Another explanation is that we simply omit \((-)^{\ast}\) as all variables in \(\Theta\) must have types of the form \(A^{\ast}\). Rule T-RCON says that we can lift a constructor to the invertible world leveraging the injective nature of the constructor. Rule T-RCASE says that the invertible case is for pattern-matching on \((-)^{\ast}\)-typed data; the pattern matching is done in the invertible world, and thus the bodies of the branches must also have \((-)^{\ast}\)-types. Recall that the with-conditions \((e'_i)\) are used for deciding which branch is used in backward computation. The type \(B \mapsto_{\omega} \text{Bool}\) indicates that they are conventional unrestricted functions, and \(\omega\Gamma'\) and \(\omega\Theta'\) in the conclusion part of the rule indicates that their uses are unconstrained. Notice that, since the linearity comes only from the use of \((-)^{\ast}\)-typed values, there is little motivation to use linear variables to define conventional functions in \(\lambda_{\pi_1}\). The operators \text{pin}, \text{unlift}, \mapsto, and \triangleleft are special in \(\lambda_{\pi_1}\). The operator \text{pin} is simply a fully applied version of the one in Section 2; so we do not repeat the explanation. Rules T-UNLIFT, T-FAPP, and T-BAPP are inherited from the types of fwd and bwd in Section 2. Recall that \(\omega\Theta\) ensures \(\Theta = \varepsilon\), and thus the arguments of \text{unlift} and constructed bijections must be closed in terms
of invertible variables. It might look a little weird that \( \varphi_1 \triangleright e_2/\varphi_1 \lhd e_2 \) uses \( \varphi_1 \) linearly; this is not a problem because \( \Theta_1 \) in T-FA/PP/T-BAPP must be empty for expressions that occur in evaluation (Lemma 3.2).

### 3.5 Operational semantics

The semantics of \( \lambda_{\pi}^{\pi} \) consists of three evaluation relations: unidirectional, forward, and backward. The unidirectional evaluation evaluates away the unidirectional constructs such as \( \lambda \)-abstractions and applications, and the forward and backward evaluation specifies bijections.

For example, let us consider an expression \( e = (\lambda_\omega f. (f \ y)) (\lambda_1 x. S^* x) \). Due to \( \lambda \)-abstractions and function applications, it is not immediately clear how we can interpret the expression as a bijection. The unidirectional evaluation \( \Downarrow \) is used to evaluate these unidirectional constructs away to make way for the forward and backward evaluation to interpret the residual term. For the above expression, we have \( e \Downarrow S^* (S^* y) \) where the residual \( S^* (S^* y) \) is ready to be interpreted bijectively. The forward evaluation \( \mu \vdash E \Rightarrow v \) evaluates a residual \( E \) under an environment \( \mu \) to obtain a value \( v \) as usual. For example, we have \( \{y \mapsto Z\} \vdash S^* (S^* y) \Rightarrow S (S Z) \). The backward evaluation \( E \Leftarrow v \vdash \mu \) does the opposite; it inversely evaluates \( E \) to find an environment \( \mu \) for a given value \( v \), so that the corresponding forward evaluation of \( E \) returns the value for the environment. For example, we have \( S^* (S^* y) \Leftarrow S (S Z) \vdash \{y \mapsto Z\} \).

This is the basic story, but computation can be more complicated in general. With \texttt{case} and \texttt{pin}, the forward \( \Rightarrow \) and backward \( \Leftarrow \) evaluation depend on the unidirectional evaluation \( \Downarrow \); and with \( \triangleright \) and \( \lhd \), the unidirectional evaluation \( \Downarrow \) also depends on the forward \( \Rightarrow \) and backward \( \Leftarrow \) ones. Technically, the linear type system is also the key to the latter type of dependency, which is an important difference from related work in bidirectional programming (Matsuda & Wang, 2018c).

#### 3.5.1 Values and residuals

We first define a set of \textit{values} \( v \) and a set of \textit{residuals} \( E \) as below.

**Values:** \( v ::= \lambda_\pi x.e \mid C \triangleright E \mid \langle x. E \rangle \)

**Residuals:** \( E ::= x \mid C^* \triangleright E \mid \texttt{case} \ E_0 \ \text{of} \ \{p^* \mapsto e \ \text{with} \ \lambda_\omega x. e'\} \mid \texttt{pin} \ E_1 (\lambda_\omega x_2. e_2) \)

The residuals are \((-)^*\)-typed expressions, which are subject to the forward and backward evaluations. The syntax of residuals makes it clear that branch bodies in invertible \texttt{cases} are not evaluated in the unidirectional evaluation; otherwise, recursive definitions involving them usually diverge. A variable is also a value. Indeed, our evaluation targets expressions/residuals that may be open in term of invertible variables. The value \( \langle x. E \rangle \) represents a bijection. Intuitively, \( \langle x. E \rangle \) is a single-holed residual \( E \) where the hole is represented by the variable \( x \). The type system ensures that the \( x \) is the only free variable in \( E \) so that \( E \) is ready to be interpreted as a bijection. Since \( \langle x. E \rangle \) is not an expression defined so far, we extend expressions to include this form as \( e ::= \cdots \mid \langle x. E \rangle \) together with the following typing rule:

\[
\frac{
\Gamma; \Theta; x : A \vdash E : B^* \\
\omega \varepsilon \Theta; \omega \Theta \vdash \langle x. E \rangle : A \equiv B \quad \text{T-HOLED}
}{\omega \varepsilon \Gamma; \omega \Theta \vdash E : B^*}
\]
Unidirectional Evaluation \( e \downarrow v \)

- \( \lambda x.e \downarrow \lambda x.e \)
- \( e_1 \downarrow \lambda x.e \)
- \( e_2 \downarrow v \)
- \( e[v/x] \downarrow v \)
- \( e \downarrow v \)
- \( e_0 \downarrow v \)
- \( p_1 \mu = v_0 \)
- \( e_1 \downarrow v \)

Forward Evaluation \( \mu \vdash E \Rightarrow v \)

- \( [x \mapsto v] \vdash x \Rightarrow v \)
- \( \mu \vdash E \Rightarrow v \)
- \( \mu \vdash E_1 \Rightarrow v_1 \)
- \( e_2[v/x] \downarrow E_2 \)
- \( E_2 \downarrow v_2 \)
- \( \mu \vdash \mu_1 \vdash \mu_2 \vdash \mu \)
- \( \mu_0 = E_0 \vdash p_1 \mu_1 \)
- \( \mu_0 \vdash \mu \vdash \mathit{case} E_0 \text{ of } \{ p^* \rightarrow e \} \)

Backward Evaluation \( E \leftarrow v \vdash \mu \)

- \( x \leftarrow v \vdash [x \mapsto v] \)
- \( \mu \leftarrow E \leftarrow v \vdash \mu \)
- \( E_1 \leftarrow v_1 \vdash \mu_1 \)
- \( e_2[v/x] \downarrow E_2 \)
- \( E_2 \leftarrow v_2 \vdash \mu_2 \)
- \( \mu_1 \leftarrow \mu_2 \leftarrow \mu \)
- \( \mu_0 = E_0 \leftarrow p_1 \mu_1 \)
- \( \mu_0 \leftarrow \mu \vdash \mathit{case} E_0 \text{ of } \{ p^* \rightarrow e \} \)

Fig. 4. Evaluation relations: unidirectional, forward and backward.

It is crucially important that \( x \) is added to the invertible environment. Recall again that \( \omega \Theta \) ensures \( \Theta = \varepsilon \). Also, since values are closed in terms of unidirectional variables, a value of the form \( (x.E) \) cannot have any free variables.

3.5.2 Three evaluation relations: Unidirectional, forward, and backward

The evaluation relations are shown in Figure 4, which are defined by mutually dependent evaluation rules.

The unidirectional evaluation is rather standard, except that it treats invertible primitives (such as lifted constructors, invertible cases, lift, and pin) as constructors. A subtlety is that we assume dynamic \( \alpha \)-renaming of invertible cases to avoid variable capturing. The evaluation rules can evaluate open expressions by having \( x \vdash x \); recall that residuals can contain variables. The unlift operator uses a fresh variable \( y \) in the evaluation to make a single-holed residual \( (y.E) \) as a representation of bijection. Such single-holed residuals can be used in the forward direction by \( e_1 \triangleright e_2 \) and in the backward direction by \( e_1 \triangleleft e_2 \), by triggering the corresponding evaluation.

The forward evaluation \( \mu \vdash E \Rightarrow v \) states that under environment \( \mu \), a residual \( E \) evaluates to a value \( v \), and the backward evaluation \( E \leftarrow v \vdash \mu \) inversely evaluates \( E \) to return
the environment \( \mu \) from a value \( v \): the forward and backward evaluation relations form a bijection. For variables and invertible constructors, both forward and backward evaluation rules are rather straightforward. The rules for invertible case expression are designed to ensure that every branch taken in one direction may and must be taken in the other direction too. This is why we check the with conditions even in the forward evaluation: the condition is considered as a post-condition that must exclusively hold after the evaluation of a branch. The pin operator changes the behavior of the backward computation of the second argument based on the result of the first argument; notice that \( v_1 \), the parameter for the second argument, is obtained as the evaluation result of the first argument in the forward evaluation, and as the first component of the result pair in the backward evaluation. Notice that the unidirectional evaluation \( \Downarrow \) involved in the presented evaluation rules is performed in the same way in both evaluation, which is the key to bijectivity of \( E \).

### 3.6 Metatheory

In this subsection, we present the key properties about \( \lambda_{PI} \).

#### 3.6.1 Subject reduction

First, we show a substitution lemma for \( \lambda_{PI} \). We only need to consider substitution for unidirectional variables because substitution for invertible variables never happens in evaluation; recall that we use environments \( (\mu) \) in the forward and backward evaluation.

**Lemma 3.1.** \( \Gamma, x : \pi A; \Theta \vdash e : B \) and \( \Theta' \vdash e' : A \) implies \( \Gamma + \pi \Gamma'; \Theta + \pi \Theta' \vdash e[e'/x] : B \).

Note that the substitution is only valid when \( \Theta + \pi \Theta' \) satisfy the assumption that invertible variables have multiplicity 1. This assumption is guaranteed by typing of the constructs that trigger substitution.

Then, by Lemma 3.1, we have the subject reduction properties as follows:

**Lemma 3.2** (subject reduction). The following properties hold:

- Suppose \( \varepsilon; \Theta \vdash e : A \) and \( e \Downarrow v \). Then, \( \varepsilon; \Theta \vdash v : A \) holds.
- Suppose \( \varepsilon; \Theta \vdash E : A^* \) and \( \mu \vdash E \Rightarrow v \). Then, \( \text{dom}(\Theta) = \text{dom}(\mu) \) holds, and \( \varepsilon; \varepsilon \vdash \mu(x) : \Theta(x) \) for all \( x \in \text{dom}(\Theta) \) implies \( \varepsilon; \varepsilon \vdash v : A \).
- Suppose \( \varepsilon; \Theta \vdash E : A^* \) and \( E \Leftarrow v \vdash \mu \). Then, \( \text{dom}(\Theta) = \text{dom}(\mu) \) holds, and \( \varepsilon; \varepsilon \vdash v : A \) implies \( \varepsilon; \varepsilon \vdash \mu(x) : \Theta(x) \) for all \( x \in \text{dom}(\Theta) \).

**Proof** By (mutual) induction on the derivation steps of evaluation.
actually an invariant; even though open expressions and values are involved in the unidirectional evaluation, the forward and backward evaluations always take and return closed values.

### 3.6.2 Bijectivity

Roughly speaking, correctness means that every value of type $A \equiv B$ forms a bijection. Values of type $A \equiv B$ has the form $\langle x. E \rangle$. By Lemma 3.2 and T-HOLED, values that occur in the evaluation of a well-typed term can be typed as $\varepsilon; \varepsilon \vdash \langle x. E \rangle : A \equiv B$, which implies $\varepsilon; x : A \vdash E : B^\star$. Since values $\langle x. E \rangle$ can only be used by $\triangleright$ and $\triangleleft$, bijectivity is represented as: $\{x \mapsto v\} \vdash E \Rightarrow v'$ if and only if $E \Leftarrow v' \vdash \{x \mapsto v\}$.

To do so, we prove the following more general correspondence between the forward and backward evaluation relations, which is rather straightforward as the rules of the two evaluations are designed to be symmetric.

**Lemma 3.3** (bijectivity of residuals). $\mu \vdash E \Rightarrow v$ if and only if $E \Leftarrow v \vdash \mu$.

**Proof** Each direction is proved by induction on a derivation of the corresponding evaluation. Note that every unidirectional evaluation judgment $e' \downarrow v'$ occurring in a derivation of one direction also appears in the corresponding derivation of the other direction, and hence we can treat the unidirectional evaluation as a block box in this proof.

Then, by Lemma 3.2, we have the following corollary stating that $\langle x. E \rangle : A \equiv B$ actually implements a bijection between $A$-typed values and $B$-typed values.

**Corollary 3.4** (bijectivity of bijection values). Suppose $\varepsilon; \varepsilon \vdash \langle x. E \rangle : A \equiv B$. Then, for any $v$ and $u$ such that $\varepsilon; \varepsilon \vdash v : A$ and $\varepsilon; \varepsilon \vdash v' : B$, we have $\{x \mapsto v\} \vdash E \Rightarrow v'$ if and only if $E \Leftarrow v' \vdash \{x \mapsto v\}$.

### 3.6.3 Note on the progress property

Progress is another important property that, together with subjection reduction, proves the absence of certain errors during evaluation. However, a standard progress property is usually based on small-step semantics, and yet $\lambda_{\Pi}$ has a big-step operational semantics, which was chosen for its advantage in clarifying the input-output relationship of the forward and backward evaluation, as demonstrated by Lemma 3.3. A standard small-step semantics, which defines one-step evaluation as a relation between terms, is not suitable in this regard. Abstract machines are also unsatisfactory, as they will obscure the correspondence between the forward and backward evaluations.

We instead establish progress by directly showing that the evaluations do not get stuck other than with branching-related errors. This is done as an Agda implementation (mentioned in Sections 1 and 2.7) of definitional (Reynolds, 1998) interpreters, which use the (sized) delay monad (Capretta, 2005; Abel & Chapman, 2014) and manipulate intrinsically typed (i.e., Church style) expressions, values, and residuals. The interpreter uses sums, products, and iso-recursive types instead of constructors. Also, instead of substitution,

---

13 Here, we consider syntactic (definitional) equality of values, but it is rather easy to extend the discussion to observational equivalence.
value environments are used in the unidirectional evaluation to avoid the shifting of de Bruijn terms. See Section 4 for details of the implementation. We note that, as a bonus track, the Agda implementation comes with a formal proof of Lemma 3.3.

3.6.4 Reversible Turing completeness

Reversible Turing completeness (Bennett, 1973) is an important property that general-purpose reversible languages are expected to have. Similar to the standard Turing completeness, being reversible Turing complete for a language means that all bijections can be expressed in the language (Bennett, 1973).

It is unsurprising that \( \lambda_{\Pi} \) is reversible Turing complete, as it has recursion (via \( \text{fix}_\pi \) in Section 3.3) and reversible branching (i.e., invertible \text{case}).

**Theorem 3.5.** \( \lambda_{\Pi} \) is reversible Turing complete.

The proof is done by constructing a simulator for a given reversible Turing machine, which is presented in Appendix A. We follow the construction in Yokoyama et al. (2011) except the last step, in which we use a general reversible looping operator as below.\(^{14}\)

\[
\text{trace} : ((a \oplus x)^* \rightarrow (b \oplus x)^*) \rightarrow a^* \rightarrow b^*
\]

As its type suggests, \text{trace} \( h \) applies \( h \) to \( \text{InL} \ a \) repeatedly until it returns \( \text{InL} \ b \); the function loops while \( h \) returns a value of the form \( \text{InR} \ x \). Intuitively, this behavior corresponds to the reversible loop (Lutz, 1986). In functional programming, loops are naturally encoded as tail recursions, which, however, are known to be difficult to handle in the contexts of program inversion (Glück & Kawabe, 2004; Mogensen, 2006; Matsuda et al., 2010; Nishida & Vidal, 2011). In fact, our implementation uses a non-trivial reversible programming technique, namely Yokoyama et al. (2012)'s optimized version of Bennett (1973)'s encoding. The higher-orderness of \( \lambda_{\Pi} \) (and \text{SPARCL}) helps here, as the effort is made once and for all.

3.7 Extension with the \text{lift} operator

One feature we have not yet discussed is the \text{lift} operator that creates primitive bijections from unidirectional programs, for example, \text{sub} as we have seen in Section 2.

Adding \text{lift} to \( \lambda_{\Pi} \) is rather easy. We extend expressions to include \text{lift} as \( e ::= \cdots | \text{lift} \ e_1 \ e_2 \ e_3 \) together with the following typing rule.

\[
\frac{\Gamma_1 ; \Theta_1 \vdash e_1 : A \to \omega B \quad \Gamma_2 ; \Theta_2 \vdash e_2 : B \to \omega A \quad \Gamma_3 ; \Theta_3 \vdash e_3 : A^*}{\omega \Gamma_1 + \omega \Gamma_2 + \Gamma_3 ; \omega \Theta_1 + \omega \Theta_2 + \Theta_3 \vdash \text{lift} \ e_1 \ e_2 \ e_3 : B^*} \quad \text{T-LIFT}
\]

Accordingly, we extend evaluation by adding residuals of the form \( \text{lift} \ (\lambda_{\omega} x_1.e_1) (\lambda_{\omega} x_2.e_2) E_3 \) together with the following forward and backward evaluation rules (we omit the obvious unidirectional evaluation rule for obtaining residuals of this form).

\(^{14}\) The operator is named after the trace operator (Joyal et al., 1996) in the category of bijections (Abramsky et al., 2002).
we have the unit type (), product types \( A \otimes B \) as one can see, there are no user-defined types.

The substitution lemma (Lemma 3.1) and the subject reduction properties (Lemma 3.2) are also lifted to \( \text{lift} \).

However, \( \text{lift} \) is by nature unsafe, which requires an additional condition to ensure correctness. Specifically, the bijectivity of \( A \Rightarrow B \) typed values is only guaranteed if \( \text{lift} \) is used for pairs of functions that actually form bijections. For example, the uses of \( \text{lift} \) to construct \( \text{sub} \) in Section 2 are indeed safe. In Section 5.2.1, we will see another interesting example showing the use of conditionally safe \( \text{lifts} \) (see \text{unsafeNew} in Section 5.2.1).

## 4 Mechanized proof in Agda

In this section, we provide an overview of our implementation of SPARCL in Adga which serves as a witness of the subjection reduction and the progress properties. Also, the implementation establishes the invariant that the multiplicities of the variables in \( \Theta \) are always 1. This is crucial for the correctness but non-trivial to establish in our setting, because an expression and the value obtained as the evaluation result of the expression may have different free invertible variables due to the unidirectional free variables in the expression.

The Agda implementation also comes with the proof of Lemma 3.3.

### 4.1 Differences in formalization

We first spell out the differences in our Agda formalization from the system \( \lambda^{\pi}_{\alpha} \) described in Section 3. As mentioned earlier, the implementation uses products, sums, and iso-recursive types instead of constructors and uses environments instead of substitutions to avoid tedious shifting of de Bruijn terms.

In addition, the Agda version comes with a slight extension to support \( ! \) in linear calculi.

We begin with the difference in types. The Agda version targets the following set of types.

\[
A, B ::= A \rightarrow_{\pi} B | () | A \otimes B | A \oplus B | !_{\pi} A | \alpha | \mu \alpha . A | A^* | A \Rightarrow B
\]

As one can see, there are no user-defined types \( \mathcal{T} \mathcal{A} \) that come with constructors; instead, we have the unit type \( () \), product types \( A \otimes B \), sum types \( A \oplus B \), and (iso-) recursive types \( \mu \alpha . A \). As for the extension mentioned earlier, there are also types of the form of \( !_{\pi} A \) which intuitively denote \( A \)-typed values together with the witness of \( \pi \)-many copyability of the values.

The expressions are updated to match the types.

\[
e ::= x | \lambda_{\pi} x . e | e_1 . e_2 |
\]

- \( () \) | \( \text{let}_{\pi} () \) = \( e_1 \) \( \text{in} \) \( e_2 \) | \( (e_1, e_2) \) | \( \text{let}_{\pi} (x_1, x_2) = e_1 \) \( \text{in} \) \( e_2 \)
- \( \text{ln} \) \( e \) \( \text{in} \) \( e \) | \( \text{case}_{\pi} e_0 \) of \( \{ \text{ln} \) \( x_1 \rightarrow e_1 ; \text{ln} \) \( x_2 \rightarrow e_2 \} \)
- \( '_{\pi} e \) | \( \text{let}_{\pi} !x = e_1 \) \( \text{in} \) \( e_2 \) | \( \text{roll} \) \( e \) | \( \text{unroll} \) \( e \)
- \( x^* \) | \( * \) | \( \text{let} (x^* = e_1 \) \( \text{in} \) \( e_2 \) | \( (e_1, e_2)^* \) | \( \text{let} (x_1, x_2)^* = e_1 \) \( \text{in} \) \( e_2 \)
- \( \text{ln} \) \( x^* \) \( e \) | \( \text{ln} \) \( e^* \) \( e \) | \( \text{case} e_0 \) of \( \{ \text{ln} \) \( x^* \rightarrow e_1 ; \text{ln} \) \( x^* \rightarrow e_2 \} \) \( \text{with} \) \( e' \)
- \( \text{roll} \) \( x^* \) \( e \) | \( \text{unroll} \) \( x^* \) \( e \)
- \( \text{pin} e_1 e_2 \) | \( \text{unlift} \) \( e \) | \( e_1 \) \( <> \) \( e_2 \) | \( e_1 \) \( <> \) \( e_2 \)
Instead of constructors and pattern matching, this version includes the introduction and elimination forms for each form of types except $A^\ast$. And for types $\langle A \otimes B, A \oplus B, \mu A, A \rangle$, there are corresponding invertible versions. For example, we have the introduction form $(e_1, e_2)$ and the elimination form $\text{let}_\pi (x_1, x_2) = e_1 \in e_2$ for the product types, and their invertible counterparts $(e_1, e_2)^\ast$, and $\text{let} (x_1, x_2)^\ast = e_1 \in e_2$. Here, $\pi$ ensures that $e_1$ is used $\pi$-many times and so as the variables $x_1$ and $x_2$, similarly to the $\pi$ of $\text{case}_\pi$ in the original calculus $\lambda^\Pi_1$ (see T-CASE in Figure 3). Note that both (unidirectional and invertible) sorts of cases are only for sum types and have exactly two branches. For simplicity, the invertible case has one with condition instead of two, as one is enough to select one of the two branches. Since we use intrinsically typed terms, the syntax of terms must be designed so that the typing relation becomes syntax-directed. Hence, we have two sorts of variable expressions (not variables themselves) $x$ and $x^\ast$, which will be typed by $T\text{-VAR}$ and $T\text{-RVAR}$, respectively.

The typing rules for the expressions can be obtained straightforwardly from Figure 3, except for the newly introduced ones that manipulate $!_\pi A$-typed values.

$$
\begin{align*}
\frac{\Gamma; \Theta \vdash e : A \quad \pi \Gamma; \pi \Theta \vdash !_\pi e : !_\pi A}{\pi \Gamma; \pi \Theta \vdash !e : !_\pi A} \\
\frac{\Gamma_1; \Theta_1 \vdash e_1 : !_\pi A \quad \Gamma_2, x : !_\pi \pi_1 A; \Theta_2 \vdash e_2 : B}{\pi \Gamma_1 + \Gamma_2; \pi \Theta_1 + \Theta_2 \vdash \text{let}_\pi !x = e_1 \in e_2 : B}
\end{align*}
$$

An intuition underlying the rules is that $!_\pi A$ is treated as a GADT (Many $\pi A$) with the constructor $\text{MkMany} : A \otimes_\pi \text{Many} \pi A$ capturing the multiplicity $\pi$. As the constructor discharges the multiplicity $\pi$ when pattern matched, the latter rule says that the copyability $\pi_1$ is discharged by the binding, regardless of the use of the examined expression $e_1$. For example, we have $x_1 : \langle (), \pi : A \rangle; e \vdash \text{let}_\pi !z = (\text{let}_\pi () = x \in !_\pi y) \in (z, x) : A \otimes A$ where $x$ is used once in the expression but $z$ can be used twice as the binding discharged the copyability witnessed by $!_\pi y$.

The sets of values and residuals are also updated accordingly. Here, the main change is the use of environments $\theta := \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \}$.

$$
\begin{align*}
v & := \langle \lambda_{\pi} x.e, \theta \rangle | () | (v_1, v_2) | \ln L v | \ln R v | !_\pi v | \text{roll} v | \langle x.E \rangle | E \\
E & := x^\ast | ()^\ast | \text{let}(x)^\ast = E_1 \in E_2 | (E_1, E_2)^\ast | \text{let} (x_1, x_2)^\ast = E_1 \in E_2 \\
& | \ln L^\ast E | \ln R^\ast E | \text{case} E_0 \text{ of } \{(\ln L x)^\ast \mapsto e_1, \theta_1 \}; \{(\ln R x)^\ast \mapsto e_2, \theta_2 \} \text{ with } u' \\
& | \text{roll}^\ast E | \text{unroll}^\ast E \\
& | \text{pin} E v
\end{align*}
$$

We intentionally used different metavariables $\theta$ and $\mu$ for environments: the former is used in the unidirectional evaluation and may contain invertible free variables, while the latter is used in the forward and backward evaluations. The typing relation $\Theta \vdash \theta : \Gamma$ must be aware of such free invertible variables as below.

$$
\{ x_1, \ldots, x_n \} = \text{dom}(\theta) = \text{dom}(\Gamma) \quad \{ \pi_1 \Theta_1 \vdash \theta(x_i) : A_i \} \quad \text{where } \Gamma(x_i) = (A_i, \pi_i)_i \\
\pi_1 \Theta_1 + \cdots + \pi_n \Theta_n \vdash \theta : \Gamma
$$

The typing rules for values and residuals ($\Theta \vdash v : A$ and $\Theta \vdash E : A^\ast$) are obtained straightforwardly from the rules for expressions (Figure 3) with $\Gamma = \varepsilon$, except for the two new forms of value and residual involving closures. One of the two is a function closure expression $\langle \lambda_{\pi} x.e, \theta \rangle$, which comes with the following typing rule.

$$
\begin{align*}
\Theta_{\text{env}} \vdash \theta : \Gamma \\
\text{let}_\pi x : A, \Gamma; \Theta_{\text{body}} \vdash e : B \\
\Theta_{\text{env}} + \Theta_{\text{body}} \vdash (\lambda_{\pi} x.e, \theta) : A \rightarrow_\pi B
\end{align*}
$$
where the bind operation is frozen (i.e., represented as a constructor). This deviation from
all-no-omega
The predicate
The other is the invertible case residual, which has the following type rule.

\[ \Theta_0 \vdash E : (A_1 \oplus A_2)^* \quad \{ \Theta_{\text{env}} \vdash \theta_i : \Gamma_i ; x : A_i, \Theta_{\text{body}} \vdash e_i : C^* \}_i \quad \varepsilon \vdash v' : C \rightarrow \omega \text{ Boolean} \]

\[ \Theta_0 + \Theta_{\text{env}} + \Theta_{\text{body}} \vdash \text{case } E_0 \text{ of } \{ (\text{InL x})^* \rightarrow e_1, \theta_1) ; (\text{InR x})^* \rightarrow e_2, \theta_2 \} \text{ with } v' : C^* \]

Here, we write Boolean for () \(\oplus\) () for readability.

We omit the concrete representations of expressions, values, and residuals as they are straightforward. A subtlety is that the Agda version adopts separate treatment of types and multiplicities: that is, \(\Gamma = \{ x_1 : \pi_1, \ldots, x_n : \pi_n, A_n \} \) is separated into \(\Gamma_i = \{ x_1 : A_1, \ldots, x_n : A_n \} \) and \(\Gamma_m = \{ x_1 : \pi_1, \ldots, x_n : \pi_n \} \), so that complex manipulation of multiplicities happens only for the latter. Also, \(\Theta\) environments are separated into \(\Theta_i\) and \(\Theta_m\) in a similar way.

### 4.2 Evaluation functions

The Agda implementations include two definitional (Reynolds, 1998) interpreters for intrinsically typed terms: one is for the unidirectional evaluation \(\downarrow\) and the other is for the forward and backward evaluations \(\Rightarrow\) and \(\Leftarrow\). More specifically, the former one takes \(\Theta' \vdash \theta : \Gamma\) and \(\Theta \vdash e : A\) to produce a value \(\Theta' + \Theta \vdash v : A\) if terminates, and the latter takes a residual \(\Theta \vdash E : A^*\) to yield a not-necessarily-total bijection between \(\mu : \Theta\) and \(\vdash v : A\), where \(\mu : \Theta\) means \(\vdash \mu(x) : \Theta(x)\) for any \(x\).

In our Agda development, an environment-in-context \(\Theta' \vdash \theta : \Gamma\), a term-in-context \(\Gamma; \Theta \vdash e : A\), and a value-in-context \(\Theta \vdash v : A\) are represented by types \(\text{ValEnv} \ \Gamma_t \ \Gamma_m \ \Theta_t \ \Theta_m\), \(\text{Term} \ \Gamma_t \ \Gamma_m \ \Theta_t \ \Theta_m \ A\), and \(\text{Value} \ \Theta_t \ (\Theta_m +_m \Theta_m) \ A\), respectively. Recall that we have adopted the separate treatment of types and multiplicities. Hence, instead of having a single \(\Theta\), we have \(\Theta_t\) and \(\Theta_m\) where the former typing environment is treated in the usual way. Also, regarding the latter evaluation, a residual-in-context \(\Theta \vdash E : A^*\), a typed-environment (for the forward/backward evaluation) \(\mu : \Theta\), and a value-in-context \(\vdash v : A\) are represented by types \(\text{Residual} \ \Theta_t \ \Theta_m \ (A^*), \ \text{RValEnv} \ \Theta_t \ \Theta_m, \ \text{Value} \ [\ ] \ \emptyset \ A\), respectively. The different representations ([ ]) and \(\emptyset\) are used for the empty typing environment and the empty multiplicity environment: the former type is just a list of types, while the latter is a type indexed by the former.

Now, we are ready to give the signatures of the two evaluation functions.

\[ \text{eval : } \forall \{ \Theta_t \ \Theta_m \ \Gamma_t \ \Gamma_m \ \Theta_t \ \Theta_m \ A \} \ (i : \text{Size}) \rightarrow \text{all-no-omega} \ (\Theta_m +_m \Theta_m) \]
\[ \rightarrow \text{ValEnv} \ \Gamma_t \ \Gamma_m \ \Theta_t \ \Theta_m \rightarrow \text{Term} \ \Gamma_t \ \Gamma_m \ \Theta_t \ \Theta_m \ A \]
\[ \rightarrow \text{DELAY} \ (\text{Value} \ \Theta_t \ (\Theta_m +_m \Theta_m) \ A) \ i \]
\[ \text{evalR : } \forall \{ \Theta_t \ \Theta_m \ A \} \ (i : \text{Size}) \rightarrow \text{all-no-omega} \ \Theta_m \rightarrow \text{Residual} \ \Theta_t \ \Theta_m \ (A^*), \ \text{RValEnv} \ \Theta_t \ \Theta_m \ \Leftrightarrow \ \text{Value} \ [\ ] \ \emptyset \ A \]

The predicate all-no-omega asserts that a given multiplicity environment does not contain the multiplicity \(\omega\), respecting the assumption on the core system that the multiplicities involved in \(\Theta\) are always 1. This property is considered as an invariant, because we need to have a witness of the property to call eval and evalR recursively. The type constructor DELAY is a variant the (sized) delay monad (Capretta, 2005; Abel & Chapman, 2014), where the bind operation is frozen (i.e., represented as a constructor). This deviation from the original is useful for the proof of Lemma 3.3 (Section 4.3). The record type \(i \vdash_f a \Leftrightarrow b\)
K. Matsuda and M. Wang

represents not-necessarily-total bijections and has two fields: \( \text{Forward} : a \to \text{DELAY} \ b \ i \) and \( \text{Backward} : b \to \text{DELAY} \ a \ i \).

The fact that we have implemented these two functions in Agda witnesses the subject reduction and the progress property. For the two functions to be type correct, they must use appropriate recursive calls for intrinsically typed subterms, which is indeed what the subject reduction requires. Also, Agda is a total language, meaning that we need to give the definition for every possible structures—in other words, every typed term is subject to evaluation. Note that, by \( \text{DELAY} \), the evaluations are allowed to go into infinite loops, which is legitimate for the progress property. We also use infinite loops to represent errors, which are thrown only in the following situations.

- forward evaluation of invertible \texttt{cases} with imprecise \texttt{with} conditions, and
- backward evaluation of \texttt{lnL} and \texttt{lnR} \texttt{E} that receive opposite values.

The fact that the interpreters are typechecked in Agda serves as a constructive proof that there are no other kind of errors.

\textbf{Caveat: sized types.} As their signatures suggest, the definitions of \texttt{eval} and \texttt{evalR} rely on (a variant of) the sized delay monad. However, the sized types are in fact an unsafe feature in Agda 2.6.2, which may lead to contradictions in cases,\textsuperscript{15} and, as far as we are aware, the safe treatment of sized types is still open in Agda. Nevertheless, we believe that our use of sized types, mainly regarding sized delay monads, is safe as the use is rather standard (namely, we use the finite sized types in the definitions of \texttt{eval} and \texttt{evalR} to ensure productivity, and then use the infinite size when we discuss the property of the computation).

\section*{4.3 Bijectivity of the forward and backward evaluation}

The statement of Lemma 3.3 is formalized in Agda as the signatures of the following functions:

\begin{align*}
\text{forward-backward} : & \\
\forall \{ \Theta, m \ A \} \to (\text{ano} : \text{all-no-omega} \ \Theta, m) \to (E : \text{Residual} \ \Theta, m \ (A \bullet)) \\
& \to \forall \mu \nu \\
& \to \text{Forward} (\text{evalR} \ \infty \ \text{ano} \ E) \ \mu \to \nu \to \text{Backward} (\text{evalR} \ \infty \ \text{ano} \ E) \ \nu \to \mu \\
\text{backward-forward} : & \\
\forall \{ \Theta, m \ A \} \to (\text{ano} : \text{all-no-omega} \ \Theta, m) \to (E : \text{Residual} \ \Theta, m \ (A \bullet)) \\
& \to \forall \mu \nu \\
& \to \text{Backward} (\text{evalR} \ \infty \ \text{ano} \ E) \ \nu \to \mu \to \text{Forward} (\text{evalR} \ \infty \ \text{ano} \ E) \ \mu \to \nu
\end{align*}

Here, \( m \to \nu \), which reads that \( m \) evaluates to \( \nu \), is an inductively defined predicate asserting that \( m : \text{DELAY} \ a \ \infty \) terminates and produces the final outcome \( \nu \). This relation has a similar role to \( \Sigma \ (m \downarrow) (\lambda m \to \text{extract} \ w \equiv \nu) \), where \( \downarrow \) and \( \text{extract} \) are defined in the module \texttt{Codata.Sized.Delay} in the Agda standard library, but the key difference is its explicit bind structures. Thanks to the explicit bind structures, we can perform the proof.

\textsuperscript{15} See, e.g., \texttt{https://github.com/agda/agda/issues/1201} and \texttt{https://github.com/agda/agda/issues/6002}.
straightforwardly by induction on $E$ and case analysis on $Forward (evalR \bowtie ano E) \mu \rightarrow v$ or $Backward (evalR \bowtie ano E) v \rightarrow \mu$, leveraging the fact that the forward/backward evaluation “mirrors” the backward/forward evaluation also in the bind structures.

5 Larger examples

In this section, we demonstrate the utility of SPARCL with four examples, in which partial invertibility supported by SPARCL is the key for programming. The first one is rebuilding trees from preorder and inorder traversals (Mu & Bird, 2003), and the latter three are simplified versions of compression algorithms (Salomon, 2008), namely, the Huffman coding, arithmetic coding, and LZ77 (Ziv & Lempel, 1977).16

5.1 Rebuilding trees from a preorder and an inorder traversals

It is well known that we can rebuild a node-labeled binary tree from its preorder and inorder traversals, provided that all labels in the tree are distinct. That is, for binary trees of type

```
data Tree = L | N Int Tree Tree
```

the following Haskell function $pi$ is bijective.

```haskell
pi :: Tree \rightarrow ([Int], [Int])
pit = (preorder t, inorder t)
preorder L = []
preorder (N a l r) = a : preorder l ++ preorder r
inorder L = []
inorder (N a l r) = inorder l ++ [a] ++ inorder r
```

For example, for binary trees

$t_1 = N \ 1 \ (N \ 2 \ (N \ 3 \ L \ L) \ L) \ L,$

$t_2 = N \ 1 \ (N \ 2 \ (N \ 3 \ L \ L)) \ L,$

$t_3 = N \ 1 \ (N \ 2 \ L \ L) \ (N \ 3 \ L \ L),$

$t_4 = N \ 1 \ L \ (N \ 2 \ (N \ 3 \ L \ L) \ L),$

$t_5 = N \ 1 \ L \ (N \ 2 \ L \ (N \ 3 \ L \ L))$

that share the preorder traversal $[1, 2, 3]$, the inorder traversals distinguish them:

```
inorder \ t_1 = [3, 2, 1],
inorder \ t_2 = [2, 3, 1],
inorder \ t_3 = [2, 1, 3],
inorder \ t_4 = [1, 3, 2],
inorder \ t_5 = [1, 2, 3].
```

The uniqueness of labels is key to the bijectivity of $pi$. It is clear that $pi^{-1}$ returns $L$ for $([], [])$, so the nontrivial part is how $pi^{-1}$ will do for a pair of nonempty lists. Let us write $(a : p, i)$ for the pair. Then, since $i$ contains exactly one $a$, we can unambiguously split $i$ as $i = i_1 ++ [a] ++ i_2$. Then, by $pi^{-1}(\text{take (length } i_1) \ p, i_1)$, we can recover the left child $l,$

16 They are included in the Examples directory in the prototype implementation repository (https://github.com/kztk-m/sparcl/) as Pi.sparcl, Huff.sparcl, ArithmeticCoding.sparcl, and LZ77.sparcl respectively.
and, by $p^{-1}(\text{drop} \ (\text{length} \ i_1) \ p, i_2)$, we can recover the right child $r$. After that, from $a$, $l$, and $r$, we can construct the original input as $N \ a \ l \ r$. Notice that this inverse computation already involves partial invertibility such as the splitting of the inorder traversal list based on $a$, which is invertible for fixed $a$ with the uniqueness assumption.

It is straightforward to implement the above procedure in SPARCL. However, such a program is inefficient due to the cost of splitting. Program calculation is an established technique for deriving efficient programs through equational reasoning (Gibbons, 2002), and in this case of tree-rebuilding, it is known that a linear-time inverse exists and can be derived (Mu & Bird, 2003).

In the following, we demonstrate that program calculation works well in the setting of SPARCL. Interestingly, thinking in terms of partial-invertibility not only produces a Sparcl program, but actually improves the calculation by removing some of the more-obscure steps. Our calculation presented below basically follows Mu & Bird (2003, Section 3), although the presentation is a bit different as we focus on partial invertibility, especially the separation of unidirectional and invertible computation.

Note that Glück & Yokoyama (2019) give a reversible version of tree rebuilding using (an extension of) $\text{R-}\text{WHILE}$ (Glück & Yokoyama, 2016), a reversible imperative language inspired by Janus (Lutz, 1986; Yokoyama et al., 2008). However, $\text{R-}\text{WHILE}$ only supports a very limited form of partial invertibility (Section 6.1), and the difference between their definition and ours is similar to what is demonstrated by the $\text{goSubs}$ and $\text{goSubsF}$ examples in Figure 2.

### 5.1.1 Calculation of the original definition

The first step is tupling (Chin, 1993; Hu et al., 1997) which eliminates multiple data traversals. The elimination of multiple data traversals is known to be useful for program inversion (Eppstein, 1985; Matsuda et al., 2012).

\[
\begin{align*}
pi &:: \text{Tree} \rightarrow ([\text{Int}], [\text{Int}]) \\
pi \ L & = ([], []) \\
pi (N \ a \ l \ r) & = \text{let} (pr, ir) = pi \ r; (pl, il) = pi \ l \ \text{in} \ (a : pl ++ pr, il ++ [a] ++ ir)
\end{align*}
\]

Mu & Bird (2003, Section 3) also use tupling as the first step in their derivation.

The next step is to eliminate $++$, a source of inefficiency. The standard technique is to use accumulation parameters (Kühnemann et al., 2001). Specifically, we obtain $piA$ satisfying $piA \ t \ py \ iy = \text{let} (p, i) = pi \ t \ \text{in} \ (p ++ py, i ++ iy)$ as below.

\[
\begin{align*}
piA &:: \text{Tree} \rightarrow [\text{Int}] \rightarrow [\text{Int}] \rightarrow ([\text{Int}], [\text{Int}]) \\
piA \ L & = (py, iy) \\
piA (N \ a \ l \ r) & = \text{let} (pr, ir) = piA \ r \ py \ iy; (pl, il) = piA \ l \ pr \ (a : ir) \ \text{in} \ (a : pl, il)
\end{align*}
\]

The invertibility of $piA$ is still not clear because $piA$ is called with two different forms of the accumulation parameter $iy$: one is the case where $iy$ is empty (e.g., the initial call $pi \ x = piA \ x \ [ ] \ [ ]$), and the other is the case where it is not (e.g., the recursion for the left child $piA \ l \ pr \ (a : ir)$). This distinction between the two is important because, unlike the former, an inverse for the latter is responsible for searching for the appropriate place to separate the inorder-traversal list. Nevertheless, this separation can be achieved by deriving
a specialized version $\pi$ of $\pi A$ satisfying $\pi x = \pi A x [\ ] [\ ]$ (we reuse the name as it implements the same function).

$$
\begin{align*}
\pi &:: \text{Tree} \rightarrow ([\text{Int}], [\text{Int}]) \\
\pi L & = ([\ ] , [\ ] ) \\
\pi (N \ a \ l \ r) & = \text{let} \ (pr, ir) = \pi R \ r \ \text{in} \ (pl, il) = \pi A \ l \ pr \ (a : ir) \ \text{in} \ (a : pl, il)
\end{align*}
$$

Fig. 5. Invertible pre- and in-order traversal in SPARCL.

Having this new version of $\pi$, we now have an invariant that $iy$ of $\pi A t py iy$ is always nonempty; the other case is separated into a call to $\pi$. Moreover, we can determine the head $h$ of $iy$ beforehand in both forward and backward computations; this is exactly the label we search for to split the inorder-traversal list. Indeed, if we know the head $h$ of $iy$ beforehand, we can distinguish the ranges of the two branches of $\pi A$: for the first branch $(py, iy)$, as $iy$ is returned as is, the head of the second component is the same as $h$, and for the second branch $(a : pl, il)$, the head of the second component of the return value cannot be equal to $h$, i.e., the head of $iy$. Recall that $\pi A t py iy = \text{let} \ (p, i) = \pi t \ \text{in} \ (p ++ py, i ++ iy)$; thus, $ir$ in the definition of $\pi A$ must have the form of $\cdots ++ iy$, and then $il$ must have the form of $\cdots ++ [a] ++ \cdots ++ iy$.

Thus, as the last step of our calculation, we clarify the unidirectional part, namely the head of the second component of the accumulation parameters of $\pi A$, by changing it to a separate parameter. Specifically, we prepare the function $\pi A S$ satisfying $\pi A S h t py iy = \pi A t py (h : iy)$ as below.

$$
\begin{align*}
\pi A S :: & \text{Int} \rightarrow \text{Tree} \rightarrow [\text{Int}] \rightarrow [\text{Int}] \rightarrow ([\text{Int}], [\text{Int}]) \\
\pi A S h L & = py iy = (py, h : iy) \\
\pi A S h (N \ a \ l \ r) & = \text{let} \ (pr, ir) = \pi A S h r py iy; \ (pl, il) = \pi A S a l pr ir \ \text{in} \ (a : pl, il)
\end{align*}
$$

Also, we replace the function call of $\pi A$ in $\pi$ appropriately.

$$
\begin{align*}
\pi &:: \text{Tree} \rightarrow ([\text{Int}], [\text{Int}]) \\
\pi L & = ([\ ] , [\ ] ) \\
\pi (N \ a \ l \ r) & = \text{let} \ (pr, ir) = \pi r \ \text{in} \ (pl, il) = \pi A S a l pr ir \ \text{in} \ (a : pl, il)
\end{align*}
$$

5.1.2 Making partial-invertibility explicit

An efficient implementation in SPARCL falls out from the above calculation (see Figure 5): the only additions are the types and the use of $\text{pin}$. Recall that $\text{let} \ p^* = e_1 \ \text{in} \ e_2$ is syntactic sugar for $\text{case} \ e_1 \ \text{of} \ (p^* \rightarrow e_2 \ \text{with} \ \lambda . True)$. Recall also that the first match principle is assumed and the catch-all $\text{with}$ conditions for the second branches are omitted. The
function new in the program lifts an \(A\)-typed value \(a\) to an \(A^*\)-typed value, corresponding to a bijection between () and \([a]\).

\[
\text{new} : (a \rightarrow a \rightarrow \text{Bool}) \rightarrow a \rightarrow a^*
\]

\[
\text{new eq } c = \text{lift} (\lambda_\_c . \text{case } eq c c' \text{ of } \{ \text{True } \rightarrow ( )\}) ( )^*
\]

Note that the arguments of lift in new eq form a not-necessarily-total bijection, provided that eq implements the equality on \(A\).

The backward evaluation of \(\pi\text{R}\) has the same behavior as that Mu & Bird (2003, Section 3) derived. The partial bijection that \(\pi\text{ASR}\) defines indeed corresponds to reb in their calculation. Their reb function is introduced as a rather magical step; our calculation can be seen as a justification of their choice.

### 5.1.3 new and delete

In the above example, we used new, which can be used to introduce redundancy to the output. For example, it is common to include checksum information in encoded data. The new function is effective for this scenario, as demonstrated below.

\[
\text{checkSum} : \text{List } \text{Int}^* \rightarrow \text{List } \text{Int}^*
\]

\[
\text{checkSum } xs = \begin{cases} 
\text{let } (xs, s) = \text{pin} xs (\lambda xs'. \text{new eqInt} (\text{sum } xs')) \text{ in } -- \text{sum} : \text{List } \text{Int} \rightarrow \text{Int} \\
\text{Cons}^* s xs
\end{cases}
\]

In the forward direction, checkSum computes the sum of the list and prepends it to the list. In the backward direction, it checks if the head of the input list is the sum of its tail: if the check succeeds, the backward computation of checkSum returns the tail, and (correctly) fails otherwise.

It is worth mentioning that the pattern \(\text{new eq}\) is a finer operation than reversible copying where the inverse is given by equivalence checking (Glück and Kawabe, 2003); reversible copying can be implemented as \(\lambda x. \text{pin} x (\text{new eq}) : A^* \rightarrow (A \otimes A)^*\), assuming appropriate \(eq : A \rightarrow A \rightarrow \text{Bool}\).

The new function has the corresponding inverse delete, which can be used to remove redundancy from the input.

\[
\text{delete} : (a \rightarrow a \rightarrow \text{Bool}) \rightarrow a \rightarrow a^* \rightarrow ( )^*
\]

\[
\text{delete eq } c a = \text{lift} (\lambda c'. \text{case } eq c c' \text{ of } \{ \text{True } \rightarrow ( )\}) (\lambda_\_c) a
\]

It is interesting to note that new and delete can be used to define a safe variant of lift.

\[
\text{safeLift} : (a \rightarrow a \rightarrow \text{Bool}) \rightarrow (b \rightarrow b \rightarrow \text{Bool}) \rightarrow (a \rightarrow b) \rightarrow (b \rightarrow a) \rightarrow a^* \rightarrow b^*
\]

\[
\text{safeLift eqA eqB } f g a = \begin{cases} 
\text{let } (a, b)^* = \text{pin} a (\lambda a'. \text{new eqB} (f a')) \text{ in } \\
\text{let } (b, ())^* = \text{pin} b (\lambda b'. \text{delete eqA} (g b') a) \text{ in } \\
b
\end{cases}
\]

In the forward computation, the function applies \(f\) to the input and tests whether \(g\) is an inverse of \(f\) by applying \(g\) to the output and checking if the result is the same as the original input by eqA. The backward computation does the opposite: it applies \(g\) and tests the result by using \(f\) and eqB. This function is called “safe”, as it guarantees correctness by the runtime check, provided that eqA and eqB implement the equality on the domains.
huffCompress : (List Symbol)^* → (Huff ⊗ List Bit)^*
huffCompress s =
   let (s, h)^* = pin s (λs'. new eqHuff (makeHuff s')) in
   pin h (λh'. encode h' s)

encode : Huff → (List Symbol)^* → (List Bit)^*
encode h Nil^* = Nil^* with null
encode h (Cons s ss)^* = encR h s (encode h ss)

Fig. 6. Two-pass Huffman coding in SPARCL.

5.2 Huffman coding

The Huffman coding is one of the most popular compression algorithms (Salomon, 2008). The idea of the algorithm is to assign short code to frequently occurring symbols. For example, consider that we have symbols \(a\), \(b\), \(c\), and \(d\) that occur in the text to be encoded with probability 0.6, 0.2, 0.1, and 0.1, respectively. If we assign code as \(a:0\), \(b:10\), \(c:110\), and \(d:111\), then a text \(aabacabdaa\) will be encoded into 16-bit code \(001011010010111000\), which is smaller than the 20-bit code obtained under the naive encoding that assigns two bits for each symbol.

5.2.1 Two-pass Huffman coding

Assume that we have a data structure for a Huffman coding table, represented by type Huff. The table may be represented as an array (or arrays) or a tree, and in practice one may want to use different data structures for encoding and decoding (for example, an array for encoding and a trie for decoding). In this case, Huff is a pair of two data structures, where each one is used only in one direction. To handle such a situation, we treat it as an abstract type with the following functions.

\[
\begin{align*}
\text{makeHuff} &: \text{List Symbol} \rightarrow \text{Huff} \\
\text{enc} &: \text{Huff} \rightarrow \text{Symbol} \rightarrow \text{List Bit} \\
\text{dec} &: \text{Huff} \rightarrow \text{List Bit} \rightarrow \text{Symbol} \otimes \text{List Bit}
\end{align*}
\]

Here, \(\text{enc}\) and \(\text{dec}\) satisfy the properties \(\text{dec } h (\text{enc } h s ++ ys) = (s, ys)\) and \(\text{dec } h ys = (s, ys')\) implies \(\text{enc } s ++ ys = ys'\), where ++ is the list append function.

Then, by \(\text{enc}\) and \(\text{dec}\), we can define an bijective version \(\text{encR}\) as below.

\[
\begin{align*}
\text{encR} &: \text{Huff} \rightarrow \text{Symbol}^* \rightarrow (\text{List Bit})^* \rightarrow (\text{List Bit})^* \\
\text{encR} h s r &= \text{lift} (\lambda(s, ys). \text{enc } h s ++ ys) (\lambda ys. \text{dec } h ys) (s, r)^*
\end{align*}
\]

An encoder can be defined by first constructing a Huffman coding table and then encoding symbol by symbol. We can program this procedure in a natural way in SPARCL (Figure 6) by using \text{pin}. This is an example where multiple \text{pins} are used to convert data. The input symbol list is first passed to \text{makeHuff} under \text{new} to create a Huffman table \(h\) in the first \text{pin}; here the input symbol list is unidirectional (static), while the constructed Huffman table is invertible. Then, the input symbol list is encoded with the constructed Huffman table in the second \text{pin}; here the input symbol list is invertible, while the Huffman table is unidirectional (static). A subtlety here is the use of \text{eqHuff} : Huff → Huff → Bool to test the equality of the Huffman encoding tables. This check ensures the property that
\textbf{fwd} \texttt{huffCompress (bwd huffCompress (h, ys)) = (h, ys)}. This equation holds only when \( h \) is the table obtained by applying \texttt{makeHuff} to the decoded text; indeed, \texttt{eqHuff} checks the condition. One could avoid this check by using the following \texttt{unsafeNew} instead.

\begin{align*}
\texttt{unsafeNew} : & \quad a \to a^* \\
\texttt{unsafeNew} a = & \quad \texttt{lift} \left( \lambda() . a \right) (\lambda a'.()) \quad -- \text{assuming} \quad a = a'
\end{align*}

The use of \texttt{unsafeNew} \( a \) is safe only when its backward execution always receives \( a \). Replacing \texttt{new} with \texttt{unsafeNew} violates this assumption, but for this case, the replacement just widens the domain of \texttt{bwd huffCompress}, which is acceptable even though \texttt{fwd huffCompress} and \texttt{bwd huffCompress} do not form a bijection due to \texttt{unsafeNew}. But in general this outcome is unreliable, unless the condition above can be guaranteed.

\section*{5.2.2 Concrete representation of Huffman tree in \textsc{Sparcl}}

In the above we have modeled the case where different data structures are used for encoding and decoding, which demands the use of abstract type and consequently the use of lifting. In this section, we define \texttt{encR} directly in \textsc{Sparcl}, which is possible when the same data structure is used for encoding and decoding.

To do so, we first give a concrete representation of \texttt{Huff}.

\begin{verbatim}
data Huff = Lf Symbol | Br Huff Huff
\end{verbatim}

Here, \texttt{Lf s} encodes \( s \) into the empty sequence, and \texttt{Br l r} encodes \( s \) into \texttt{Cons 0 c} if \( l \) encodes \( s \) to \( c \), and \texttt{Cons 1 c} if \( r \) encodes \( s \) to \( c \). For example, \texttt{Br (Lf 'a') (Br (Lf 'b') (Br (Lf 'c') (Lf 'd')))} is the Huffman tree used to encode the example presented in the beginning of Section 5.2.

Now let us define \texttt{encR} to be used in \texttt{encode} above. It is easier to define it via its inverse \texttt{decR}.

\begin{verbatim}
decR : Huff \to (List Bit)^* \to (Symbol \otimes List Bit)^* \\
\texttt{decR (Lf s) ys} = (\texttt{new eqSym s, ys})^* \\
\texttt{decR (Br l r) ys} = \texttt{case ys of (Cons 0 ys')^* \to decR l ys' with \lambda (s, _).member s l} (\texttt{Cons 1 ys'}^* \to decR r ys'
\end{verbatim}

Here, \texttt{member : Symbol \to Huff \to Bool} is a membership test function. Recall that \texttt{invert} implements inversion of a bijection (Section 2). One can find that searching \( s \) in \( l \) for every recursive call is inefficient, and this cost can be avoided by additional information on \texttt{Br} that makes a Huffman tree a search tree. Another solution is to use different data structures for encoding and decoding as we demonstrated in Section 5.2.1.

\section*{5.2.3 Adaptive Huffman coding}

In the above \texttt{huffCompress}, a Huffman coding table is fixed during compression which requires the preprocessing \texttt{makeHuff} to compute the table. This is sometimes suboptimal: for example, a one-pass method is preferred for streaming while a text could consist of several parts with very different frequency distributions of symbols.
SPARCL: A language for partially invertible computation

huffCompress: (List Symbol)* → (Huff \* List Bit)*
huffCompress = encode initHuff
encode: Huff → (List Symbol)* → (List Bit)*
encode h Nil* = Nil* with null
encode h (Cons s ss)* =
  let (s, r) = pin s (\s'. encode (updHuff s' h) ss) in
  encR h s r

Fig. 7. Adaptive Huffman coding in SPARCL.

Being adaptive means that we have the following two functions instead of makeHuff.

initHuff : Huff  updHuff : Symbol → Huff → Huff

Instead of constructing a Huffman coding table beforehand, the Huffman coding table is constructed and changed throughout compression here.

The updating process of the Huffman coding table is the same in both compression and decompression, which means that SPARCL is effective for writing an invertible and adaptive version of Huffman coding in a natural way (Figure 7). This is another demonstration of the SPARCL’s strength in partial invertibility. Programming the same bijection in a fully invertible language gets a lot more complicated due to the irreversible nature of updHuff.

5.3 Arithmetic coding

The idea of arithmetic coding is to encode the entire message into a single number in the range [0, 1). It achieves this by assigning a range to each symbol and encode the symbol sequence by narrowing the ranges. For example, suppose that symbols a, b, c, and d are assigned with ranges [0, 0.6), [0.6, 0.8), [0.8, 0.9), and [0.9, 1.0). The compression algorithm retains a range [l, r), narrows the range to [l + (r - l)ls, l + (r - l)rs) when it reads a symbol s to which [ls, rs) is associated, and finally yields a real in [l, r). For example, reading a text aabacabdad, the range is narrowed into [0.25258176, 0.2526004224) and a real 0.010000001010101 (in base 2) can be picked. Since the first and last bits are redundant, the number can be represented by a 14-bit code 01000000101010, which is smaller than the 20 bit code produced by the naive encoding. Notice that the code 0 corresponds to multiple texts a, aa, aaa, ... . There are several ways to avoid this ambiguity in decoding; here we assume a special end-of-stream symbol EOS whose range does not appear in the symbol range list.

As a simplification, we only consider ranges defined by rational numbers \( \mathbb{Q} \). Specifically, we assume the following type and functions.

\[
\text{type} \quad \text{Range} = (\mathbb{Q}, \mathbb{Q})
\]

\[
\text{rangeOf} : \text{Symbol} \rightarrow \text{Range}
\]

\[
\text{find} : \text{Range} \rightarrow \mathbb{Q} \rightarrow \text{Symbol}
\]

Here, rangeOf returns a range assigned to a given symbol, and find takes a range and a rational in the range, and returns a symbol of which the subdivision of the range contains the rational. In addition, we will use the following functions.
The narrowing of ranges can be implemented straightforwardly as below.

\[
narrow : \text{Range} \to \text{Range} \to \text{Range}
\]

\[
narrow(l, r)(l_s, r_s) = (l + (r - l) \cdot l_s, l + (r - l) \cdot r_s)
\]

In what follows, \(\narrow\) is used only with \(\text{rangeOf}\). So, we define the following function for convenience.

\[
narrowBySym : \text{Range} \to \text{Symbol} \to \text{Range}
\]

\[
narrowBySym ran s = \narrow ran (\text{rangeOf } s)
\]

These functions satisfy the following property.

\[
narrowBySym(l, r)s = (l', r') \Rightarrow (l \leq l' \land r' \leq r) \land (\forall q \in \mathbb{Q}. l' \leq n < r' \Rightarrow \text{find } (l, r) q = s)
\]

Although \(\lambda s. \narrow(l, r)(\text{rangeOf } s)\) is an injection (provided that \(r - l > 0\)), the arithmetic coding does not use the property in decompression because in decompression the result is a rational number instead of a range.

As the first step, we define a unidirectional version that return a rational instead of a bit sequence for simplicity.

\[
\text{arithComp} : (\text{List Symbol}) \to \mathbb{Q}
\]

\[
\text{arithComp} = \text{encode} (0, 1)
\]

\[
\text{encode} : \text{Range} \to (\text{List Symbol}) \to \mathbb{Q}
\]

\[
\text{encode} (l, r) \text{ Nil} = l
\]

\[
\text{encode} (l, r) (\text{Cons } s \text{ ss}) = \text{encode} (\narrowBySym (l, r) s) \text{ ss}
\]

We can see from the definition that unidirectional and invertible computation is mixed together. On one hand, the second component of the range is nonlinear (discarded when \(\text{encode}\) meets \text{Nil}), meaning that the range must be treated as unidirectional. On the other hand, a rational in the range (here we just use the lower bound for simplicity) goes to the final result of \(\text{arithComp}\), which means that the range should be treated as invertible. The \text{pin} operator could be a solution to the issue. Since we want to use the unidirectional function \(\narrowBySym\), it is natural to \text{pin} the symbol \(s\) to narrow the range, which belongs to the unidirectional world. However, there is a problem. Using \text{pin} produces an invertible product \((\text{Symbol} \otimes \mathbb{Q})^*\) with the symbol remaining in the output. In Huffman coding as we have seen, this is not a problem because the two component are combined as the final product. But here the information of Symbol is redundant as it is already retained by the rational in the second component. We need a way to reveal this redundancy and safely discard the symbol.

The solution lies with the \text{delete} function in Section 5.1.3. For this particular case of the arithmetic coding, the following derived version is more convenient.

\[
\text{deleteBy} : (b \to b \to \text{Bool}) \to (a \to b) \to a^* \to b^* \to a^*
\]

\[
\text{deleteBy } eq f a b = \text{let } (a, (!))^* = \text{pin } a (\lambda a'. \text{delete } eq (f \ a') b) \text{ in } a
\]
By using `deleteBy` with `find` (part of the arithmetic encoding API), we can write an invertible version as below.

\[
\begin{align*}
\text{arithComp} &: (\text{List Symbol})^* \rightarrow \mathbb{Q}^* \\
\text{encode} &: \text{Range} \rightarrow (\text{List Symbol})^* \rightarrow \mathbb{Q}^* \\
\text{encode} (l, r) (\text{Nil})^* &= \text{new } eqQ \ l \ \text{with } eqQ \ l \\
\text{encode} (l, r) (\text{Cons } s \ ss)^* &= \\
&\quad \text{let } (s, q) = \text{pin } s \ \lambda s'. \text{encode} (\text{narrowBySym} (l, r) s') ss \ \text{in} \\
&\quad \text{deleteBy } eqSym (\text{find} (l, r)) q s
\end{align*}
\]

Here, `eqQ` and `eqSym` are equivalence tests on `\mathbb{Q}` and `Symbol`, respectively. The operator `($)`, defined by `($) = \lambda f. \lambda x. f x`, is used to avoid parentheses, which is right-associative and has the lowest precedence unlike function application. The with-condition `enQ l` becomes false for any result from the second branch of `encode`; the assumption on EOS guarantees that `encode` eventually meets EOS and changes the lower bound of the range. It is worth noting that, in this case, the check `eqSym` involved in `deleteBy` always succeeds thanks to the property about `narrowBySym` and `find` above. Thus, we can use the “unsafe” variants of `delete` and `deleteBy` safely here. Also, for this particular case, we can replace `new` with `unsafeNew`, if we admit some unsafety: this replacement just makes `bwd arithComp` accept more inputs than what `fwd arithComp` can return.

As a general observation, programming in a compositional way in SPARCL is easier when a component function, after fixing some arguments, transforms all and only the information of the input to the output. In the Huffman coding example, where a bounded number of bits are transmitted for a symbol, both `enR` and `encode` satisfy this criterion; and as a result, its definition is mostly straightforward. In contrast, in arithmetic coding, even recursive calls of `encode` do not satisfy the criterion, as a single bit of an input could affect an unbounded number of positions in the output, which results in the additional programming effort as we demonstrated in the above.

### 5.4 LZ77 compression

LZ77 (Ziv & Lempel, 1977) and its variant (such as LZ78 and LZSS) are also some of the most popular compression algorithms. The basic idea is to use a string of a fixed length (called a `window`) from the already traversed part of the message as a dictionary and repeatedly replace to be traversed strings with their entries (matching positions and lengths) in the dictionary. To do so, LZ77 maintains two buffers: the window and the look-ahead buffer (Salomon, 2008), where the window is searched for the matching position and length of the string in the look-ahead buffer. When the search succeeds, the algorithm emits the matching position and length and shifts both buffers by the matching length. Otherwise, it emits the first character and shifts the two buffers by one. For example, when the window size is 4 and the look-ahead buffer size is 3, for an window `dabc` and an input

\[17 \text{ Here, we consider an LZSS-flavored variant that emits either a character or a pair of matching position and length, unlike the original one that always emits a triple of matching position, length and the following character even when the matched length is zero.}\]
string abda, the algorithm yields (3, 2)da, as below

\[
\begin{array}{ccc}
\text{dabc} & \text{abcd} & \text{emits} & (3, 2) \\
\text{da} & \text{bcab} & \text{emits} & d \\
\text{dabc} & \text{abda} & \text{emits} & a
\end{array}
\]

where (3, 2) means that the string ab of length 2 appears in the window at position 3 (counted from the last). In general, the end of a matched string may not be in the window but the look-ahead buffer. For example, for \texttt{bbbaaaa} the algorithm emits (1, 4).

The basic idea of our implementation is to use \texttt{pin} to convert the input string from invertible to unidirectional to allow overlapping in searching. Hence, we prepare the following unidirectional functions for the manipulation of the window, which is an abstract type Window.

\[
\begin{align*}
\text{emptyWindow} & : \text{Window} \\
\text{extendWindow} & : \text{Window} \rightarrow \text{List Symbol} \rightarrow \text{Window} \\
\text{findMatch} & : \text{Window} \rightarrow \text{List Symbol} \rightarrow \text{Maybe (Int} \otimes \text{Int)} \\
\text{takeMatch} & : \text{Window} \rightarrow (\text{Int} \otimes \text{Int}) \rightarrow \text{List Symbol}
\end{align*}
\]

Here, the last two functions satisfy the following property.

\[
\text{findMatch w s} = \text{Just} (p, l) \implies \text{takeMatch w} (p, l) = \text{take l s}
\]

Also, we use the following type for the output code.

\[
\text{data LZCode} = \text{Lit Symbol} | \text{Entry} (\text{Int} \otimes \text{Int})
\]

We do not need to represent the look-ahead buffer explicitly, as it is hidden in the \texttt{findMatch} function. Instead of using custom-sized integers, we use \texttt{Int} to represent both matching positions (bounded by the size of the window) and matching lengths (bounded by the size of the look-ahead buffer) for simplicity.

Figure 8 shows an implementation of an invertible LZ77 compression in \texttt{Sparcl}. We omit the definition of eqMatchRes : \texttt{Maybe (Int} \otimes \text{Int}) \rightarrow \texttt{Maybe (Int} \otimes \text{Int}) \rightarrow \texttt{Bool} and eqStr : \texttt{List Symbol} \rightarrow \texttt{List Symbol} \rightarrow \texttt{Bool}. Similarly to the arithmetic coding example, we also use the new/delete trick here. The property above of \texttt{findMatch} and \texttt{takeMatch} ensures that the delete in \texttt{encode} must succeed in the forward evaluation, meaning that we can replace the delete by its unsafe variant similarly to the arithmetic coding example.

It is also similar to the previous examples that the backward evaluation of \texttt{lz77} can only accept the encoded string that the corresponding forward evaluation can produce. This is inconvenient in practice, because there in general are many compression algorithms that correspond to a decompression algorithm. Fortunately, the same solution to the previous examples also apply to this example: for this particular case, replacing \texttt{new} with \texttt{unsafeNew} is widen the domain of the backward execution, without risking the expected behavior that decompression after compression should yield the original data.
lz77 : (List Symbol)• → (List LZCode)•
lz77 = encode emptyWindow
encode : Window → (List Symbol)• → (List LZCode)•
encode w Nil• = Nil• with null
encode w inp• =
  let (inp, matchRes)• = pin inp (λ inp′. new eqMatchRes (findMatch w inp)) in
case matchRes of
  Nothing• → 
    let (Cons s ss)• = inp in
    let (s, r)• = pin s (λ s′. encode (extendWindow w (Cons s′ Nil)) ss) in
    Cons• (revised Lit s) r
with isLit ◦ head
  (Just (p, l))• → 
    let (l, (mstr, rest))• = pin l (λ l′. split l inp) in
    let (mstr, r)• = pin mstr (λ mstr′. encode (extendWindow w mstr′) rest) in
    let (c, ())• = pin (p, l)• (λ c. delete eqStr (takeMatch w c) mstr) in
    Cons• (Entry c) r

split : : Int → (List a)• → (List a)•
split n Nil• = (Nil•, Nil•)• with λ (a, b). null a && null b
split n (Cons a as)• = if n = 0 then (Cons• a as, Nil•)•
else let (t, d)• = split (n − 1) as in (Cons a t, d)•

Fig. 8. LZ77 in SPARCL.

6 Related work
6.1 Program inversion and invertible/reversible computation

In the literature of program inversion (a program transformation technique to find \( f^{-1} \) for a given \( f \)), it is known that an inverse of a function may not arise from reversing all the execution steps of the original program. Partial inversion (Romanenko, 1991; Nishida et al., 2005) addresses the problem by classifying inputs/outputs into known and unknown, where known information is available also for inverses. This classification can be viewed as a binding-time analysis (Gomard & Jones, 1991; Jones et al., 1993) where the known part is treated as static. The partial inversion is further extended so that the return values of inverses are treated as known as well (Almendros-Jiménez & Vidal, 2006; Kirkeby & Glück, 2019, 2020); in this case, it can no longer be explained as a binding-time analysis. This extension introduces additional power, but makes inversion fragile as success depends on which function is inverted first. For example, the partial inversion for \( \text{goSubs} \) succeeds when it inverts \( x − n \) first, but fails if it tried to invert \( \text{goSubs} x xs \) first. The design of SPARCL is inspired by these partial inversion methods: we use (−)•-types to distinguish the known and unknown parts, and pin together with case to control orders. Semi inversion (Mogensen, 2005) essentially converts a program to logic programs and then tries to convert it back to a functional inverse program, which also allows the original and inverse programs to have common computations. Its extension (Mogensen, 2008) can handle a limited form of function arguments. Specifically, such function arguments must be names of top-level functions; neither closures nor partial applications is supported. The Inversion
Framework (Kirkeby & Glück, 2020) unifies the partial and semi inversion methods based on the authors’ reformulation (Kirkeby & Glück, 2019) of semi inversion for conditional constructor term rewriting systems (Terese, 2003). The PINS system allows users to specify control structures as they sometimes differ from the original program (Srivastava et al., 2011). As we mentioned in Section 1, these program inversion methods may fail, and often for reasons that are not obvious to programmers.

Embedded languages can be seen as two-staged (a host and a guest), and there are several embedded invertible/reversible programming languages. A popular approach to implement such languages is based on combinators (Mu et al., 2004b; Rendel & Ostermann, 2010; Kennedy & Vytiniotis, 2012; Wang et al., 2013), in which users program by composing bijections through designated combinators. To the best of our knowledge, only (Kennedy & Vytiniotis, 2012) has an operator like $\text{pin} : A^* \rightarrow\left( A \rightarrow B^*\right) \rightarrow A \otimes B^*$, which is key to partial invertibility. More specifically, Kennedy & Vytiniotis (2012) has an operator $\text{depGame} :: \text{Game} a \rightarrow (a \rightarrow \text{Game} b) \rightarrow \text{Game} (a, b)$. The types suggest that Game and $(-)^*$ play a similar role; indeed they both represent invertibility but in different ways. In their system, Game $a$ represents (total) bijections from bit sequences and $a$-typed values, while in our system $A^*$ represents a bijection whose range is $A$ but domain is determined when $\text{unlift}$ is applied. One consequence of this difference is that, in their domain-specific system, there is no restriction of using a value $v :: \text{Game} a$ linearly, because there is no problem of using an encoder/decoder pair for type $a$ multiple times, even though nonlinear use of $v :: A^*$, especially discarding, leads to non-bijectivity. Another consequence of the difference is that their system is hardwired to bit sequences and therefore does not support deriving general bijections between $a$ and $b$ from Game $a \rightarrow \text{Game} b$, whereas we can obtain a (not-necessarily-total) bijections between $A$ and $B$ from any function of type $A^* \rightarrow B^*$ that does not contain linear free variables.

The $\text{pin}$ operator can be seen as a functional generalization of reversible update statements (Axelsen et al., 2007) $x \oplus= e$ in reversible imperative languages (Lutz, 1986; Frank, 1997; Yokoyama et al., 2008; Glück & Yokoyama, 2016), of which the inverse is given by $x \ominus= e$ with $\ominus$ satisfying $(x \ominus y) \ominus y = x$ for any $y$; examples of $\oplus$ (and $\ominus$) include addition, subtraction, bitwise XOR, and replacement of nil (Glück & Yokoyama, 2016) as a form of reversible copying (Glück and Kawabe, 2003). Having $(x \ominus y) \ominus y$ means that $\oplus$ and $\ominus$ are partially invertible, and indicates that they correspond to the second argument of $\text{pin}$. Whereas the operators such as $\oplus$ and $\ominus$ are fixed in those languages, in SPARCL, leveraging its higher-orderness, any function of an appropriate type can be used as the second argument of $\text{pin}$, which leads to concise function definitions as demonstrated in $\text{goSub}$ in Section 2 and the examples in Section 5.

Most of the existing reversible programming languages (Lutz, 1986; Baker, 1992; Frank, 1997; Mu et al., 2004b; Yokoyama et al., 2008, 2011; Wang et al., 2013) do not support function values, and higher-order reversible programming languages are uncommon. One notable exception is Abramsky (2005) that shows a subset of the linear $\lambda$-calculus concerning $\rightarrow$ and ! (more precisely, a combinator logic that corresponds to the subset) can be interpreted as manipulations of (not-necessarily-total) bijections. However, it is known to be difficult to extend their system to primitives such as constructors and invertible pattern matching (Abramsky, 2005, Section 7). Abramsky (2005)’s idea is based on
the fact that a certain linear calculus is interpreted in a compact closed category, which
has a dual object \( A^* \) such that \( A^* \otimes B \) serves as a function (i.e., internal hom) object, and
that we can construct (Joyal et al., 1996) a compact closed category from the category
of not-necessary-total bijections (Abramsky et al., 2002). Recently, Chen & Sabry (2021)
designed a language that has fractional and negative types inspired by compact closed
categories. In the language, a negative type \(-A\) is a dual of \( A \) for \( \oplus \), and constitutes a
“function” type \(-A \oplus B\) that satisfies the isomorphism \( A \oplus B \leftrightarrow C \cong A \leftrightarrow -B \oplus C \), where \( \leftrightarrow \) denotes bijections. One of the applications of the negative type is to define a loop like
operation called the trace operator, which has a similar behavior to \textit{trace} in Section 3.6.4.
The fractional types in the language are indexed by values as \( 1/(v : A) \), which represents
the obligation to erase an ancilla value \( v \), and hence the corresponding application form
does perform the erasure. However, behavior of both \(-A \oplus B\) and \( 1/(v : A) \otimes B\) is different
from what we expect for functions: the former operates on \( \oplus \) instead of \( \otimes \), and the latter
only accepts the input \( v \).

A few reversible functional programming languages also support a limited form of
partial invertibility. RFunT,\(^18\) a typed variant of RFun (Yokoyama et al., 2011) with
Haskell-like syntax, allows a function to take additional parameters called ancilla param-
eters. The reversibility restriction is relaxed for ancilla parameters, and they can be
discarded and pattern-matched without requiring a way to determine branching from
their results. However, these ancilla parameters are supposed to be translated into auxiliary inputs and outputs that stay the same before and after reversible computation, and
mixing unidirectional computation is not their primary purpose. In fact, very limited oper-
ations are allowed for these ancilla data by the system. CoreFun also supports ancilla parameters (Jacobsen et al., 2018). Their ancilla parameters are treated as static inputs
to reversible functions, and arguments that appear at ancilla positions are free from the
linearity restriction.\(^19\) The system is overly conservative: all the functions are (partially)
reversible, and thus functions themselves used in the ancilla positions must obey the lin-
earity restriction. Jeopardy (Kristensen et al., 2022b)\(^20\) is a work-in-progress reversible
language, which plans to support partial invertibility via program analysis. The implicit
argument analysis (Kristensen et al., 2022a), which Jeopardy uses, identifies which argu-
ments are available (or, known (Nishida et al., 2005)) for each functional call and for the
forward/backward execution. However, the inverse execution based on the analysis has
neither been formalized nor implemented to the best of the authors’ knowledge. More cru-
cially, RFunT, CoreFun and Jeopardy are first-order languages (to be precise, they allow
top-level function names to be used as values, but not partial application or \( \lambda \)-abstraction),
which limits flexible programming. In contrast, \( A^* \) is an ordinary type in \textsc{sparcl}, and
there is no syntactic restriction on expressions of type \( A^* \). This feature, combined with
the higher-orderness, gives extra flexibility in mixing unidirectional and invertible pro-
gramming. For example, \textsc{sparcl} allows a function composition operator that can be used
for both unidirectional (hence unrestricted) and invertible (hence linear) functions, using
multiplicity polymorphism (Bernardy et al., 2018; Matsuda, 2020).

\(^18\) https://github.com/kirkedal/rfun-interp.
\(^19\) A correction to Jacobsen et al. (2018) (personal communication with Michael-Kirkedal Thomsen, Jun 2020).
\(^20\) Don’t confuse it with the program inversion method with the same name (Dershowitz & Mitra, 1999).
6.2 Functional quantum programming languages

In quantum programming, many operations are reversible, and there are a few higher-order quantum programming languages (Selinger & Valiron, 2006; Rios & Selinger, 2017). Among them, the type system of Proto-Quipper-M (Rios & Selinger, 2017) is similar to \( \lambda^\Pi \) in the sense that it also uses a linear-type system and distinguishes two sorts of variable environments as we do with \( \Gamma \) and \( \Theta \), although the semantic back-ends are different. They do not have any language construct that introduces new variables to the second sort of environments (a counterpart of our \( \Theta \)), because their language does not have a counterpart to our invertible case.

It is also interesting to see that some quantum languages allow weakening (i.e., discarding) (Selinger & Valiron, 2006) and some allow contraction (i.e., copying) (Altenkirch & Grattage, 2005). In these frameworks, weakening is allowed because one can throw away a quantum bit after measuring, and contraction is allowed because states can be shared through introducing entanglements. As our goal is to obtain a bijection as final product, weakening in general is not possible in our context. On the other hand, it is a design choice whether or not contraction is allowed. Since the inverse of copying can be given by equivalence checking and vice versa (Glück and Kawabe, 2003). However, careless uses of copying may result in unintended domain restriction. Moreover supporting such a feature requires hard-wired equivalence checks for all types of variables that can be in \( \Theta \) (notice that multiple uses of a variable in \( \Gamma \) will be reduced to multiple uses of variables in \( \Theta \) (Matsuda & Wang, 2018c)). This requires the type system to distinguish types that can be in \( \Theta \) from general ones, as types such as \( A \to B \) do not have decidable equality. Moreover, the hard-wired equivalence checks would prevent users from using abstract types such as Huff in Section 5, for which the definition of equivalence can differ from that on their concrete representations.

6.3 Bidirectional programming languages

It is perhaps not surprising that many of the concerns in designing invertible/bijective/reversible languages are shared by the closely related field of bidirectional programming (Foster et al., 2007). A bidirectional transformation is a generalization of a pair of inverses that allows a component to be non-bijective; for example, an (asymmetric) bidirectional transformation between \( a \) and \( b \) are given by two functions called \( \text{get} : a \to b \) and \( \text{put} : a \to b \to a \) (Foster et al., 2007). Similarly to ours, in the bidirectional language HOBiT (Matsuda & Wang, 2018c), a bidirectional transformation between \( a \) and \( b \) is represented by a function from \( B a \) to \( B b \), and top-level functions of type \( B a \to B b \) can be converted to a bidirectional transformation between \( a \) and \( b \). Despite the similarity, there are unique challenges in invertible programming: notably, the handling of partial invertibility that this paper focuses on and the introduction of the operator \( \text{pin} \) as a solution. Another difference is that \( \text{SPARCL} \) is based on a linear type system, which, as we have seen, perfectly supports the need for the intricate connections between unidirectional and inverse computation in addressing partial invertibility. One of the consequences of this difference in the underlying type system is that Matsuda & Wang (2018c) can only interpret top-level functions of type \( B a \to B b \) as bidirectional transformations between \( a \) and \( b \), yet we can
interpret functions of type $A^* \to B^*$ in any places as bijections between $A$ and $B$, as long as they have no linear free variables. Linear types also clarify the roles of values and prevent users from unintended failures caused by erroneous use of variables. For example, the type $A^* \to (A \to B^*) \to (A \otimes B^*)^*$ of $\text{pin}$ clarifies that the function argument of $\text{pin}$ can safely discard or copy its input as the nonlinear uses do not affect the domain of the resulting bijection.

It is worth mentioning that, in addition to bidirectional transformations, HOBiT provides a way to lift bidirectional combinators (i.e., functions that take and return bidirectional transformations). However, the same is not obvious in SPARCL due to its linear type system, as the combinators need to take care of the manipulation of $\Theta$ environments such as splitting $\Theta = \Theta_1 + \Theta_2$. On the other hand, there is less motivation to lift combinators in the context of bijective/reversible programming especially for languages that are expressive enough to be reversible Turing complete (Bennett, 1973).

The applicative-lens framework (Matsuda & Wang, 2015a, 2018a), which is an embedded domain-specific language in Haskell, provides a function $\text{lift}$ that converts a bidirectional transformation $(a \to b, a \to b \to a)$ to a function of type $L s a \to L s b$ where $L$ is an abstract type parameterized by $s$. As in HOBiT, bidirectional transformations are represented as functions so that they can be composed by unidirectional functions; the name applicative in fact comes from the applicative (point-wise functional) programming style. (To be precise, $L$ together with certain operations forms a lax monoidal functor (Mac Lane, 1998, Section XI.2) as Applicative instances (McBride & Paterson, 2008; Paterson, 2012) but not endo to be an Applicative instance (Matsuda & Wang, 2018a).) The type parameter $s$ has a similar role to the $s$ of the ST $s$ monad (Launchbury & Jones, 1994), which enables the unfolding that converts a polymorphic function $\forall s. L s a \to L s b$ back to a bidirectional transformation $(a \to b, a \to b \to a)$. That is, unlike HOBiT, functions that will be interpreted as bidirectional transformations are not limited to top-level ones. However, in exchange for this utility, the expressive power of the applicative lens is limited compared with HOBiT; for example, bidirectional cases are not supported in the framework, and resulting bidirectional transformations cannot propagate structural updates as a result.

As a remark, duplication (contraction) of values is also a known challenge in bidirectional transformation, for the purpose of supporting multiple views of the same data and synchronization among them (Hu et al., 2004). However, having unrestricted duplication makes compositional reasoning of correctness very difficult; in fact most of the fundamental properties of bidirectional transformation, including well-behavedness (Foster et al., 2007) and its weaker variants (Mu et al., 2004a; Hidaka et al., 2010), are not preserved in the presence of unrestricted duplication (Matsuda & Wang, 2015b).

### 6.4 Linear type systems

SPARCL is based on $\lambda^L_\omega$, a core system of Linear Haskell (Bernardy et al., 2018), with qualified typing (Jones, 1995; Vytiniotis et al., 2011) for effective inference (Matsuda, 2020). An advantage of this system is that the only place where we need to explicitly handle linearity is the manipulation of $(-)^*$-typed values; there is no need of any special annotations for the unidirectional parts, as demonstrated in the examples. This is different from Wadler (1993)’s linear type system, which would require a lot of $!$ annotations in
the code. Linear Haskell is not the only approach that is able to avoid the scattering of !s. Mazurak et al. (2010) use kinds (◦ and ∗) to distinguish types that are treated in a linear way (◦) from those that are not (∗). Thanks to the subkinding ∗ ≤ ◦, no syntactic annotations are required to convert the unrestricted values to linear ones. Their system has two sort of function types: → ◦ for the functions that themselves are treated in the linear way and → ∗ for the functions that are unrestricted. As a result, a function can have multiple incomparable types; e.g., the K combinator can have four types (Morris, 2016). Universal types accompanied by kind abstraction (Tov & Pucella, 2011) addresses the issue to some extent; it works well especially for K, but still gives the B combinator two incomparable types (Morris, 2016). Morris (2016) further extends these two systems to overcome the issue by using qualified types (Jones, 1995), which can infer principal types thank to inequality constraints. Note that the implementation of SPARCL uses an inference system by Matsuda (2020), which, based on OUTSIDE IN(X) (Vytiniotis et al., 2011), also uses qualified typing with inequality constraints for λq, inspired by Morris (2016).

7 Conclusion

We have designed SPARCL, a language for partially invertible computation. The key idea of SPARCL is to use types to distinguish data that are subject to invertible computation and those that are not; specifically the type constructor (−)∗ is used for marking the former. A linear type system is utilized for connecting the two worlds. We have presented the syntax, type system, and semantics of SPARCL and proved that invertible computations defined in SPARCL are in fact invertible (and hence bijective). To demonstrate the utility of our proposed language, we have proved its reversible Turing completeness and presented non-trivial examples of tree rebuilding and three compression algorithms (Huffman coding, arithmetic coding, and LZ77).

There are several future directions of this research. One direction is to use finer type systems. Recall that we need to check with conditions even in the forward computation, which can be costly. We believe that refinement types and their inference (Xi & Pfenning, 1998; Rondon et al., 2008) would be useful for addressing this issue. Currently, our prototype implementation is standalone, preventing users from writing functions in another language to be used in lift, and from using functions obtain by fwd and bwd in the other language. Although prototypical implementation of a compiler of SPARCL to Haskell is in progress, a seamless integration through an embedded implementation would be desirable (Matsuda & Wang, 2018b). Another direction is to extend our approach to bidirectional transformations (Foster et al., 2007) to create the notion of partially bidirectional programming. As discussed in Section 6, handling copying (i.e., contraction) is an important issue; we want to find the sweet spot of allowing flexible copying without compromising reasoning about correctness.

Acknowledgments

We thank the IFIP 2.1 members for their critical but constructive comments on a preliminary version of this research, Anders Ågren Thuné for the LZ77 example in Section 5.4 and finding bugs in our prototype implementation and Agda proofs since the publication of
the conference version, and Samantha Frohlich for her helpful suggestions and comments on the presentation of this paper. We also thank the anonymous reviewers of ICFP 2020 for their constructive comments. This work was partially supported by JSPS KAKENHI Grant Numbers JP15H02681, JP19K11892, JP20H04161 and JP22H03562, JSPS Bilateral Program, Grant Number JPJSBP120199913, the Kayamori Foundation of Informational Science Advancement, EPSRC Grant EXHIBIT: Expressive High-Level Languages for Bidirectional Transformations (EP/T008911/1), and Royal Society Grant Bidirectional Compiler for Software Evolution (IES\R3\170104).

Conflict of Interests

None.

References

Abel, A. & Chapman, J. (2014) Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014, pp. 51–67.

Abramov, S. M., Glück, R. & Klimov, Y. A. (2006) An universal resolving algorithm for inverse computation of lazy languages. In Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27–30, 2006. Revised Papers, Virbitskaite, I. & Voronkov, A. (eds), Lecture Notes in Computer Science, vol. 4378. Springer, pp. 27–40.

Abramsky, S. (2005) A structural approach to reversible computation. Theor. Comput. Sci. 347(3), 441–464.

Abramsky, S., Haghverdi, E. & Scott, P. J. (2002) Geometry of interaction and linear combinatory algebras. Math. Struct. Comput. Sci. 12(5), 625–665.

Almendros-Jiménez, J. M. & Vidal, G. (2006) Automatic partial inversion of inductively sequential functions. In Implementation and Application of Functional Languages, 18th International Symposium, IFL 2006, Budapest, Hungary, September 4–6, 2006, Revised Selected Papers. Springer, pp. 253–270.

Altenkirch, T., Chapman, J. & Uustalu, T. (2010) Monads need not be endofunctors. In Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20–28, 2010. Proceedings. Springer, pp. 297–311.

Altenkirch, T. & Grattage, J. (2005) A functional quantum programming language. In 20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26–29 June 2005, Chicago, IL, USA, Proceedings. IEEE Computer Society, pp. 249–258.

Antoy, S., Echahed, R. & Hanus, M. (2000) A needed narrowing strategy. J. ACM 47(4), 776–822.

Axelsen, H. B., Glück, R. & Yokoyama, T. (2007) Reversible machine code and its abstract processor architecture. In Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3–7, 2007, Proceedings. Springer, pp. 56–69.

Baker, H. G. (1992) NREVERSAL of fortune - the thermodynamics of garbage collection. In Memory Management, International Workshop IWMM 92, St. Malo, France, September 17–19, 1992, Proceedings. Springer, pp. 507–524.

Bennett, C. H. (1973) Logical reversibility of computation. IBM J. Res. Dev. 17(6), 525–532.

Bernardy, J., Boespflug, M., Newton, R. R., Peyton Jones, S. & Spiwack, A. (2018) Linear haskell: Practical linearity in a higher-order polymorphic language. PACMPL 2(POPL), 5:1–5:29.
Capretta, V. (2005) General recursion via coinductive types. *Logical Methods Comput. Sci.* **1**(2), Article number 1.

Chen, C. & Sabry, A. (2021) A computational interpretation of compact closed categories: Reversible programming with negative and fractional types. *Proc. ACM Program. Lang.* **5**(POPL), 1–29.

Chin, W. (1993) Towards an automated tupling strategy. In Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM’93, Copenhagen, Denmark, June 14–16, 1993. ACM, pp. 119–132.

Davies, R. & Pfennig, F. (2001) A modal analysis of staged computation. *J. ACM* **48**(3), 555–604.

Dershowitz, N. & Mitra, S. (1999) Jeopardy. In Rewriting Techniques and Applications, 10th International Conference, RTA-99, Trento, Italy, July 2–4, 1999. Proceedings. Springer, pp. 16–29.

Eppstein, D. (1985) A heuristic approach to program inversion. In IJCAI, pp. 219–221.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C. & Schmitt, A. (2007) Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* **29**(3), Article number 17.

Frank, M. P. (1997) The R programming language and compiler. MIT Reversible Computing Project Memo #M8, MIT AI Lab. Available on: https://github.com/mikepfrank/Rlang-compiler/blob/master/docs/MIT-RCP-MemoM8-RProgLang.pdf.

Gibbons, J. (2002) Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Backhouse, R., Crole, R. & Gibbons, J. (eds). Lecture Notes in Computer Science, vol. 2297. Springer-Verlag, pp. 148–203. Available at: http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/acmmpc-calcfp.pdf.

Glück, R. & Kawabe, M. (2003) A program inverter for a functional language with equality and constructors. In Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27–29, 2003, Proceedings, Ohori, A. (ed). Lecture Notes in Computer Science, vol. 2895. Springer, pp. 246–264.

Glück, R. & Kawabe, M. (2004) Derivation of deterministic inverse programs based on LR parsing. In FLOPS. Springer, pp. 291–306.

Glück, R. & Yokoyama, T. (2016) A linear-time self-interpreter of a reversible imperative language. *Comput. Softw.* **33**(3), 3_108–3_128.

Glück, R. & Yokoyama, T. (2019) Constructing a binary tree from its traversals by reversible recursion and iteration. *Inf. Process. Lett.* **147**, 32–37.

Gomard, C. K. & Jones, N. D. (1991) A partial evaluator for the untyped lambda-calculus. *J. Funct. Program.* **1**(1), 21–69.

Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K. & Nakano, K. (2010) Bidirectionalizing graph transformations. In Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27–29, 2010. ACM, pp. 205–216.

Hu, Z., Iwasaki, H., Takeuchi, M. & Takano, A. (1997) Tupling calculation eliminates multiple data traversals. In Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP’97), Amsterdam, The Netherlands, June 9–11, 1997. ACM, pp. 164–175.

Hu, Z., Mu, S. & Takeuchi, M. (2004) A programmable editor for developing structured documents based on bidirectional transformations. In Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2004, Verona, Italy, August 24–25, 2004. ACM, pp. 178–189.

Jacobsen, P. A. H., Kaarsgaard, R. & Thomsen, M. K. (2018) CoreFun: A typed functional reversible core language. In Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12–14, 2018, Proceedings. Springer, pp. 304–321.

James, R. P. & Sabry, A. (2012) Information effects. In Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22–28, 2012. ACM, pp. 73–84.
SPARC L: A language for partially invertible computation

Jones, M. P. (1995) Qualified Types: Theory and Practice. New York, NY, USA: Cambridge University Press.

Jones, N. D., Gomard, C. K. & Sestoft, P. (1993) Partial Evaluation and Automatic Program Generation. Prentice Hall International Series in Computer Science. Prentice Hall.

Joyal, A., Street, R. & Verity, D. (1996) Traced monoidal categories. Math. Proc. Cambridge Philos. Soc. 119(3), 447–468.

Kennedy, A. J. & Vytiniotis, D. (2012) Every bit counts: The binary representation of typed data and programs. J. Funct. Program. 22(4–5), 529–573.

Kirkeby, M. H. & Glück, R. (2019) Semi-inversion of conditional constructor term rewriting systems. In Logic-Based Program Synthesis and Transformation - 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8–10, 2019, Revised Selected Papers. Springer, pp. 243–259.

Kirkeby, M. H. & Glück, R. (2020) Inversion framework: Reasoning about inversion by conditional term rewriting systems. In PPDP’20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9–10 September, 2020. ACM, pp. 9:1–9:14.

Kristensen, J. T., Kaarsgaard, R. & Thomsen, M. K. (2022a) Branching execution symmetry in jeopardy by available implicit arguments analysis. CoRR. abs/2212.03161.

Kristensen, J. T., Kaarsgaard, R. & Thomsen, M. K. (2022b) Jeopardy: An invertible functional programming language. CoRR. abs/2209.02422.

Kühnemann, A., Glück, R. & Kakehi, K. (2001) Relating accumulative and non-accumulative functional programs. In RTA. Springer, pp. 154–168.

Landauer, R. (1961) Irreversibility and heat generation in the computing process. IBM J. Res. Dev. 5(3), 183–191.

Launchbury, J. & Jones, S. L. P. (1994) Lazy functional state threads. In Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20–24, 1994. ACM, pp. 24–35.

Lutz, C. (1986) Janus: A time-reversible language. Letter to R. Landauer. Available on: http://tetsuo.jp/ref/janus.pdf.

Mac Lane, S. (1998) Categories for the Working Mathematician, 2nd ed. Graduate Texts in Mathematics, vol. 5. Springer.

Matsuda, K. (2020) Modular inference of linear types for multiplicity-annotated arrows. In Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings. Springer. pp. 456–483. The full version is available on: http://arxiv.org/abs/1911.00268v2.

Matsuda, K., Hu, Z., Nakano, K., Hamana, M. & Takeichi, M. (2007) Bidirectionalization transformation based on automatic derivation of view complement functions. In Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1–3, 2007. ACM, pp. 47–58.

Matsuda, K., Inaba, K. & Nakano, K. (2012) Polynomial-time inverse computation for accumulative functions with multiple data traversals. Higher-Order Symb. Comput. 25(1), 3–38.

Matsuda, K., Mu, S.-C., Hu, Z. & Takeichi, M. (2010) A grammar-based approach to invertible programs. In ESOP. Springer, pp. 448–467.

Matsuda, K. & Wang, M. (2013) FliPpr: A prettier invertible printing system. In ESOP. Springer, pp. 101–120.

Matsuda, K. & Wang, M. (2015a) Applicative bidirectional programming with lenses. In ICFP. ACM, pp. 62–74.

Matsuda, K. & Wang, M. (2015b) “Bidirectionalization for free” for monomorphic transformations. Sci. Comput. Program. 111, 79–109.

Matsuda, K. & Wang, M. (2018a) Applicative bidirectional programming: Mixing lenses and semantic bidirectionalization. J. Funct. Program. 28, e15.

Matsuda, K. & Wang, M. (2018b) Embedding invertible languages with binders: A case of the FliPpr language. In Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27–17, 2018. ACM, pp. 158–171.
Matsuda, K. & Wang, M. (2018c) HOBiT: Programming lenses without using lens combinators. In ESOP. Springer, pp. 31–59.
Matsuda, K. & Wang, M. (2020) Sparcl: A language for partially-invertible computation. Proc. ACM Program. Lang. 4(ICFP), 118:1–118:31.
Mazurak, K., Zhao, J. & Zdancewic, S. (2010) Lightweight linear types in system fdegree. In TLDI. ACM, pp. 77–88.
McBride, C. & Paterson, R. (2008) Applicative programming with effects. J. Funct. Program. 18(1), 1–13.
Mogensen, T.Æ. (2005) Semi-inversion of guarded equations. In Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29–October 1, 2005, Proceedings. Springer, pp. 189–204.
Mogensen, T.Æ. (2006) Report on an implementation of a semi-inverter. In Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27–30, 2006. Revised Papers, Virbitskaite, I. & Voronkov, A. (eds). Lecture Notes in Computer Science, vol. 4378. Springer, pp. 322–334.
Mogensen, T.Æ. (2008) Semi-inversion of functional parameters. In Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7–8, 2008. ACM, pp. 21–29.
Moggi, E. (1998) Functor categories and two-level languages. In Foundations of Software Science and Computation Structure, First International Conference, FoSSaCS’98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS’98, Lisbon, Portugal, March 28–April 4, 1998, Proceedings. Springer, pp. 211–225.
Morris, J. G. (2016) The best of both worlds: linear functional programming without compromise. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016. ACM, pp. 448–461.
Mu, S. & Bird, R. S. (2003) Rebuilding a tree from its traversals: A case study of program inversion. In Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27–29, 2003, Proceedings, Ohori, A. (ed). Lecture Notes in Computer Science, vol. 2895. Springer, pp. 265–282.
Mu, S., Hu, Z. & Takeichi, M. (2004a) An algebraic approach to bi-directional updating. In Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4–6, 2004. Proceedings. Springer, pp. 2–20.
Mu, S., Hu, Z. & Takeichi, M. (2004b) An injective language for reversible computation. In Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12–14, 2004, Proceedings. Springer, pp. 289–313.
Nielsen, F. & Nielson, H. R. (1992) Two-Level Functional Languages. Cambridge Tracts in Theoretical Computer Science. Cambridge University.
Nishida, N., Sakai, M. & Sakabe, T. (2005) Partial inversion of constructor term rewriting systems. In Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19–21, 2005. Proceedings. Springer, pp. 264–278.
Nishida, N. & Vidal, G. (2011) Program inversion for tail recursive functions. In RTA. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 283–298.
Ohori, A. (ed) (2003) Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27–29, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2895. Springer.
Paterson, R. (2012) Constructing applicative functors. In MPC. Springer, pp. 300–323.
Rendel, T. & Ostermann, K. (2010) Invertible syntax descriptions: Unifying parsing and pretty printing. In Haskell. ACM, pp. 1–12.
Reynolds, J. C. (1998) Definitional interpreters for higher-order programming languages. Higher-Order Symb. Comput. 11(4), 363–397.
Rios, F. & Selinger, P. (2017) A categorical model for a quantum circuit description language. In Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3–7 July 2017, pp. 164–178.
Romanenko, A. (1991) Inversion and metacomputation. In Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM’91, Yale University, New Haven, Connecticut, USA, June 17–19, 1991. ACM, pp. 12–22.

Rondon, P. M., Kawaguchi, M. & Jhala, R. (2008) Liquid types. In Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7–13, 2008. ACM, pp. 159–169.

Salomon, D. (2008) *A Concise Introduction to Data Compression*. Undergraduate Topics in Computer Science. Springer.

Selinger, P. & Valiron, B. (2006) A lambda calculus for quantum computation with classical control. *Math. Struct. Comput. Sci.* 16(3), 527–552.

Srivastava, S., Gulwani, S., Chaudhuri, S. & Foster, J. S. (2011) Path-based inductive synthesis for program inversion. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4–8, 2011. ACM, pp. 492–503.

Terese. (2003) *Term Rewriting Systems*. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University.

Tov, J. A. & Pucella, R. (2011) Practical affine types. In Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011. ACM, pp. 447–458.

Virbitskaite, I. & Voronkov, A. (eds) (2007) Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, June 27–30, 2006. Revised Papers. Lecture Notes in Computer Science, vol. 4378. Springer.

Vytiniotis, D., Peyton Jones, S. L., Schrijvers, T. & Sulzmann, M. (2011) *OutsideIn(X)* modular type inference with local assumptions. *J. Funct. Program.* 21(4–5), 333–412.

Wadler, P. (1993) A taste of linear logic. In Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS’93, Gdansk, Poland, August 30–September 3, 1993, Proceedings. Springer, pp. 185–210.

Walker, D. (2004) Substructral type systems. In *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (ed). MIT, pp. 3–43.

Wang, M., Gibbons, J., Matsuda, K. & Hu, Z. (2013) Refactoring pattern matching. *Sci. Comput. Program.* 78(11), 2216–2242. Special section on Mathematics of Program Construction (MPC 2010) and Special section on methodological development of interactive systems from Interaccion 2011.

Xi, H. & Pfenning, F. (1998) Eliminating array bound checking through dependent types. In Proceedings of the ACM SIGPLAN’98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17–19, 1998. ACM, pp. 249–257.

Yokoyama, T., Axelsen, H. B. & Glück, R. (2008) Principles of a reversible programming language. In Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5–7, 2008. ACM, pp. 43–54.

Yokoyama, T., Axelsen, H. B. & Glück, R. (2011) Towards a reversible functional language. In RC. Springer, pp. 14–29.

Yokoyama, T., Axelsen, H. B. & Glück, R. (2012) Optimizing reversible simulation of injective functions. *Multiple-Valued Logic Soft Comput.* 18(1), 5–24.

Ziv, J. & Lempel, A. (1977) A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23(3), 337–343.

---

**A Appendix: Proof of the reversible Turing completeness**

As we mentioned before, the proof will be done by implementing a given reversible Turing machine. We follow Yokoyama et al. (2008) for the construction except the last step. For convenience, we shall use SPARCL instead of $\lambda_{\Pi}$ for construction, but the discussions in this section can be adapted straightforwardly to $\lambda_{\Pi}$. 
Following Yokoyama et al. (2008) means that we basically do not make use of the partial invertibility in the implementation, which is unsurprising as a reversible Turing machine is fully-invertible by nature. A notable exception is the last step, which uses a general looping operator represented as a higher-order function, where function parameters themselves are static (i.e., unidirectional).

A.1 Reversible Turing machines

We start with reviewing ordinary Turing machines.

**Definition A.1** (Turing Machine). A (nondeterministic) Turing machine is a 5-tuple \((Q, \Sigma, \delta, q_0, q_f)\) where \(Q\) is a finite set of states, \(\Sigma\) is a finite set of symbols, \(\delta\) is a finite set of transition rules whose element has a form \((q_1, (\sigma_1, \sigma_2), q_2)\) or \((q_1, d, q_2)\) where \(q_1, q_2 \in Q\), \(q_1 \neq q_f, q_2 \neq q_0, \sigma_1, \sigma_2 \in \Sigma\) and \(d \in \{-1, 0, 1\}\), \(q_0 \in Q\) is the initial state and \(q_f \in Q\) is the final state.

We assume that \(\Sigma\) contains a special symbol \(\_\) called blank. A Turing machine, with a state and a head on a tape with no ends, starts with the initial state \(q_0\) and a tape with the finite non-black cells and repeats transitions accordingly to the rules \(\delta\) until it reaches the final state \(q_f\). Intuitively, a rule \((q_1, (\sigma_1, \sigma_2), q_2)\) states that, if the current state of a machine is \(q_1\) and the head points to the cell containing \(\sigma_1\), then it writes \(\sigma_2\) to the cell and changes the current state to \(q_2\). A rule \((q_1, d, q_2)\) states that, if the current state of a machine is \(q_1\) and its head is located at position \(i\) in the tape, then it moves the head to the position \(i + d\) and changes the state to \(q_2\). A reversible Turing machine is a Turing machine whose transitions are deterministic both forward and backward.

**Definition A.2** (Reversible Turing Machine (Bennett, 1973; Yokoyama et al., 2008)). A reversible Turing machine is a Turing machine \((Q, \Sigma, \delta, q_0, q_f)\) satisfying the following conditions for any distinct rules \((q_1, a, q_2)\) and \((q'_1, a', q'_2)\).

- If \(q_1 = q'_1\), then \(a\) and \(a'\) must have the forms \((\sigma_1, \sigma_2)\) and \((\sigma'_1, \sigma'_2)\), respectively, and \(\sigma_1 \neq \sigma'_1\).
- If \(q_2 = q'_2\), then \(a\) and \(a'\) must have the forms \((\sigma_1, \sigma_2)\) and \((\sigma'_1, \sigma'_2)\), respectively, and \(\sigma_2 \neq \sigma'_2\).

A.2 Programming a reversible Turing machine

Consider a given reversible Turing machine \((Q, \Sigma, \delta, q_0, q_f)\). We first prepare types used for implementing the given reversible Turing machine. We assume types \(T_Q\) and \(T_{\Sigma}\) for states and symbols, and \(Q_{\sigma} : T_Q\) and \(S_{\sigma} : T_{\Sigma}\) for constructors corresponding to \(q \in Q\) and \(\sigma \in \Sigma\), respectively. Then, a type for tapes is give by a product \(\text{Tape} = \text{List} T_{\Sigma} \otimes T_{\Sigma} \otimes \text{List} T_{\Sigma}\), where a triple \((l, a, r) : \text{Tape}\) means that \(a\) is the symbol at the current head, \(l\) is the symbols to the left of the head, and \(r\) is the symbols to the right to the head. For uniqueness of the representation, the last elements of \(l\) and \(r\) are assumed not to be \(S_{\_}\) if they are not empty.
Then, we prepare the function $moveR$ below that moves the head to the right.

$$moveR : \text{Tape}^* \rightarrow \text{Tape}^*$$

$$moveR (l, a, r)^* = \text{let} \ (a', l')^* = \text{invert} \ \text{push} \ l \ \text{in} \ \text{push} \ (S, \text{Nil})^* = \text{Nil}^* \ \text{with} \ \text{null} \ \ (l', a', \text{push} \ (a, r)^*)^* \ \text{push} \ (a, xs)^* = \text{Cons}^* \ a \ xs$$

Here, $(-, \ldots, -, -)^*$ is a lifted version of the tuple constructor $(-, \ldots, -)$, let $p^* = e$ in $e'$ is a shorthand notation for case $e$ of $\{ p^* \rightarrow e' \ \text{with} \ \lambda_e. \text{True} \}$, and the function $\text{invert} : (A^* \rightarrow B^*) \rightarrow B^* \rightarrow A^*$ implements the inversion of a invertible function (Section 2).

Then, we define the one-step transition of the given reversible Turing machine.

$$\text{step} : (T_Q \odot \text{Tape})^* \rightarrow (T_Q \odot \text{Tape})^*$$

$$\text{step} \ t = \text{case} \ t \ \text{of} \ ([r])_{r \in \delta}$$

Here, the translation $[r]$ of each rule $r$ is defined as below.

$$[[q_1, (\sigma_1, \sigma_2), q_2]] = (Q_{q_1}, (l, S_{\sigma_1}, r))^* \rightarrow (Q_{q_2}, (l, S_{\sigma_2}, r))^*$$

$$\ \text{with} \ \lambda (q, (\_ , s, \_)).\text{is}_{Q_{q_2}} q \ k \& \text{is}_{S_{\sigma_2}} s$$

$$[[q_1, 0, q_2]] = (Q_{q_1}, t)^* \rightarrow (Q_{q_2}, t)^* \ \text{with} \ \lambda (q, \_).\text{is}_{Q_{q_2}} q$$

$$[[q_1, 1, q_2]] = (Q_{q_1}, t)^* \rightarrow (Q_{q_2}, \text{moveR} \ t)^* \ \text{with} \ \lambda (q, \_).\text{is}_{Q_{q_2}} q$$

$$[[q_1, -1, q_2]] = (Q_{q_1}, t)^* \rightarrow (Q_{q_2}, \text{invert} \ \text{moveR} \ t)^* \ \text{with} \ \lambda (q, \_).\text{is}_{Q_{q_2}} q$$

Here, $\text{is}_{Q_{q}} : T_Q \rightarrow \text{Bool}$ is a function that returns True for $Q_{q}$ and False otherwise, and $\text{is}_{S_{\sigma}} : T_{\Sigma} \rightarrow \text{Bool}$ is similar but defined for symbols. Notice that, by the reversibility of the Turing machine, patterns are nonoverlapping and at most one with-condition becomes True.

The last step is to apply step repeatedly from the initial state to the final state, which can be performed by a reversible loop (Lutz, 1986). Since we do not have reversible loop as a primitive, manual reversible programming is required. In functional programming, loops are naturally encoded as tail recursions, which are known to be difficult to handle in the contexts of program inversion (Glück & Kawabe, 2004; Mogensen, 2006; Matsuda et al., 2010; Nishida & Vidal, 2011). Roughly speaking, for a tail recursion (such as $x = \text{case} \ x \ \text{of} \ (p \rightarrow g \ e; p' \rightarrow e')$, with-conditions are hardly effective in choosing branches, as due to the tail call of $g$, the set of possible results of a branch coincides with the other’s. So we need to program such loop-like computation without tail recursions.

The higher-orderness of SPARCL (and $\lambda_{\Pi_1}^*$) is useful here, as the effort can be made once for all. Specifically, we prepare the following higher-order function implementing general loops.

$$\text{trace} : (a^* \rightarrow (a \otimes a)^*) \rightarrow (b^* \rightarrow (b \otimes b)^*) \rightarrow ((a \otimes x)^* \rightarrow (b \otimes x)^*) \rightarrow a^* \rightarrow b^*$$

$$\text{trace} \ \text{dupA} \ \text{dupB} \ h \ a = \text{let} \ (a_1, a_2)^* = \text{dupA} \ a \ \text{in}$$

$$\ \ \ \text{let} \ (b_1, n)^* = g (h (\text{lnL}^* \ a_1)) \ \text{in}$$

$$\ \ \ \text{let} \ (\text{lnL}^* b_2)^* = h (g oN \ a_2 \ n) \ \text{in}$$

$$\ \ \ \text{invert} \ \text{dupB} \ (b_1, b_2)^*$$

where $g o : (b \otimes x)^* \rightarrow (b \otimes \text{Nat})^*$

$$\ \ \ \text{go} (\text{lnL}^* b)^* = (b, Z^*) \ \text{with} \ \text{isZ} \circ \text{snd}$$

$$\ \ \ \text{go} (\text{lnR}^* x)^* = \text{let} \ (h, n)^* = g (h (\text{lnR}^* x)) \ \text{in} \ (b, S^* n)^*$$

$$\ \ \ \text{go} N : a^* \rightarrow \text{Nat}^* \rightarrow (a \otimes x)^*$$
\[\text{goN } a \text{ Z}^* = \text{lnL}^* \text{ a with isLnL} \]
\[\text{goN } a \text{ (S n)}^* = \text{let } (\text{lnR } x)^* = h (\text{goN } a \text{ n}) \text{ in } \text{lnR}^* \text{ x} \]

The \textit{trace} \textit{dupA} \textit{dupB} \textit{h} \textit{a} applies the forward/backward computation of \textit{h} repeatedly to \textit{lnL} \textit{a}; it returns \textit{b} if \textit{h} returns \textit{lnL} \textit{b}, and otherwise (if \textit{h} returns \textit{lnR} \textit{x}) it applies the same computation again for \textit{h} (\textit{lnR} \textit{x}). Here, \textit{dupA} and \textit{dupB} are supposed to be the reversible duplication (Glück and Kawabe, 2003). This implementation essentially uses Yokoyama \textit{et al.} (2012)'s optimized version of Bennett (1973)'s encoding. That is, if we have an injective \textit{f} : A \rightarrow B of which invertibility is made evident (i.e., locally reversible) by outputting and consuming the \textit{same} trace (or, history (Bennett, 1973)) of type \textit{H} as \textit{f}_1 : A \rightarrow B \otimes H and \textit{f}_2 : A \otimes H \rightarrow B, respectively, then we can implement the version \textit{f}' : A \rightarrow B of which invertibility is evident by (1) copying the input \textit{a} as \textit{(a}_1, \textit{a}_2), (2) applying \textit{f}_1 to \textit{a}_1 to obtain \textit{(b}_1, \textit{h}), (3) applying \textit{f}_2 to \textit{a}_2 and \textit{h} to obtain \textit{b}_2, and (4) applying the inverse of copying (i.e., equivalence check (Glück and Kawabe, 2003)) to \textit{(b}_1, \textit{b}_2) to obtain \textit{b} (= \textit{b}_1 = \textit{b}_2). Note that the roles of \textit{f}_1 and \textit{f}_2 are swapped in the backward execution. Above, we use loop counts as the trace \textit{H}, and \textit{go} and \textit{goN} correspond to \textit{f}_1 and \textit{f}_2, respectively. The construction implies that the inverse of copying must always succeeds, and thus we can safely replace \textit{dupA} by unsafe copying \textit{λa} \textit{pin a unsafeNew} and \textit{dupB} by \textit{invert} (\textit{λb} \textit{pin b unsafeNew}).

The version presented in the main body of this paper assumes this optimization.

By using \textit{trace}, we conclude the proof by \textit{rtm} below that implements the behavior of the given reversible Turing machine.

\[
\text{rtm} : \text{Tape}^* \rightarrow \text{Tape}^*
\]
\[
\text{rtm} = \text{trace} \text{ dupTape dupTape} (\text{checkFinal} \circ \text{step} \circ \text{assertInit})
\]

\[
\text{assertInit} : (\text{Tape} \oplus (\text{T}_Q \otimes \text{Tape}))^* \rightarrow (\text{T}_Q \otimes \text{Tape})^*
\]
\[
\text{assertInit} (\text{lnL } t)^* = (\text{Q}_{q_0}^*, t)^* \text{ with isQ}_{q_0} \circ \text{fst}
\]
\[
\text{assertInit} (\text{lnR } (q, t))^* = (q, t)^*
\]

\[
\text{checkFinal} : (\text{T}_Q \otimes \text{Tape})^* \rightarrow (\text{Tape} \oplus (\text{T}_Q \otimes \text{Tape}))^*
\]
\[
\text{checkFinal} (\text{Q}_{q_f}, t)^* = \text{lnL}^* t \text{ with isL}
\]
\[
\text{checkFinal} (q, t)^* = \text{lnR}^* (q, t)^*
\]

Here, \textit{dupTape} : \text{Tape}^* \rightarrow (\text{Tape} \otimes \text{Tape})^* is the reversible duplication of tapes. Recall that \textit{q}_0 cannot be the destination of a transition and \textit{q}_f cannot be the source. Note that, thanks to \textit{trace}, the above definition of \textit{rtm} is more straightforward than Yokoyama \textit{et al.} (2008) in which \textit{rtm} is defined by forward and backward simulations of a reversible Turing machine with step counting.