JFP, pages, 2022.
doi:10.1017/xxxxx

SPARCL: *A Language for Partially-Invertible Computation*

KAZUTAKA MATSUDA
Tohoku University,6-3-09 Aramaki, Aza-Aoba, Aoba-ku, Sendai, Japan
(e-mail: kztk@ecei.tohoku.ac.jp)

MENG WANG
University of Bristol, BS8 1TH, Bristol, United Kingdom
(e-mail: meng.wang@bristol.ac.uk)

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.

Key Words: reversible computation, linear types

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 forwards and backwards 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 and Vytiniotis 2012), parser and printer (Rendel and Ostermann 2010; Matsuda and Wang 2013, 2018), 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 (Yokoyama et al. 2008; Lutz 1986), Frank's R (Frank 1997), Ψ-Lisp (Baker 1992), RFun (Yokoyama et al. 2011), Π/Π^o (James and Sabry 2012) and Inv (Mu et al. 2004). 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 $\mathit{subs}\ [1,2,5,2,3] = [1,1,3,-3,1]$.

$$
\begin{array}{l}
\mathit{subs} :: [\mathsf{Int}] \to [\mathsf{Int}] \\
\mathit{subs}\ \mathit{xs} = \mathit{goSubs}\ 0\ \mathit{xs} \\[4pt]
\mathit{goSubs} :: \mathsf{Int} \to [\mathsf{Int}] \to [\mathsf{Int}] \\
\mathit{goSubs}\ \_\ [\,] \qquad\quad = [\,] \\
\mathit{goSubs}\ n\ (x : \mathit{xs}) = (x - n) : \mathit{goSubs}\ x\ \mathit{xs}
\end{array}
$$

Despite being simple, these kind of transformations are nevertheless useful. For example, a function similar to *subs* can be found in the pre-processing step of image compression algorithms such as those used for PNG.[1] Another example is the encoding of bags (multisets) of integers, where *subs* can be used to convert sorted lists to lists of integers without any constraints (Kennedy and Vytiniotis 2012).

The function *subs* is invertible. We can define its inverse as below.

$$
\begin{array}{l}
\mathit{subs}^{-1} :: [\mathsf{Int}] \to [\mathsf{Int}] \\
\mathit{subs}^{-1}\ \mathit{ys} = \mathit{goSubs}'\ 0\ \mathit{ys} \\[4pt]
\mathit{goSubs}' :: \mathsf{Int} \to [\mathsf{Int}] \to [\mathsf{Int}] \\
\mathit{goSubs}'\ \_\ [\,] \qquad\quad = [\,] \\
\mathit{goSubs}'\ n\ (y : \mathit{ys}) = \mathbf{let}\ x = y + n\ \mathbf{in}\ x : \mathit{goSubs}'\ x\ \mathit{ys}
\end{array}
$$

However, *subs* cannot be expressed directly in existing reversible programming languages. The problem is that, though *subs* is perfectly invertible, its sub-component *goSubs* is not

[1] https://www.w3.org/TR/2003/REC-PNG-20031110/

(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 defintion, 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 and 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 and Vidal 2006; Kirkeby and Glück 2019, 2020), success depends on the (heuristic) processing order of the expressions.

In this paper, we propose a novel programming language SPARCL[2] 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 and Wang (2018), our type system contains a special type constructor $(-)^\bullet$ (pronounced as "invertible"), where $A^\bullet$ represents A-typed values that are subject to invertible computation. However, having invertible types like $A^\bullet$ 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 and Jones 1991) is not enough (Almendros-Jiménez and Vidal 2006).) This interaction demands conversion from invertible values of type $A^\bullet$ 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 and 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; Yokoyama et al. 2011; Matsuda and Wang 2013; James and Sabry 2012; Mu et al. 2004) commonly assume a form of linearity or relevance, and in SPARCL this assumption is made explicit by a linear type system based on $\lambda^q_\to$ (the core system of Linear Haskell (Bernardy et al. 2018)).

As a teaser, an invertible version of *subs* in SPARCL is show in Fig. 1.[3] In SPARCL, invertible functions from A to B are represented as functions of type $A^\bullet \multimap B^\bullet$, where $\multimap$ 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.

$$
\begin{array}{l}
\mathit{subs} : (\mathsf{List\ Int})^\bullet \multimap (\mathsf{List\ Int})^\bullet \\
\mathit{subs}\ \mathit{xs} = \mathit{goSubs}\ 0\ \mathit{xs} \\
\mathit{goSubs} : \mathsf{Int} \rightarrow (\mathsf{List\ Int})^\bullet \multimap (\mathsf{List\ Int})^\bullet \\
\mathit{goSubs}\ \_\ \mathsf{Nil}^\bullet \qquad\qquad = \mathsf{Nil}^\bullet\ \mathbf{with}\ \mathit{null} \\
\mathit{goSubs}\ n\ (\mathsf{Cons}\ x\ \mathit{xs})^\bullet = \\
\quad \mathbf{let}\ (x, r)^\bullet = \mathbf{pin}\ x\ (\lambda x'.\mathit{goSubs}\ x'\ \mathit{xs})\ \mathbf{in} \quad \text{-- } x' : \mathsf{Int} \text{ is a static version of } x : \mathsf{Int}^\bullet. \\
\quad \mathsf{Cons}^\bullet\ (\mathit{sub}\ n\ x)\ r\ \mathbf{with}\ \mathit{not} \circ \mathit{null} \\
\mathit{sub} : \mathsf{Int} \rightarrow \mathsf{Int}^\bullet \multimap \mathsf{Int}^\bullet \\
\mathit{sub}\ n = \mathbf{lift}\ (\lambda x.x - n)\ (\lambda x.x + n)
\end{array}
$$

Fig. 1: Invertible *subs* in SPARCL

the constructor for linear functions. Partial invertibility is conveniently expressed by taking additional parameters as in $\mathsf{Int} \rightarrow \mathsf{Int}^\bullet \multimap \mathsf{Int}^\bullet$ and $\mathsf{Int} \rightarrow (\mathsf{List\ Int})^\bullet \multimap (\mathsf{List\ Int})^\bullet$. The **pin** : $A^\bullet \multimap (A \rightarrow B^\bullet) \multimap (A \otimes B)^\bullet$ 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 **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 *subs* is defined in a conventional language: *goSubs* has the same recursive pattern with two cases for empty and non-empty lists. This close resemblance to the conventional programming style is what we strive for in the design of SPARCL.

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

$$
\begin{array}{l}
>\ \mathbf{fwd}\ \mathit{subs}\ [1, 2, 5, 2, 3] \\
[1, 1, 3, -3, 1] \\
>\ \mathbf{bwd}\ \mathit{subs}\ [1, 1, 3, -3, 1] \\
[1, 2, 5, 2, 3]
\end{array}
$$

This guarantee of bijectivity is clearly different from the case of (functional) logic programming languages such as Prolog and Curry. Those languages relies 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 (Yokoyama et al. 2008; Lutz 1986; Frank 1997; Baker 1992; Yokoyama et al. 2011; James and Sabry 2012; Mu et al.

2004) including SPARCL, the forward and backward executions of a program take the same steps.

One might notice from the type of **pin** that SPARCL is a higher-order language, in the sense that it contains the simply-typed λ-calculus (more precisely, the simple multiplicity fragment of $\lambda^q_\rightarrow$ (Bernardy et al. 2018)) as a subsystem. Thus, we can, for example, write an invertible map function in SPARCL as below.

$$
\begin{array}{lll}
\mathit{mapR} : (a^\bullet \multimap b^\bullet) \to (\mathsf{List}\ a)^\bullet \to (\mathsf{List}\ b)^\bullet & & \\
\mathit{mapR}\ f\ \mathsf{Nil}^\bullet & = \mathsf{Nil}^\bullet & \textbf{with}\ \mathit{null} \\
\mathit{mapR}\ f\ (\mathsf{Cons}\ x\ \mathit{xs})^\bullet & = \mathsf{Cons}^\bullet\ (f\ x)\ (\mathit{mapR}\ f\ \mathit{xs}) & \textbf{with}\ \mathit{not} \circ \mathit{null}
\end{array}
$$

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 $\multimap$ 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 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 and Vidal 2006; Jacobsen et al. 2018; Mogensen 2008) while with restrictions (specifically, no closures), and the advantage is inherited from Matsuda and Wang (2018) from which SPARCL is inspired, we claim that SPARCL is the first invertible programming language that achieved a proper design for higher-order programming.

In summary, our main contributions are:

- We design 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 $\lambda^{\mathrm{PI}}_\rightarrow$ (Section 3).
- We state and prove several properties about $\lambda^{\mathrm{PI}}_\rightarrow$ (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[4] (Section 4).
- We demonstrate the utility of SPARCL with nontrivial examples: tree rebuilding from inorder and preoder traversals (Mu and Bird 2003), and simplified versions of compression algorithms including Huffman coding, arithmetic coding and LZ77 (Ziv and Lempel 1977) (Section 5).

In addition, a prototype implementation of SPARCL is available from https://github.com/kztk-m/sparcl, which also contains more examples. All the artifacts are linked from the SPARCL web page (https://bx-lang.github.io/EXHIBIT/sparcl.html).

[4] Available from https://github.com/kztk-m/sparcl-agda

A preliminary version of this paper appeared in ICFP20 (Matsuda and 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 and 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; Yokoyama et al. 2011; Matsuda and Wang 2013; James and Sabry 2012; Mu et al. 2004) to exclude non injective 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 $\lambda^q_\to$ (the core system of Linear Haskell (Bernardy et al. 2018)). A feature of the type system is its function type $A \to_p B$, where the arrow is annotated by the argument's multiplicity (1 or ω). Here, $A \to_1 B$ denotes *linear functions* that use the input *exactly once*, while $A \to_\omega B$ denotes *unrestricted functions* that have no restriction on the use of its input. The following are some examples of linear and unrestricted functions.

$$\begin{array}{lll} id : a \to_1 a & \quad double : \mathsf{Int} \to_\omega \mathsf{Int} & \quad const : a \to_1 b \to_\omega a \\ id\ x = x & \quad double\ x = x + x & \quad const\ x\ y = x \end{array}$$

Observe that the *double* used x twice and *const* discards y; hence, the corresponding arrows must be annotated by ω. 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 $A \to_\omega B$ as $!A \multimap B$.

A small deviation from the (simply-typed fragment of) $\lambda^q_\to$ 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.

$$\begin{array}{lll} id : a \to_p a & \quad const : a \to_p b \to a & \quad app : (p \le q) \Rightarrow (a \to_p b) \to_r a \to_q b \\ id\ x = x & \quad const\ x\ y = x & \quad app\ f\ x = f\ x \end{array}$$

In first two examples, p is arbitrary (i.e., 1 or ω); in the last example, the predicate $p \le q$ states an ordering of multiplicity, where $1 \le \omega$.[5] 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

[5] 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 in GHC 9.0.X[6] which checks linearity only when type signatures are given explicitly, SPARCL can infer linear types thanks to the use of qualified typing.

For simplicity, we sometimes write $\multimap$ for $\to_1$ and simply $\to$ for $\to_\omega$ when showing programming examples in SPARCL.

2.2 *Multiplication*

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

$$\mathbf{data}\ \mathsf{Nat} = \mathsf{Z} \mid \mathsf{S}\ \mathsf{Nat}$$

In SPARCL, constructors have linear types: $\mathsf{Z} : \mathsf{Nat}$ and $\mathsf{S} : \mathsf{Nat} \multimap \mathsf{Nat}$.

We define multiplication in term of addition, which is also partially-invertible.[7]

$$\begin{aligned}
&add : \mathsf{Nat} \to \mathsf{Nat}^\bullet \multimap \mathsf{Nat}^\bullet \\
&add\ \mathsf{Z}\ \ \ \ \ \ \ y = y \\
&add\ (\mathsf{S}\ x)\ y = \mathsf{S}^\bullet\ (add\ x\ y)
\end{aligned}$$

As mentioned in the introduction, we use the type constructor $(-)^\bullet$ to distinguish data that are subject to invertible computation (such as $\mathsf{Nat}^\bullet$) and those that are not (such as Nat): when the latter is fixed, a partially invertible function is turned into a (not-necessarily-total) bijection, for example, $add\ n : \mathsf{Nat}^\bullet \multimap \mathsf{Nat}^\bullet$. (For those who read the paper with colors, the arguments of $(-)^\bullet$ are highlighted in dark red.) Values of $(-)^\bullet$-types are constructed by *lifted constructors* such as $\mathsf{S}^\bullet : \mathsf{Nat}^\bullet \multimap \mathsf{Nat}^\bullet$. In the forward direction, $\mathsf{S}^\bullet$ 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 $\mathsf{C} : \sigma_1 \multimap \ldots \multimap \sigma_n \multimap \tau$ automatically give rise to a corresponding lifted version $\mathsf{C}^\bullet : {\sigma_1}^\bullet \multimap \ldots \multimap {\sigma_n}^\bullet \multimap \tau^\bullet$.

A partially invertible multiplication function can be defined by using *add* as below.[8]

$$\begin{aligned}
&mul : \mathsf{Nat} \to \mathsf{Nat}^\bullet \multimap \mathsf{Nat}^\bullet \\
&mul\ z\ \mathsf{Z}^\bullet \ \ \ \ \ \ \ = \mathsf{Z}^\bullet \mathbf{with}\ isZ \\
&mul\ z\ (\mathsf{S}\ x)^\bullet = add\ z\ (mul\ z\ x)\ \mathbf{with}\ not \circ isZ
\end{aligned}$$

An interesting feature in the *mul* program is the *invertible pattern matching* (Yokoyama et al. 2008) indicated by patterns $\mathsf{Z}^\bullet$ and $(\mathsf{S}\ x)^\bullet$ (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 **with** clause to determine the branch to be taken. For example, $mul\ n : \mathsf{Nat}^\bullet \multimap \mathsf{Nat}^\bullet$, in the forward direction values are matched against the forms Z and $\mathsf{S}\ x$; in the backward direction, the **with** conditions are checked upon an output of the

[6] 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)

[7] This type is an instance of the most general type $\mathsf{Nat} \to_p \mathsf{Nat}^\bullet \to_q \mathsf{Nat}^\bullet$ of *add*; recall that there is no problem in using unrestricted inputs only once. We want to avoid overly polymorphic functions for simplicity of presentation.

[8] Nishida et al. (2005) discusses a slightly more complicated but efficient version.

function *mul n*: if $isZ : \mathsf{Nat} \to \mathsf{Bool}$ returns True, the first branch is chosen, otherwise the second branch is chosen. When the second branch is taken, the backward computation of *add n* is performed, which essentially subtracts n, followed by recursively applying the backward computation of *mul n* to the result. As the last step, the final result is enclosed with S and returned. In other words, the backward behavior of *mul n* recursively tries to subtract n, and returns the count of successful subtractions.

In SPARCL, **with** conditions are provided by programmers and expected to be exclusive; the conditions are enforced at run-time: the **with** conditions are asserted to be postconditions on the branches' values. Specifically, the branch's **with** condition is a positive assertion while all the other branches' ones are negative assertions. Thus, the forward computation fails when the branch's **with** condition is not satisfied, or any of the other **with** conditions is also satisfied. This exclusiveness enables the backward computation to uniquely identify the branch (Yokoyama et al. 2008; Lutz 1986). Sometimes we may omit the **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 *goSubs* the second branch's **with** condition is $not \circ null$. One could use sophisticated types such as refinement types to infer **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 *mul* Z defines, as zero times n is zero for any n. In fact, it defines a non-total bijection that in the forward direction the domain of the function contains only Z, i.e., the trivial bijection between $\{\mathsf{Z}\}$ and $\{\mathsf{Z}\}$.

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 $\multimap$ is not equivalent to not-necessarily-total bijections. For example, the function $\lambda x.x\ (\lambda y.y) : ((\sigma \multimap \sigma) \multimap (\sigma \multimap \sigma)) \multimap \sigma \multimap \sigma$ returns $\lambda y.y$ for both $\lambda x.x\ (\lambda z.z)$ and $\lambda x.x$. 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 ($\otimes$) and linear functions ($\multimap$) (even with exponentials (!)) can be modelled in the Int-construction (Joyal et al. 1996) of the category of not-necessarily-total bijections (Abramsky 2005; Abramsky et al. 2002). 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 $(-)^\bullet$ types, function *add* can have type $\mathsf{Nat} \multimap \mathsf{Nat} \multimap \mathsf{Nat}$ (note the linear use of the first argument), which does not specify which parameter is a fixed one. It even has type $\mathsf{Nat} \otimes \mathsf{Nat} \multimap \mathsf{Nat}$ after uncurrying though addition is clearly not fully invertible. These are the reasons why we separate the invertible world and the unidirectional world by using $(-)^\bullet$, inspired by staged languages (Moggi 1998; Davies and Pfenning 2001; Nielson and Nielson 1992). Readers familiar with staged languages may see $A^\bullet$ as residual *code* of type A, which will be executed forwards or backwards 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 (Yokoyama et al. 2011; Mu et al. 2004).

Another design choice we made is to admit types like $(A \multimap B)^\bullet$ 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: **fwd** : $(A^\bullet \multimap B^\bullet) \to A \to B$ and **bwd** : $(A^\bullet \multimap B^\bullet) \to B \to A$. For example, we have:

> **fwd** (*add* (S Z)) (S Z)	-- $(1+)\ 1$
S (S Z)	-- $= 2$
> **bwd** (*add* (S Z)) (S (S Z))	-- $(1+)^{-1}\ 2$
S Z	-- $= 1$
> **fwd** (*mul* (S (S Z))) (S (S (S Z)))	-- $(2\times)\ 3$
S (S (S (S (S (S Z)))))	-- $= 6$
> **bwd** (*mul* (S (S Z))) (S (S (S (S (S (S Z))))))	-- $(2\times)^{-1}\ 6$
S (S (S Z))	-- $= 3$

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

> **bwd** (*mul* (S (S Z))) (S (S (S Z))) -- $(2\times)^{-1}\ 3$
`Runtime Error:...`

The guarantee SPARCL offers is that derived bijections are total with respect to the functions' actual domains and ranges; i.e., **fwd** $e\ v$ results in u, then **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 **fwd/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 $A^\bullet \multimap B^\bullet$ passed to **fwd/bwd** cannot use any resources other than *one* value of type $A^\bullet$ to produce *one* value of type $B^\bullet$. In other words, all and only information from $A^\bullet$ is retained in $B^\bullet$, guaranteeing bijectivity. As a result, SPARCL's type system effectively rejects code like $\mathbf{bwd}\ (\lambda x.\mathsf{Z}^\bullet)$ and $\mathbf{bwd}\ \big(\lambda x.\mathbf{if}\ \mathbf{fwd}\ (\lambda ()^\bullet.x)\ ()\ \mathbf{then}\ \mathsf{Z}^\bullet\ \mathbf{else}\ \mathsf{Z}^\bullet\big)$ as x's multiplicity is ω in both cases. In the former case, x is discarded and multiplicity in our system is either 1 or ω. In the latter case, x appears in the first argument of **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 SPARCL, the language provides a mechanism to directly import existing bijections (as a pair of functions) to construct valid SPARCL programs: $\mathbf{lift} : (A \to B) \to (B \to A) \to A^\bullet \multimap B^\bullet$ converts a pair of functions into a function on $(-)^\bullet$-typed values, expecting that the pair of functions form mutual inverses. For example, by **lift**, we can define *addInt* as below

$$
\begin{aligned}
&\mathit{addInt} : \mathsf{Int} \to \mathsf{Int}^\bullet \multimap \mathsf{Int}^\bullet\\
&\mathit{addInt}\ n = \mathbf{lift}\ (\lambda x.x + n)\ (\lambda x.x - n)
\end{aligned}
$$

The use of **lift** allows one to create primitive bijections to be composed by the various constructs in SPARCL. Another interesting use of **lift** is to implement in-language inversion.

$$
\begin{aligned}
&\mathit{invert} : (A^\bullet \multimap B^\bullet) \to (B^\bullet \multimap A^\bullet)\\
&\mathit{invert}\ h = \mathbf{lift}\ (\mathbf{bwd}\ h)\ (\mathbf{fwd}\ h)
\end{aligned}
$$

2.6 *Composing Partially-Invertible Functions*

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

Let us revisit the example in Section 1. The partially-invertible version of *goSubs* can be implemented via **pin** as below.

$$
\begin{aligned}
&\mathit{subs} : (\mathsf{List\ Int})^\bullet \multimap (\mathsf{List\ Int})^\bullet\\
&\mathit{subs}\ \mathit{xs} = \mathit{goSubs}\ 0\ \mathit{xs}\\
&\mathit{goSubs} : \mathsf{Int} \to (\mathsf{List\ Int})^\bullet \multimap (\mathsf{List\ Int})^\bullet\\
&\mathit{goSubs}\ \_\ \mathsf{Nil}^\bullet \qquad\quad = \mathsf{Nil}^\bullet\ \mathbf{with}\ \mathit{null}\\
&\mathit{goSubs}\ n\ (\mathsf{Cons}\ x\ \mathit{xs})^\bullet =\\
&\quad \mathbf{let}\ (x, r)^\bullet = \mathbf{pin}\ x\ (\lambda x'.\mathit{goSub}\ x'\ \mathit{xs})\ \mathbf{in}\\
&\quad \mathsf{Cons}^\bullet\ (\mathit{sub}\ n\ x)\ r\ \mathbf{with}\ \mathit{not} \circ \mathit{null}\\
&\\
&\mathit{sub} : \mathsf{Int} \to \mathsf{Int}^\bullet \multimap \mathsf{Int}^\bullet\\
&\mathit{sub}\ n = \mathbf{lift}\ (\lambda x.x - n)\ (\lambda x.x + n)
\end{aligned}
$$

(a) partially-invertible version

$$
\begin{aligned}
&\mathit{subsF} : (\mathsf{List\ Int})^\bullet \multimap (\mathsf{List\ Int})^\bullet\\
&\mathit{subsF}\ \mathit{xs} = \mathbf{let}\ (0, r)^\bullet = \mathit{goSubsF}\ 0^\bullet\ \mathit{xs}\ \mathbf{in}\ r\\
&\mathit{goSubsF} : \mathsf{Int}^\bullet \multimap (\mathsf{List\ Int})^\bullet \multimap (\mathsf{Int} \otimes \mathsf{List\ Int})^\bullet\\
&\mathit{goSubsF}\ n\ \mathit{Nil}^\bullet \qquad\quad = (n, \mathsf{Nil}^\bullet)^\bullet\ \mathbf{with}\ \mathit{null} \circ \mathit{snd}\\
&\mathit{goSubsF}\ n\ (\mathsf{Cons}\ x\ \mathit{xs})^\bullet =\\
&\quad \mathbf{let}\ (x, r)^\bullet = \mathit{goSubsF}\ x\ \mathit{xs}\ \mathbf{in}\\
&\quad \mathbf{let}\ (n, x')^\bullet = \mathit{subF}\ (n, x)^\bullet\ \mathbf{in}\\
&\quad (n, \mathsf{Cons}^\bullet\ x'\ r)^\bullet\ \mathbf{with}\ \mathit{not} \circ \mathit{null} \circ \mathit{snd}\\
&\mathit{subF} : (\mathsf{Int} \otimes \mathsf{Int})^\bullet \multimap \mathsf{Int} \otimes \mathsf{Int}^\bullet\\
&\mathit{subF} = \mathbf{lift}\ (\lambda(n, x).(n, x - n))\ (\lambda(n, x).(n, x + n))
\end{aligned}
$$

(b) fully-invertible version

Fig. 2: Side-by-side comparison of partially-invertible (a) and fully-invertible (b) versions of *subs*

$$
\begin{aligned}
&\mathit{goSubs} : \mathsf{Int} \to (\mathsf{List\ Int})^\bullet \multimap (\mathsf{List\ Int})^\bullet\\
&\mathit{goSubs}\ \_\ \mathsf{Nil}^\bullet \qquad\quad = \mathsf{Nil}^\bullet\ \mathbf{with}\ \mathit{null}\\
&\mathit{goSubs}\ n\ (\mathsf{Cons}\ x\ \mathit{xs})^\bullet = (\mathbf{case\ pin}\ x\ (\lambda x'.\mathit{goSubs}\ x' \mathit{xs})\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad (x, r)^\bullet \to \mathsf{Cons}^\bullet\ (\mathit{sub}\ n\ x)\ r\ \mathbf{with}\ \lambda\_.\mathsf{True})\ \mathbf{with}\ \mathit{not} \circ \mathit{null}
\end{aligned}
$$

Here, we used **pin** to convert $x : \mathsf{Int}^\bullet$ to $x' : \mathsf{Int}$ in order to pass it to the recursive call of *goSubs*. In the backward direction, *goSubs n* executes as follows.[9]

$$
\begin{aligned}
&\quad \mathbf{bwd}\ (\mathit{goSubs}\ 0)\ [1, 1, 3, -3, 1]\\
&= \{\ \mathsf{Cons\ branch\ is\ taken;\ Cons}\ (\mathit{sub}\ 0\ x)\ r = [1, 1, 3, -3, 1] \Longrightarrow x = 1, r = [1, 3, -3, 1].\ \}\\
&\quad \mathsf{Cons}\ 1\ (\mathbf{bwd}\ (\mathit{goSubs}\ 1)\ [1, 3, -3, 1])\\
&= \{\ \mathsf{Cons\ branch\ is\ taken;\ Cons}\ (\mathit{sub}\ 1\ x)\ r = [1, 3, -3, 1] \Longrightarrow x = 2, r = [3, -3, 1].\ \}\\
&\quad \mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ (\mathbf{bwd}\ (\mathit{goSubs}\ 2)\ [3, -3, 1]))\\
&= \{\ \mathsf{Cons\ branch\ is\ taken;\ Cons}\ (\mathit{sub}\ 2\ x)\ r = [3, -3, 1] \Longrightarrow x = 5, r = [-3, 1].\ \}\\
&\quad \mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 5\ (\mathbf{bwd}\ (\mathit{goSubs}\ 5)\ [-3, 1])))\\
&= \ \ldots\\
&= \mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 5\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 3\ (\mathbf{bwd}\ (\mathit{goSubs}\ 3)\ [])))))\\
&= \{\ \mathsf{Nil\ branch\ is\ taken.}\ \}\\
&\quad \mathsf{Cons}\ 1\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 5\ (\mathsf{Cons}\ 2\ (\mathsf{Cons}\ 3\ \mathsf{Nil})))) = [1, 2, 5, 2, 3]
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{goSubs}'\ \_\ [\,] \qquad\ \ = [\,]\\
&\mathit{goSubs}'\ n\ (y : \mathit{ys}) = \mathbf{let}\ x = y + n\ \mathbf{in}\ x : \mathit{goSubs}'\ x\ \mathit{ys}
\end{aligned}
$$

The use of **pin** commonly results in an invertible **case** with a single branch, as we see in *goSubs* above. We capture this pattern with an invertible **let** as a shorthand notation, which enables us to write $\mathbf{let}\ p^\bullet = e_1\ \mathbf{in}\ e_2$ for $\mathbf{case}\ e_1\ \mathbf{of}\ \{p^\bullet \to e_2\ \mathbf{with}\ \lambda\_.\mathsf{True}\}$. The definition of *goSubs* shown in Section 1 uses this shorthand notation, which is reproduced in Fig. 2a.

[9] This execution trace is (overly) simplified for illustration purpose. See Section 3.5 for the actual operational semantics.

We would like to emphasise that partial invertibility, as supported in SPARCL, is key to concise function definitions. In Fig. 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 *goSubs* and *sub*). As a result, *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, Huffman Coding in Section 5.2).

We end this section with a theoretical remark. One might wonder why $(-)^{\bullet}$ 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, $(-)^{\bullet}$, which brings a type in the second stage into the first stage, forms a functor, but the functor is not endo. Recall that $A^{\bullet}$ represents residual code in an invertible system of type A; that is, $A^{\bullet}$ and its component A belong to different categories (though we have not formally described them).[10] One then might wonder whether $(-)^{\bullet}$ 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) $(-)^{\bullet}$. It is unclear whether there exists such a functor other than $(-)^{\bullet}$ 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 non-linear constructors, such as $\mathsf{MkUn} : a \to \mathsf{Un}\ a$ 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 **with** conditions are examined from top to bottom. Recall also that 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`.

[10] For curious readers, we note our conjecture that $(-)^{\bullet}$ 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.

3 Core System: $\lambda^{\mathrm{PI}}_{\rightarrow}$

This section introduces $\lambda^{\mathrm{PI}}_{\rightarrow}$, 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) $\lambda^{q}_{\rightarrow}$ (Bernardy et al. 2018), and, as mentioned in Section 2.3, it is also two-staged (Moggi 1998; Nielson and Nielson 1992) 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 $\overline{t}$ denotes a sequence such as $t_1, \dots, t_n$ or $t_1; \dots; t_n$, 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 $|\overline{t}|$. In addition, we may refer to an element in the sequence $\overline{t}$ as t_i. A simultaneous substitution of $x_1, \dots, x_n$ in t with $s_1, \dots, s_n$ is denoted as $t[s_1/x_1, \dots, s_n/x_n]$, which may also be written as $t[\overline{s/x}]$.

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 $\lambda^{\mathrm{PI}}_{\rightarrow}$ where we have a primitive *bijection type* $A \rightleftharpoons B$, which is inhabited by bijections constructed from functions of type $A^{\bullet} \multimap B^{\bullet}$. Technically, having a dedicated bijection type facilitates reasoning. For example, we may now straightforwardly state that "values of a bijection type $A \rightleftharpoons B$ are bijections between A and B" (Corollary 3.4).

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

Here, ω of λ_{ω} indicates that the bound variable can be used arbitrary many. In contrast, λ_1 indicates that the bound variable must be used linearly. Hence, for example, $\lambda_1 x.\mathsf{Z}$ and $\lambda_1 x.(x, x)$ are ill-typed, while $\lambda_1 x.x$, $\lambda_{\omega} x.\mathsf{Z}$ and $\lambda_{\omega} x.(x, x)$ are well-typed. Similarly, we also annotate (unidirectional) **case**s with the multiplicity of the variables bound by pattern matching. Thus, for example, $\mathbf{case}_1\ \mathsf{S}\ \mathsf{Z}\ \mathbf{of}\ \{\mathsf{S}\ x \rightarrow (x, x)\}$ and $\lambda_1 x.\mathbf{case}_{\omega}\ x\ \mathbf{of}\{\mathsf{S}\ y \rightarrow \mathsf{Z}\}$ are ill-typed.

3.2 Syntax

The syntax of $\lambda^{\mathrm{PI}}_{\rightarrow}$ is given as below.

$$\begin{array}{lrcl}
\text{Expressions:} & e & ::= & x \mid \lambda_{\pi} x.e \mid e_1\ e_2 \mid \mathsf{C}\ \overline{e} \mid \mathbf{case}_{\pi}\ e_0\ \mathbf{of}\ \{\overline{p \rightarrow e}\} \\
& & \mid & \mathsf{C}^{\bullet}\ \overline{e} \mid \mathbf{case}\ e_0\ \mathbf{of}\ \{\overline{p^{\bullet} \rightarrow e\ \mathbf{with}\ e'}\} \\
& & \mid & \mathbf{pin}\ e_1\ e_2 \mid \mathbf{unlift}\ e \mid e_1 \triangleright e_2 \mid e_1 \triangleleft e_2 \\
\text{Patterns:} & p & ::= & \mathsf{C}\ \overline{x} \\
\text{Multiplicities:} & \pi & ::= & 1 \mid \omega
\end{array}$$

There are three lines for the various constructs of expressions. The ones in the first line are standard except the annotations in λ and **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, **unlift** e, $e_1 \triangleright e_2$, and $e_1 \triangleleft e_2$ handles bijections which can be used to encode **fwd** and **bwd** in SPARCL. We have already seen lifted constructors, invertible **case**, and **pin** in Section 2. For simplicity, we assume that **pin**, C and $\mathsf{C}^{\bullet}$ are fully-applied. Lifted constructor expressions $\mathsf{C}^{\bullet}\ \overline{e}$ and invertible **case**s are basic invertible primitives in $\lambda^{\mathrm{PI}}_{\rightarrow}$. 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 non-overlapping both for unidirectional and invertible **case**s. We do not include **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 **unlift** (together with $\triangleright$ and $\triangleleft$) and **pin** support communication between the unidirectional world and the invertible world. The **unlift** construct together with $\triangleright$ and $\triangleleft$ runs invertible computation in the unidirectional world. The **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 $\lambda^{\mathrm{PI}}_{\rightarrow}$ are defined as below.

$$A, B ::= \alpha \mid \mathsf{T}\ \overline{A} \mid A \rightarrow_{\pi} B \mid A^{\bullet} \mid A \rightleftharpoons B$$

Here, α denotes a type variable, T denotes a type constructor, $A \rightarrow_{\pi} B$ is a function type annotated with the argument's multiplicity π, $(-)^{\bullet}$ marks invertibility, and $A \rightleftharpoons B$ is a bijection type.

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

$$\mathsf{C} : A_1 \multimap A_2 \multimap \cdots \multimap A_n \multimap \mathsf{T}\ \overline{\alpha}$$

with $\mathsf{fv}(A_i) \in \{\overline{\alpha}\}$ for any i.[11] Type variables α 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.

$$(-,-) : \alpha_1 \multimap \alpha_2 \multimap \alpha_1 \otimes \alpha_2 \qquad \mathsf{InL} : \alpha_1 \multimap (\alpha_1 \oplus \alpha_2) \qquad \mathsf{InR} : \alpha_2 \multimap (\alpha_1 \oplus \alpha_2)$$

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

$$\mathsf{Nil} : \mathsf{List}\ \alpha \qquad \mathsf{Cons} : \alpha \multimap \mathsf{List}\ \alpha \multimap \mathsf{List}\ \alpha$$

[11] 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 **case**s.

We may write $\overline{A} \multimap B$ for $A_1 \multimap A_2 \multimap \cdots \multimap A_n \multimap B$ (when n is zero, $\overline{A} \multimap B$ is B). We shall also instantiate constructors implicitly and write $\mathsf{C} : \overline{A'} \multimap \mathsf{T}\ \overline{B}$ when there is a constructor $\mathsf{C} : \overline{A} \multimap \mathsf{T}\ \overline{\alpha}$ 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 $\mathsf{MkF} : (\mathsf{F}\ \alpha \to \alpha) \multimap \mathsf{F}\ \alpha$, we have a fixpoint operator as below.

$$\begin{aligned} &\mathit{fix}_\pi \triangleq \lambda_\omega f.\lambda_\pi a.\big(\lambda_\omega x.\lambda_\pi a.f\ (\mathit{out}\ x\ x)\ a\big)\ \big(\mathsf{MkF}\ (\lambda_\omega x.\lambda_\pi a.f\ (\mathit{out}\ x\ x)\ a)\big)\ a \\ &\quad \text{where } \mathit{out} \triangleq \lambda_1 x.\mathbf{case}_1\ x\ \mathbf{of}\ \{\mathsf{MkF}\ t \to t\} \end{aligned}$$

Here, out has type $\mathsf{F}\ C \multimap \mathsf{F}\ C \to C$ for any C (in this case $C = A \to_\pi B$), and thus fix_π has type $((A \to_\pi B) \to (A \to_\pi B)) \to A \to_\pi B$.

The most special type in the language is $A^\bullet$, which is the invertible version of A. More specifically, the invertible type $A^\bullet$ represents residual code in an invertible system that are executed forwards and backwards at the second stage to output and input A-typed values. Values of type $A^\bullet$ must be treated linearly, and can only be manipulated by invertible operations, such as lifted constructors, invertible pattern matching, and **pin**. To keep our type system simple, or more specifically single-kinded, we allow types like $(A \multimap B)^\bullet$ and $(A^\bullet)^\bullet$, while the category of (not-necessarily-total) bijections are not closed and $\lambda^{\mathrm{PI}}_{\rightarrow}$ 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 **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 $(A \to B) \otimes (B \to A)$. 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 *typing environment* is a mapping from variables x to pairs of type A and its multiplicity π, meaning that x has type A and can be used π-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 ε for the empty environment. Reflecting the two stages, we adopt a dual context system (Davies and Pfenning 2001), which has *unidirectional* and *invertible* environments, denoted by Γ and Θ respectively. This separation of the two is purely theoretical, for the purpose of facilitating reasoning when we interpret $A^\bullet$-typed expressions that are closed in unidirectional variables but may have free variables in Θ as bijections. In fact, our prototype implementation does not distinguish the two environments. For all invertible environments Θ, without the loss of generality we assume that the associated multiplicities must be 1, i.e., $\Theta(x) = (A_x, 1)$ for any $x \in \mathsf{dom}(\Theta)$. Thus, we shall sometimes omit multiplicities for Θ. This assumption is actually an invariant in our system since any variables introduced in Θ must have multiplicity 1. We make this explicit in order to simplify the theoretical discussions. Moreover, we assume that the domains of Γ and Θ are disjoint.

Typing Rules for Expressions $\boxed{\Gamma; \Theta \vdash e : A}$ **and Patterns** $\boxed{p : A \rhd_\pi \Gamma}$

$$\frac{}{\omega\Gamma + x :_1 A; \omega\Theta \vdash x : A}\,\text{T-VAR}$$

$$\frac{\Gamma, x :_\pi A; \Theta \vdash e : B}{\Gamma; \Theta \vdash \lambda_\pi x.e : A \to_\pi B}\,\text{T-ABS} \qquad \frac{\Gamma_1; \Theta_1 \vdash e_1 : A \to_\pi B \quad \Gamma_2; \Theta_2 \vdash e_2 : A}{\Gamma_1 + \pi\Gamma_2; \Theta_1 + \pi\Theta_2 \vdash e_1\ e_2 : B}\,\text{T-APP}$$

$$\frac{n = |\overline{e}| = |\overline{A}| \quad \mathsf{C} : \overline{A} \multimap T\ \overline{B} \quad \{\Gamma_i; \Theta_i \vdash e_i : A_i\}_i}{\omega\Gamma_0 + \Gamma_1 + \cdots + \Gamma_n; \Theta_1 + \cdots + \Theta_n \vdash \mathsf{C}\ \overline{e} : T\ \overline{B}}\,\text{T-CON}$$

$$\frac{\Gamma_0; \Theta_0 \vdash e_0 : A \quad \{p_i : A \rhd_\pi \Gamma_i \quad \Gamma, \Gamma_i; \Theta \vdash e_i : B\}_i}{\pi\Gamma_0 + \Gamma; \pi\Theta_0 + \Theta \vdash \mathbf{case}_\pi\ e_0\ \mathbf{of}\ \{\overline{p \to e}\} : B}\,\text{T-CASE}$$

$$\frac{}{\omega\Gamma; x : A \vdash x : A^\bullet}\,\text{T-RVAR} \qquad \frac{n = |\overline{e}| = |\overline{A}| \quad \mathsf{C} : \overline{A} \multimap T\ \overline{B} \quad \{\Gamma_i; \Theta_i \vdash e_i : A_i^\bullet\}_i}{\omega\Gamma_0 + \Gamma_1 + \cdots + \Gamma_n; \Theta_1 + \cdots + \Theta_n \vdash \mathsf{C}^\bullet\ \overline{e} : (T\ \overline{B})^\bullet}\,\text{T-RCON}$$

$$\frac{\Gamma_0; \Theta_0 \vdash e_0 : A^\bullet \quad \{p_i : A \rhd_1 \Theta_i \quad \Gamma; \Theta, \Theta_i \vdash e_i : B^\bullet \quad \Gamma'; \Theta' \vdash e'_i : B \to_\omega \mathsf{Bool}\}_i}{\Gamma_0 + \Gamma + \omega\Gamma'; \Theta_0 + \Theta + \omega\Theta' \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \{\overline{p^\bullet \to e\ \mathbf{with}\ e'}\} : B^\bullet}\,\text{T-RCASE}$$

$$\frac{\Gamma_1; \Theta_1 \vdash e_1 : A^\bullet \quad \Gamma_2; \Theta_2 \vdash e_2 : A \to_\omega B^\bullet}{\Gamma_1 + \Gamma_2; \Theta_1 + \Theta_2 \vdash \mathbf{pin}\ e_1\ e_2 : (A \otimes B)^\bullet}\,\text{T-PIN} \qquad \frac{\Gamma; \Theta \vdash e : A^\bullet \to_1 B^\bullet}{\omega\Gamma; \omega\Theta \vdash \mathbf{unlift}\ e : A \rightleftharpoons B}\,\text{T-UNLIFT}$$

$$\frac{\Gamma_1; \Theta_1 \vdash e_1 : A \rightleftharpoons B \quad \Gamma_2; \Theta_2 \vdash e_2 : A}{\Gamma_1 + \omega\Gamma_2; \Theta_1 + \omega\Theta_2 \vdash e_1 \rhd e_2 : B}\,\text{T-FAPP} \qquad \frac{\Gamma_1; \Theta_1 \vdash e_1 : A \rightleftharpoons B \quad \Gamma_2; \Theta_2 \vdash e_2 : B}{\Gamma_1 + \omega\Gamma_2; \Theta_1 + \omega\Theta_2 \vdash e_1 \lhd e_2 : A}\,\text{T-BAPP}$$

$$\frac{\mathsf{C} : \overline{A} \multimap \mathsf{T}\ \overline{B}}{\mathsf{C}\ \overline{x} : \mathsf{T}\ \overline{B} \rhd_\pi \overline{x :_\pi A}}$$

Fig. 3: Typing rules for expressions and patterns

Given two unidirectional typing environments Γ_1 and Γ_2, we define the addition $\Gamma_1 + \Gamma_2$ as below.

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

If $\mathsf{dom}(\Gamma_1)$ and $\mathsf{dom}(\Gamma_2)$ are disjoint, we sometimes write Γ_1, Γ_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 Θ, $\Theta_1 + \Theta_2 = \Theta$ implicitly implies $\mathsf{dom}(\Theta_1) \cap \mathsf{dom}(\Theta_2) = \emptyset$.

We define multiplication of multiplicities as below.

$$1\pi = \pi 1 = \pi \qquad \omega\pi = \pi\omega = \omega$$

Given $\Gamma = x_1 :_{\pi_1} A_1, \ldots, x_n :_{\pi_n} A_n$, we write $\pi\Gamma$ for the environment $x_1 :_{\pi\pi_1} A_1, \ldots, x_n :_{\pi\pi_n} A_n$. 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 $\Gamma(x) = (\_, \omega)$ for all $x \in \mathsf{dom}(\Gamma)$.

The typing relation $\Gamma; \Theta \vdash e : A$ reads that under environments Γ and Θ, expression e has type A (Fig. 3). The definition basically follows $\lambda^q_\to$ (Bernardy et al. 2018) except having two environments for the two stages. Although multiplicities in Θ are always 1, some of the typing rules refers to $\omega\Theta$ (which implies $\Theta = \varepsilon$) in the conclusion parts, to emphasize that Γ and Θ 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 $\varepsilon; \Theta \vdash e : A^{\bullet}$ defines a piece of code representing a bijection between Θ and A, and hence $\varepsilon; \varepsilon \vdash e' : A \rightleftharpoons B$ 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 Fig. 3 are divided into three groups: the standard ones inherited from $\lambda^{q}_{\rightarrow}$ (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 Γ and Θ 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 π times, the multiplicities of variables in e are multiplied by π. Discarding variables, or weakening, is performed in the rules T-VAR, T-RVAR, T-CON, and T-RCON which can be leaves in a derivation tree. Note that weakening is not allowed for Θ-variables as they are linear.

The typing rules for the invertible part would need additional explanation. In T-RVAR, x has type $A^{\bullet}$ if the invertible typing environment is the singleton mapping $x : A$. One explanation for this is that Θ represents the typing environment for the object (i.e., invertible) system. Another explanation is that we simply omit $(-)^{\bullet}$ as all variables in Θ must have types of the form $A^{\bullet}$. 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 $(-)^{\bullet}$-typed data; the pattern matching is done in the invertible world, and thus the bodies of the branches must also have $(-)^{\bullet}$-types. Recall that the **with**-conditions (e'_i) are used for deciding which branch is used in backward computation. The type $B \rightarrow_{\omega} \mathsf{Bool}$ 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 $(-)^{\bullet}$-typed values, there is little motivation to use linear variables to define conventional functions in $\lambda^{\mathrm{PI}}_{\rightarrow}$. The operators **pin**, **unlift**, $\triangleright$, and $\triangleleft$ are special in $\lambda^{\mathrm{PI}}_{\rightarrow}$. The operator **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 **unlift** and constructed bijections must be closed in terms of invertible variables. It might look a little weird that $e_1 \triangleright e_2 / e_1 \triangleleft e_2$ uses e_1 linearly; this is not a problem because Θ_1 in T-FAPP/T-BAPP must be empty for expressions that occur in evaluation (Lemma 3.2).

3.5 Operational Semantics

The semantics of $\lambda^{\mathrm{PI}}_{\rightarrow}$ consists of three evaluation relations: *unidirectional*, *forward*, and *backward*. The unidirectional evaluation evaluates away the unidirectional constructs such as λ-abstractions and applications, and the forward and backward evaluation specifies bijections.

For example, let us consider an expression $e = (\lambda_{\omega} f. f\ (f\ y))\ (\lambda_1 x. \mathsf{S}^{\bullet}\ x)$. Due to λ-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 $e \Downarrow \mathsf{S}^\bullet\ (\mathsf{S}^\bullet\ y)$ where the residual $\mathsf{S}^\bullet\ (\mathsf{S}^\bullet\ y)$ is ready to be interpreted bijectively. The forward evaluation $\mu \vdash E \Rightarrow v$ evaluates a residual E under an environment μ to obtain a value v as usual. For example, we have $\{y \mapsto \mathsf{Z}\} \vdash \mathsf{S}^\bullet\ (\mathsf{S}^\bullet\ y) \Rightarrow \mathsf{S}\ (\mathsf{S}\ \mathsf{Z})$. The backward evaluation $E \Leftarrow v \dashv \mu$ does the opposite; it inversely evaluates E to find an environment μ for a given value v, so that the corresponding forward evaluation of E returns the value for the environment. For example, we have $\mathsf{S}^\bullet\ (\mathsf{S}^\bullet\ y) \Leftarrow \mathsf{S}\ (\mathsf{S}\ \mathsf{Z}) \dashv \{y \mapsto \mathsf{Z}\}$.

This is the basic story, but computation can be more complicated in general. With **case** and **pin**, the forward $\Rightarrow$ and backward $\Leftarrow$ evaluation depend on the unidirectional evaluation $\Downarrow$; and with $\triangleright$ and $\triangleleft$, 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 and Wang 2018).

3.5.1 Values and Residuals

We first define a set of *values* v and a set of *residuals* E as below.

$$\begin{array}{lrcl} \text{Values:} & v & ::= & \lambda_\pi x.e \mid \mathsf{C}\ \overline{v} \mid E \mid \langle x.E \rangle \\ \text{Residuals:} & E & ::= & x \mid \mathsf{C}^\bullet\ \overline{E} \mid \mathbf{case}\ E_0\ \mathbf{of}\ \{\overline{p^\bullet \to e\ \mathbf{with}\ \lambda_\omega x.e'}\} \mid \mathbf{pin}\ E_1\ (\lambda_\omega x_2.e_2) \end{array}$$

The residuals are $(-)^\bullet$-typed expressions, which are subject to the forward and backward evaluations. The syntax of residuals makes it clear that branch bodies in invertible **case**s 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:

$$\frac{\Gamma; \Theta, x : A \vdash E : B^\bullet}{\omega\Gamma; \omega\Theta \vdash \langle x.E \rangle : A \rightleftharpoons B}\,\textsc{T-Holed}$$

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 $\langle x.E \rangle$ cannot have any free variables.

3.5.2 Three Evaluation Relations: Unidirectional, Forward and Backward

The evaluation relations are shown in Fig. 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 **case**s, **lift**, and **pin**) as constructors. A subtlety is that we assume dynamic α-renaming of invertible **case**s to avoid variable capturing. The evaluation rules can evaluate open expressions by having $x \Downarrow x$; recall that residuals can contain variables. The **unlift** operator uses a fresh variable y in the evaluation to make a

Unidirectional Evaluation $\boxed{e \Downarrow v}$

$$\frac{}{\lambda_\pi x.e \Downarrow \lambda_\pi x.e} \qquad \frac{e_1 \Downarrow \lambda_\pi x.e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v} \qquad \frac{\overline{e \Downarrow v}}{\mathsf{C}\ \overline{e} \Downarrow \mathsf{C}\ \overline{v}} \qquad \frac{e_0 \Downarrow v_0 \quad p_i\mu = v_0 \quad e_i\mu \Downarrow v}{\textbf{case}_\pi\ e_0\ \textbf{of}\ \{\overline{p \to e}\} \Downarrow v}$$

$$\frac{}{x \Downarrow x} \qquad \frac{\overline{e \Downarrow E}}{\mathsf{C}^\bullet\ \overline{e} \Downarrow \mathsf{C}^\bullet\ \overline{E}} \qquad \frac{e_0 \Downarrow E_0 \quad \overline{e' \Downarrow \lambda_\omega x.e''} \quad \alpha\text{-renaming to make } \mathsf{fv}(p) \text{ fresh}}{\textbf{case}\ e_0\ \textbf{of}\ \{\overline{p^\bullet \to e\ \textbf{with}\ e'}\} \Downarrow \textbf{case}\ E_0\ \textbf{of}\ \{\overline{p^\bullet \to e\ \textbf{with}\ \lambda_\omega x.e''}\}}$$

$$\frac{e_1 \Downarrow E_1 \quad e_2 \Downarrow \lambda_\omega x_2.e_2'}{\textbf{pin}\ e_1\ e_2 \Downarrow \textbf{pin}\ E_1\ (\lambda_\omega x_2.e_2')} \qquad \frac{}{\langle x.E\rangle \Downarrow \langle x.E\rangle} \qquad \frac{e \Downarrow \lambda_1 x.e' \quad e'[y/x] \Downarrow E \quad y\text{: fresh}}{\textbf{unlift}\ e \Downarrow \langle y.E\rangle}$$

$$\frac{e_1 \Downarrow \langle x.E\rangle \quad e_2 \Downarrow v_2 \quad \{x \mapsto v_2\} \vdash E \Rightarrow v}{e_1 \rhd e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow \langle x.E\rangle \quad e_2 \Downarrow v_2 \quad E \Leftarrow v_2 \dashv \{x \mapsto v\}}{e_1 \lhd e_2 \Downarrow v}$$

Forward Evaluation $\boxed{\mu \vdash E \Rightarrow v}$

$$\frac{}{\{x \mapsto v\} \vdash x \Rightarrow v} \qquad \frac{\overline{\mu \vdash E \Rightarrow v}}{\biguplus \overline{\mu} \vdash \mathsf{C}^\bullet\ \overline{E} \Rightarrow \mathsf{C}\ \overline{v}} \qquad \frac{\mu_1 \vdash E_1 \Rightarrow v_1 \quad e_2[v_1/x] \Downarrow E_2 \quad \mu_2 \vdash E_2 \Rightarrow v_2}{\mu_1 \uplus \mu_2 \vdash \textbf{pin}\ E_1\ (\lambda_\omega x.e_2) \Rightarrow (v_1, v_2)}$$

$$\frac{\mu_0 \vdash E_0 \Rightarrow p_i\mu_i \quad \mathsf{dom}(\mu_i) = \mathsf{fv}(p_i) \quad e_i \Downarrow E_i \quad \mu \uplus \mu_i \vdash E_i \Rightarrow v \quad e_i'[v/x_i] \Downarrow \mathsf{True} \quad \{e_j'[v/x_j] \Downarrow \mathsf{False}\}_{j \neq i}}{\mu_0 \uplus \mu \vdash \textbf{case}\ E_0\ \textbf{of}\ \{\overline{p^\bullet \to e\ \textbf{with}\ \lambda_\omega x.e'}\} \Rightarrow v}$$

Backward Evaluation $\boxed{E \Leftarrow v \dashv \mu}$

$$\frac{}{x \Leftarrow v \dashv \{x \mapsto v\}} \qquad \frac{\overline{E \Leftarrow v \dashv \mu}}{\mathsf{C}^\bullet\ \overline{E} \Leftarrow \mathsf{C}\ \overline{v} \dashv \biguplus \overline{\mu}} \qquad \frac{E_1 \Leftarrow v_1 \dashv \mu_1 \quad e_2[v_1/x] \Downarrow E_2 \quad E_2 \Leftarrow v_2 \dashv \mu_2}{\textbf{pin}\ E_1\ (\lambda_\omega x.e_2) \Leftarrow (v_1, v_2) \dashv \mu_1 \uplus \mu_2}$$

$$\frac{e_i'[v/x_i] \Downarrow \mathsf{True} \quad \{e_j'[v/x_j] \Downarrow \mathsf{False}\}_{j \neq i} \quad e_i \Downarrow E_i \quad E_i \Leftarrow v \dashv \mu \uplus \mu_i \quad \mathsf{dom}(\mu_i) = \mathsf{fv}(p_i) \quad E_0 \Leftarrow p_i\mu_i \dashv \mu_0}{\textbf{case}\ E_0\ \textbf{of}\ \{\overline{p^\bullet \to e\ \textbf{with}\ \lambda_\omega x.e'}\} \Leftarrow v \dashv \mu_0 \uplus \mu}$$

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

single-holed residual $\langle y.E\rangle$ as a representation of bijection. Such single-holed residuals can be used in the forward direction by $e_1 \rhd e_2$ and in the backward direction by $e_1 \lhd e_2$, by triggering the corresponding evaluation.

The forward evaluation $\mu \vdash E \Rightarrow v$ states that under environment μ, a residual E evaluates to a value v, and the backward evaluation $E \Leftarrow v \dashv \mu$ inversely evaluates E to return the environment μ 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 are 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 $\lambda_{\rightarrow}^{\mathrm{PI}}$.

3.6.1 Subject Reduction

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

Lemma 3.1. $\Gamma, x :_{\pi} A; \Theta \vdash e : B$ and $\Gamma'; \Theta' \vdash e' : A$ 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 $\varepsilon; \Theta \vdash E : A^{\bullet}$ and $\mu \vdash E \Rightarrow v$. Then, $\mathsf{dom}(\Theta) = \mathsf{dom}(\mu)$ holds, and $\varepsilon; \varepsilon \vdash \mu(x) : \Theta(x)$ for all $x \in \mathsf{dom}(\Theta)$ implies $\varepsilon; \varepsilon \vdash v : A$.
- Suppose $\varepsilon; \Theta \vdash E : A^{\bullet}$ and $E \Leftarrow v \dashv \mu$. Then, $\mathsf{dom}(\Theta) = \mathsf{dom}(\mu)$ holds, and $\varepsilon; \varepsilon \vdash v : A$ implies $\varepsilon; \varepsilon \vdash \mu(x) : \Theta(x)$ for all $x \in \mathsf{dom}(\Theta)$.

Proof By (mutual) induction on the derivation steps of evaluation. ■

The statements correspond to the three evaluation relations in $\lambda_{\rightarrow}^{\mathrm{PI}}$. Note that the unidirectional evaluation targets expressions that are closed in terms of unidirectional variables, but may be open in terms of invertible variables, a property that is reflected in the first statement above. The second and third statements are more standard, assuming closed expressions in terms of both unidirectional and invertible variables. This assumption is 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 $A \rightleftharpoons B$ forms a bijection. Values of type $A \rightleftharpoons B$ 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 $\varepsilon; \varepsilon \vdash \langle x.E \rangle : A \rightleftharpoons B$, which implies $\varepsilon; x : A \vdash E : B^{\bullet}$. 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 $E \Leftarrow v' \dashv \{x \mapsto v\}$.[12]

[12] Here, we consider syntactic (definitional) equality of values, but it is rather easy to extend the discussion to observational equivalence.

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 $E \Leftarrow v \dashv \mu$.

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 $\langle x.E \rangle : A \rightleftharpoons B$ actually implements a bijection between A-typed values and B-typed values.

Corollary 3.4 (bijectivity of bijection values)**.** Suppose $\varepsilon; \varepsilon \vdash \langle x.E \rangle : A \rightleftharpoons B$. 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 $E \Leftarrow v' \dashv \{x \mapsto v\}$. ■

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 $\lambda^{\mathrm{PI}}_{\rightarrow}$ 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 (Abel and Chapman 2014; Capretta 2005) 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, 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 $\lambda^{\mathrm{PI}}_{\rightarrow}$ is reversible Turing complete, as it has recursion (via fix_π in Section 3.3) and reversible branching (i.e., invertible **case**).

Theorem 3.5. $\lambda^{\mathrm{PI}}_{\rightarrow}$ is reversible Turing complete. ■

The proof is done by constructing a simulator for a given reversible Turing machine, which is presented in Appendix 1. We follow the construction in Yokoyama et al. (2011) except the last step, in which we use a general reversible looping operator as below.[13]

$$\mathit{trace} : ((a \oplus x)^{\bullet} \multimap (b \oplus x)^{\bullet}) \rightarrow a^{\bullet} \multimap b^{\bullet}$$

As its type suggests, *trace* h applies h to $\mathsf{InL}\ a$ repeatedly until it returns $\mathsf{InL}\ b$; the function loops while h returns a value of the form $\mathsf{InR}\ x$. 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 (Nishida and Vidal 2011; Mogensen 2006; Matsuda et al. 2010; Glück and Kawabe 2004). 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 $\lambda^{\mathrm{PI}}_{\rightarrow}$ (and SPARCL) helps here, as the effort is made once and for all.

3.7 Extension with The lift *Operator*

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

Adding **lift** to $\lambda^{\mathrm{PI}}_{\rightarrow}$ is rather easy. We extend expressions to include **lift** as $e ::= \cdots \mid \mathbf{lift}\ e_1\ e_2\ e_3$ together with the following typing rule.

$$\frac{\Gamma_1; \Theta_1 \vdash e_1 : A \rightarrow_{\omega} B \quad \Gamma_2; \Theta_2 \vdash e_2 : B \rightarrow_{\omega} A \quad \Gamma_3; \Theta_3 \vdash e_3 : A^{\bullet}}{\omega\Gamma_1 + \omega\Gamma_2 + \Gamma_3; \omega\Theta_1 + \omega\Theta_2 + \Theta_3 \vdash \mathbf{lift}\ e_1\ e_2\ e_3 : B^{\bullet}}\ \text{T-LIFT}$$

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

$$\frac{\mu \vdash E_3 \Rightarrow v_3 \quad e_1[v_3/x_1] \Downarrow v}{\mu \vdash \mathbf{lift}\ (\lambda_{\omega}x_1.e_1)\ (\lambda_{\omega}x_2.e_2)\ E_3 \Rightarrow v} \qquad \frac{e_2[v/x_2] \Downarrow v_3 \quad E_3 \Leftarrow v_3 \dashv \mu}{\mathbf{lift}\ (\lambda_{\omega}x_1.e_1)\ (\lambda_{\omega}x_2.e_2)\ E_3 \Leftarrow v \dashv \mu}$$

The substitution lemma (Lemma 3.1) and the subject reduction properties (Lemma 3.2) are also lifted to **lift**.

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

[13] The operator is named after the trace operator (Joyal et al. 1996) in the category of bijections (Abramsky et al. 2002).

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 establish the invariant that the multiplicities of the variables in Θ 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 $\lambda^{\mathrm{PI}}_{\rightarrow}$ 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.

$$A, B ::= A \rightarrow_\pi B \mid () \mid A \otimes B \mid A \oplus B \mid !_\pi A \mid \alpha \mid \mu\alpha.A \mid A^\bullet \mid A \rightleftharpoons B$$

As one can see, there are no user-defined types $\mathsf{T}\,\overline{A}$ 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 π-many copyability of the values.

The expressions are updated to match the types.

$$\begin{array}{rl}
e ::= & x \mid \lambda_\pi x.e \mid e_1\ e_2 \\
\mid & () \mid \mathbf{let}_\pi\ () = e_1\ \mathbf{in}\ e_2 \mid (e_1, e_2) \mid \mathbf{let}_\pi\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2 \\
\mid & \mathsf{InL}\ e \mid \mathsf{InR} \mid \mathbf{case}_\pi\ e_0\ \mathbf{of}\ \{\mathsf{InL}\ x_1 \rightarrow e_1; \mathsf{InR}\ x_2 \rightarrow e_2\} \\
\mid & !_\pi e \mid \mathbf{let}_\pi\ !x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{roll}\ e \mid \mathbf{unroll}\ e \\
\mid & x^\bullet \mid ()^\bullet \mid \mathbf{let}\ ()^\bullet = e_1\ \mathbf{in}\ e_2 \mid (e_1, e_2)^\bullet \mid \mathbf{let}\ (x_1, x_2)^\bullet = e_1\ \mathbf{in}\ e_2 \\
\mid & \mathsf{InL}^\bullet\ e \mid \mathsf{InR}^\bullet\ e \mid \mathbf{case}\ e_0\ \mathbf{of}\ \{(\mathsf{InL}\ x)^\bullet \rightarrow e_1; (\mathsf{InR}\ x)^\bullet \rightarrow e_2\}\ \mathbf{with}\ e' \\
\mid & \mathbf{roll}^\bullet\ e \mid \mathbf{unroll}^\bullet\ e \\
\mid & \mathbf{pin}\ e_1\ e_2 \mid \mathbf{unlift}\ e \mid e_1 \triangleright e_2 \mid e_1 \triangleright e_2
\end{array}$$

Instead of constructors and pattern matching, this version includes the introduction and elimination forms for each form of types except $A^\bullet$. And for types $()$, $A \otimes B$, $A \oplus B$ and $\mu\alpha.A$, there are corresponding invertible versions. For example, we have the introduction form (e_1, e_2) and the elimination form $\mathbf{let}_\pi\ (x_1, x_2) = e_1\ \mathbf{in}\ e_2$ for the product types, and their invertible counterparts $(e_1, e_2)^\bullet$, and $\mathbf{let}\ (x_1, x_2)^\bullet = e_1\ \mathbf{in}\ e_2$. Here, π ensures that e_1 is used π-many times and so as the variables x_1 and x_2, similarly to the π of $\mathbf{case}_\pi$ in the original calculus $\lambda^{\mathrm{PI}}_{\rightarrow}$ (see T-CASE in Fig. 3). Note that both (unidirectional and invertible) sorts of **case**s 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 $x^{\bullet}$, which will be typed by T-VAR and T-RVAR, respectively.

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

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

An intuition underlying the rules is that $!_{\pi}A$ is treated as a GADT ($\mathsf{Many}\ \pi\ A$) with the constructor $\mathsf{MkMany} : A \multimap_{\pi} \mathsf{Many}\ \pi\ A$ capturing the multiplicity π. As the constructor discharges the multiplicity π when pattern matched, the latter rule says that the copyability π_1 is discharged by the binding, regardless of the use of the examined expression e_1. For example, we have $x :_1 (), y :_{\omega} A; \varepsilon \vdash \mathbf{let}_1\ !z = (\mathbf{let}_1\ () = x\ \mathbf{in}\ !_{\omega}y)\ \mathbf{in}\ (z,z) : A \otimes A$ where x is used once in the expression but z can be used twice as the binding discharged the copyability witnessed by $!_{\omega}y$.

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

$$\begin{array}{rcl}
v & ::= & \langle \lambda_{\pi}x.e, \theta\rangle \mid () \mid (v_1, v_2) \mid \mathsf{InL}\ v \mid \mathsf{InR}\ v \mid !_{\pi}v \mid \mathbf{roll}\ v \mid \langle x.E\rangle \mid E \\
E & ::= & x^{\bullet} \mid ()^{\bullet} \mid \mathbf{let}\ ()^{\bullet} = E_1\ \mathbf{in}\ E_2 \mid (E_1, E_2)^{\bullet} \mid \mathbf{let}\ (x_1, x_2)^{\bullet} = E_1\ \mathbf{in}\ E_2 \\
& \mid & \mathsf{InL}^{\bullet}\ E \mid \mathsf{InR}^{\bullet}\ E \mid \mathbf{case}\ E_0\ \mathbf{of}\ \{\langle(\mathsf{InL}\ x)^{\bullet} \to e_1, \theta_1\rangle; \langle(\mathsf{InR}\ x)^{\bullet} \to e_2, \theta_2\rangle\}\ \mathbf{with}\ v' \\
& \mid & \mathbf{roll}^{\bullet}\ E \mid \mathbf{unroll}^{\bullet}\ E \\
& \mid & \mathbf{pin}\ E\ v
\end{array}$$

We intentionally used different metavariables θ and μ 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.

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

The typing rules for values and residuals ($\Theta \vdash v : A$ and $\Theta \vdash E : A^{\bullet}$) are obtained straightforwardly from the rules for expressions (Fig. 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 $\langle \lambda x_{\pi}.e, \theta\rangle$, which comes with the following typing rule.

$$\frac{\Theta_{\mathrm{env}} \vdash \theta : \Gamma \quad x :_{\pi} A, \Gamma;\Theta_{\mathrm{body}} \vdash e : B}{\Theta_{\mathrm{env}} + \Theta_{\mathrm{body}} \vdash \langle \lambda x_{\pi}.e, \theta\rangle : A \to_{\pi} B}$$

The other is the invertible **case** residual, which has the following type rule.

$$\frac{\Theta_0 \vdash E : (A_1 \oplus A_2)^{\bullet} \quad \{\Theta_{\mathrm{env}} \vdash \theta_i : \Gamma_i \quad \Gamma_i; x :_1 A_i, \Theta_{\mathrm{body}} \vdash e_i : C^{\bullet}\}_i \quad \varepsilon \vdash v' : C \to_{\omega} \mathsf{Bool}}{\Theta_0 + \Theta_{\mathrm{env}} + \Theta_{\mathrm{body}} \vdash \mathbf{case}\ E_0\ \mathbf{of}\ \{\langle(\mathsf{InL}\ x)^{\bullet} \to e_1, \theta_1\rangle; \langle(\mathsf{InR}\ x)^{\bullet} \to e_2, \theta_2\rangle\}\ \mathbf{with}\ v' : C^{\bullet}}$$

Here, we write Bool 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, $\Gamma = \{x_1 :_{\pi_1} A_1, \dots, x_n :_{\pi_n} A_n\}$ is separated into $\Gamma_\mathrm{t} = \{x_1 : A_1, \dots, x_n : A_n\}$ and $\Gamma_\mathrm{m} = \{x_1 : \pi_1, \dots, x_n : \pi_n\}$, so that complex manipulation of multiplicities happens only for the latter. Also, Θ environments are separated into Θ_t and Θ_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 $\Gamma; \Theta \vdash e : A$ to produce a value $\Theta' + \Theta \vdash v : A$ if terminates, and the latter takes a residual $\Theta \vdash E : A^\bullet$ 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 $\Theta' + \Theta \vdash v : A$ are represented by types *ValEnv* $\Gamma_\mathrm{t}\ \Gamma_\mathrm{m}\ \Theta_\mathrm{t}\ \Theta'_\mathrm{m}$, *Term* $\Gamma_\mathrm{t}\ \Gamma_\mathrm{m}\ \Theta_\mathrm{t}\ \Theta_\mathrm{m}\ A$, and *Value* $\Theta_\mathrm{t}\ (\Theta'_\mathrm{m} +_\mathrm{m} \Theta_\mathrm{m})\ A$, respectively. Recall that we have adopted the separate treatment of types and multiplicities. Hence, instead of having a single Θ, we have Θ_t and Θ_m where the former typing environment is treated in the usual way. Also, regarding the latter evaluation, a residual-in-context $\Theta \vdash E : A^\bullet$, a typed-environment (for the forward/backward evaluation) $\mu : \Theta$, and a value-in-context $\vdash v : A$ are represented by types *Residual* $\Theta_\mathrm{t}\ \Theta_\mathrm{m}\ (A\bullet)$, *RValEnv* $\Theta_\mathrm{t}\ \Theta_\mathrm{m}$, and *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.

$$
\begin{aligned}
&\mathit{eval} : \forall \{\Theta_\mathrm{t}\ \Theta'_\mathrm{m}\ \Gamma_\mathrm{t}\ \Gamma_\mathrm{m}\ \Theta_\mathrm{m}\ A\}\ (i : \mathit{Size}) \to \mathit{all\text{-}no\text{-}omega}\ (\Theta'_\mathrm{m} +_\mathrm{m} \Theta_\mathrm{m}) \\
&\qquad \to \mathit{ValEnv}\ \Gamma_\mathrm{t}\ \Gamma_\mathrm{m}\ \Theta_\mathrm{t}\ \Theta'_\mathrm{m} \to \mathit{Term}\ \Gamma_\mathrm{t}\ \Gamma_\mathrm{m}\ \Theta_\mathrm{t}\ \Theta_\mathrm{m}\ A \\
&\qquad \to \mathit{DELAY}\ (\mathit{Value}\ \Theta_\mathrm{t}\ (\Theta'_\mathrm{m} +_\mathrm{m} \Theta_\mathrm{m})\ A)\ i \\
&\mathit{evalR} : \forall \{\Theta_\mathrm{t}\ \Theta_\mathrm{m}\ A\}\ (i : \mathit{Size}) \to \mathit{all\text{-}no\text{-}omega}\ \Theta_\mathrm{m} \to \mathit{Residual}\ \Theta_\mathrm{t}\ \Theta_\mathrm{m}\ (A\bullet) \\
&\qquad \to i \vdash_\mathrm{F} \mathit{RValEnv}\ \Theta_\mathrm{t}\ \Theta_\mathrm{m} \Leftrightarrow \mathit{Value}\ []\ \emptyset\ A
\end{aligned}
$$

The predicate *all-no-omega* asserts that a given multiplicity environment does not contain the multiplicity ω, respecting the assumption on the core system that the multiplicities involved in Θ 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 (Abel and Chapman 2014; Capretta 2005), 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 $i \vdash_\mathrm{F} a \Leftrightarrow b$ represents not-necessarily-total bijections and has two fields: *Forward* : $a \to$ *DELAY* $b\ i$ and *Backward* : $b \to$ *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 *DELAY*, the evaluations are allowed to go into infinite loops, which is legitate for the progress property. We also use infinite loops to represent errors, which are thrown only in the following situations.

- forward evaluation of invertible **case**s with imprecise **with** conditions, and
- backward evaluation of $\mathsf{InL}^\bullet\ E$ and $\mathsf{InR}^\bullet\ E$ 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.

Caveat: sized types. As their signatures suggest, the definitions of *eval* and *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,[14] and, as far as we are aware, the safe treatment of sized types is still open in Agda. Nevetheless, 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 *eval* and *evalR* to ensure productivity, and then use the infinite size when we discuss the property of the computation).

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:

$$
\begin{array}{l}
\mathit{forward\text{-}backward} : \\
\quad \forall \{\Theta_\mathrm{t}\ \Theta_\mathrm{m}\ A\} \to (\mathit{ano} : \mathit{all\text{-}no\text{-}omega}\ \Theta_\mathrm{m}) \to (E : \mathit{Residual}\ \Theta_\mathrm{t}\ \Theta_\mathrm{m}\ (A\bullet)) \\
\quad\ \to \forall \mu\ v \\
\quad\ \to \mathit{Forward}\ (\mathit{evalR}\ \infty\ \mathit{ano}\ E)\ \mu \longrightarrow v \to \mathit{Backward}\ (\mathit{evalR}\ \infty\ \mathit{ano}\ E)\ v \longrightarrow \mu \\
\mathit{backward\text{-}forward} : \\
\quad \forall \{\Theta_\mathrm{t}\ \Theta_\mathrm{m}\ A\} \to (\mathit{ano} : \mathit{all\text{-}no\text{-}omega}\ \Theta_\mathrm{m}) \to (E : \mathit{Residual}\ \Theta_\mathrm{t}\ \Theta_\mathrm{m}\ (A\bullet)) \\
\quad\ \to \forall \mu\ v \\
\quad\ \to \mathit{Backward}\ (\mathit{evalR}\ \infty\ \mathit{ano}\ E)\ v \longrightarrow \mu \to \mathit{Forward}\ (\mathit{evalR}\ \infty\ \mathit{ano}\ E)\ \mu \longrightarrow v
\end{array}
$$

Here, $m \longrightarrow v$, which reads that m evaluates to v, is an inductively-defined predicate asserting that $m : \mathit{DELAY}\ a\ \infty$ terminates and produces the final outcome v. This relation has a similar role to $\Sigma\ (m \Downarrow)\ (\lambda m \to \mathit{extract}\ w \equiv v)$, where $\_\Downarrow$ and *extract* are defined in the module `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 straightforwardly by induction on E and case analysis on $\mathit{Forward}\ (\mathit{evalR}\ \infty\ \mathit{ano}\ E)\ \mu \longrightarrow v$ or $\mathit{Backward}\ (\mathit{evalR}\ \infty\ \mathit{ano}\ E)\ v \longrightarrow \mu$, 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

[14] See, e.g., https://github.com/agda/agda/issues/1201 and https://github.com/agda/agda/issues/6002.

trees from preorder and inorder traversals (Mu and Bird 2003), and the latter three are simplified versions of compression algorithms (Salomon 2008), namely, the Huffman coding, arithmetic coding, and LZ77 (Ziv and Lempel 1977).[15]

5.1 Rebuilding Trees from a Pre-Order and an In-Order 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

$$\mathbf{data}\ \mathsf{Tree} = \mathsf{L} \mid \mathsf{N}\ \mathsf{Int}\ \mathsf{Tree}\ \mathsf{Tree}$$

the following Haskell function *pi* is bijective.

$$\begin{aligned}
&pi :: \mathsf{Tree} \to ([\mathsf{Int}], [\mathsf{Int}]) \\
&pi\ t = (preorder\ t, inorder\ t) \\
&preorder\ \mathsf{L} && = [\,] \\
&preorder\ (\mathsf{N}\ a\ l\ r) && = a : preorder\ l +\!\!+ preorder\ r \\
&inorder\ \mathsf{L} && = [\,] \\
&inorder\ (\mathsf{N}\ a\ l\ r) && = inorder\ l +\!\!+ [a] +\!\!+ inorder\ r
\end{aligned}$$

For example, for binary trees

$$\begin{aligned}
t_1 &= \mathsf{N}\ 1\ (\mathsf{N}\ 2\ (\mathsf{N}\ 3\ \mathsf{L}\ \mathsf{L})\ \mathsf{L})\ \mathsf{L}, & t_2 &= \mathsf{N}\ 1\ (\mathsf{N}\ 2\ \mathsf{L}\ (\mathsf{N}\ 3\ \mathsf{L}\ \mathsf{L}))\ \mathsf{L}, \\
t_3 &= \mathsf{N}\ 1\ (\mathsf{N}\ 2\ \mathsf{L}\ \mathsf{L})\ (\mathsf{N}\ 3\ \mathsf{L}\ \mathsf{L}), & t_4 &= \mathsf{N}\ 1\ \mathsf{L}\ (\mathsf{N}\ 2\ (\mathsf{N}\ 3\ \mathsf{L}\ \mathsf{L})\ \mathsf{L}), \\
t_5 &= \mathsf{N}\ 1\ \mathsf{L}\ (\mathsf{N}\ 2\ \mathsf{L}\ (\mathsf{N}\ 3\ \mathsf{L}\ \mathsf{L}))
\end{aligned}$$

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

$$\begin{aligned}
inorder\ t_1 &= [3,2,1], & inorder\ t_2 &= [2,3,1], \\
inorder\ t_3 &= [2,1,3], & inorder\ t_4 &= [1,3,2], \\
inorder\ t_5 &= [1,2,3].
\end{aligned}$$

The uniqueness of labels is key to the bijectivity of *pi*. It is clear that pi^{-1} returns L for $([\,],[\,])$, so the non-trivial part is how pi^{-1} will do for a pair of non-empty 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 $pi^{-1}(take\ (length\ i_1)\ p, i_1)$, we can recover the left child l, and, by $pi^{-1}(drop\ (length\ i_1)\ p, i_2)$, we can recover the right child r. After that, from a, l, and r, we can construct the original input as $\mathsf{N}\ a\ l\ r$. 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 and Bird 2003).

[15] 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.

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 and 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 and Yokoyama (2019) gives a reversible version of tree rebuilding using (an extension of) `R-WHILE` (Glück and Yokoyama 2016), a reversible imperative language inspired by Janus (Lutz 1986; Yokoyama et al. 2008). However, `R-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 *goSubs* and *goSubsF* examples in Fig. 2.

5.1.1 Calculation of the Original Definition

The first step is tupling (Hu et al. 1997; Chin 1993) 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).

$$
\begin{array}{l}
\mathit{pi} :: \mathsf{Tree} \to ([\mathsf{Int}], [\mathsf{Int}]) \\
\mathit{pi}\ \mathsf{L} \qquad\quad = ([\,], [\,]) \\
\mathit{pi}\ (\mathsf{N}\ a\ l\ r) = \mathbf{let}\ (\mathit{pr}, \mathit{ir}) = \mathit{pi}\ r;\ \ (\mathit{pl}, \mathit{il}) = \mathit{pi}\ l\ \mathbf{in}\ (a : \mathit{pl} +\!\!+ \mathit{pr}, \mathit{il} +\!\!+ [a] +\!\!+ \mathit{ir})
\end{array}
$$

Mu and 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 $\mathit{piA}\ t\ \mathit{py}\ \mathit{iy} = \mathbf{let}\ (p, i) = \mathit{pi}\ t\ \mathbf{in}\ (p +\!\!+ \mathit{py}, i +\!\!+ \mathit{iy})$ as below.

$$
\begin{array}{l}
\mathit{piA} :: \mathsf{Tree} \to [\mathsf{Int}] \to [\mathsf{Int}] \to ([\mathsf{Int}], [\mathsf{Int}]) \\
\mathit{piA}\ \mathsf{L} \qquad\quad \mathit{py}\ \mathit{iy} = (\mathit{py}, \mathit{iy}) \\
\mathit{piA}\ (\mathsf{N}\ a\ l\ r)\ \mathit{py}\ \mathit{iy} = \mathbf{let}\ (\mathit{pr}, \mathit{ir}) = \mathit{piA}\ r\ \mathit{py}\ \mathit{iy};\ \ (\mathit{pl}, \mathit{il}) = \mathit{piA}\ l\ \mathit{pr}\ (a : \mathit{ir})\ \mathbf{in}\ (a : \mathit{pl}, \mathit{il})
\end{array}
$$

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 $\mathit{pi}\ x = \mathit{piA}\ x\ [\,]\ [\,]$), and the other is the case where it is not (e.g., the recursion for the left child $\mathit{piA}\ l\ \mathit{pr}\ (a : \mathit{ir})$). 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 *piA* satisfying $\mathit{pi}\ x = \mathit{piA}\ x\ [\,]\ [\,]$ (we reuse the name as it implements the same function).

$$
\begin{array}{l}
\mathit{pi} :: \mathsf{Tree} \to ([\mathsf{Int}], [\mathsf{Int}]) \\
\mathit{pi}\ \mathsf{L} \qquad\quad = ([\,], [\,]) \\
\mathit{pi}\ (\mathsf{N}\ a\ l\ r)\ \ = \mathbf{let}\ (\mathit{pr}, \mathit{ir}) = \mathit{pi}\ r;\ \ (\mathit{pl}, \mathit{il}) = \mathit{piA}\ l\ \mathit{pr}\ (a : \mathit{ir})\ \mathbf{in}\ (a : \mathit{pl}, \mathit{il})
\end{array}
$$

Having this new version of *pi*, we now have an invariant that *iy* of $\mathit{piA}\ t\ \mathit{py}\ \mathit{iy}$ is always non-empty; 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,

$$
\begin{array}{l}
\mathit{piR} : \mathsf{Tree}^{\bullet} \multimap (\mathsf{List\ Int} \otimes \mathsf{List\ Int})^{\bullet} \\
\mathit{piR}\ \mathsf{L}^{\bullet} \qquad = (\mathsf{Nil}^{\bullet}, \mathsf{Nil}^{\bullet})^{\bullet} \\
\qquad\qquad\qquad \mathbf{with}\ \mathit{null} \circ \mathit{fst} \\
\mathit{piR}\ (\mathsf{N}\ a\ l\ r)^{\bullet} = \\
\quad \mathbf{let}\ (\mathit{pr}, \mathit{ir})^{\bullet} = \mathit{piR}\ r\ \mathbf{in} \\
\quad \mathbf{let}\ (a, (\mathit{pl}, \mathit{il}))^{\bullet} = \\
\qquad\quad \mathbf{pin}\ a\ (\lambda a'.\mathit{piASR}\ a'\ l\ \mathit{pr}\ \mathit{ir})\ \mathbf{in} \\
\quad (\mathsf{Cons}^{\bullet}\ a\ \mathit{pl}, \mathit{il})^{\bullet}
\end{array}
$$

$$
\begin{array}{l}
\mathit{piASR} : \mathsf{Int} \to \mathsf{Tree}^{\bullet} \multimap (\mathsf{List\ Int})^{\bullet} \\
\qquad\qquad \multimap (\mathsf{List\ Int} \otimes \mathsf{List\ Int})^{\bullet} \\
\mathit{piASR}\ h\ L^{\bullet} \qquad \mathit{py}\ \mathit{iy} = (\mathit{py}, \mathsf{Cons}^{\bullet}\ (\mathit{new}\ \mathit{eqInt}\ h)\ \mathit{iy})^{\bullet} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{with}\ \mathit{eqInt}\ h \circ \mathit{head} \circ \mathit{snd} \\
\mathit{piASR}\ h\ (\mathsf{N}\ a\ l\ r)^{\bullet}\ \mathit{py}\ \mathit{iy} = \\
\quad \mathbf{let}\ (\mathit{pr}, \mathit{ir})^{\bullet} = \mathit{piASR}\ h\ r\ \mathit{py}\ \mathit{iy}\ \mathbf{in} \\
\quad \mathbf{let}\ (a, (\mathit{pl}, \mathit{il}))^{\bullet} = \mathbf{pin}\ a\ (\lambda a'.\mathit{piASR}\ a'\ l\ \mathit{pr}\ \mathit{ir})\ \mathbf{in} \\
\quad (\mathsf{Cons}^{\bullet}\ a\ \mathit{pl}, \mathit{il})^{\bullet}
\end{array}
$$

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

we can distinguish the ranges of the two branches of *piA*: 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 $\mathit{piA}\ t\ \mathit{py}\ \mathit{iy} = \mathbf{let}\ (p, i) = \mathit{pi}\ t\ \mathbf{in}\ (p +\!\!+\ \mathit{py}, i +\!\!+\ \mathit{iy})$; thus, ir in the definition of *piA* must have the form of $\cdots +\!\!+\ \mathit{iy}$, and then il must have the form of $\cdots +\!\!+ [a] +\!\!+ \cdots +\!\!+\ \mathit{iy}$.

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 *piA*, by changing it to a separate parameter. Specifically, we prepare the function *piAS* satisfying $\mathit{piAS}\ h\ t\ \mathit{py}\ \mathit{iy} = \mathit{piA}\ t\ \mathit{py}\ (h : \mathit{iy})$ as below.

$$
\begin{array}{l}
\mathit{piAS} :: \mathsf{Int} \to \mathsf{Tree} \to [\mathsf{Int}] \to [\mathsf{Int}] \to ([\mathsf{Int}], [\mathsf{Int}]) \\
\mathit{piAS}\ h\ \mathsf{L} \qquad\quad \mathit{py}\ \mathit{iy} = (\mathit{py}, h : \mathit{iy}) \\
\mathit{piAS}\ h\ (\mathsf{N}\ a\ l\ r)\ \mathit{py}\ \mathit{iy} = \mathbf{let}\ (\mathit{pr}, \mathit{ir}) = \mathit{piAS}\ h\ r\ \mathit{py}\ \mathit{iy};\ (\mathit{pl}, \mathit{il}) = \mathit{piAS}\ a\ l\ \mathit{pr}\ \mathit{ir}\ \mathbf{in}\ (a : \mathit{pl}, \mathit{il})
\end{array}
$$

Also, we replace the function call of *piA* in *pi* appropriately.

$$
\begin{array}{l}
\mathit{pi} :: \mathsf{Tree} \to ([\mathsf{Int}], [\mathsf{Int}]) \\
\mathit{pi}\ \mathsf{L} \qquad\quad = ([\,], [\,]) \\
\mathit{pi}\ (\mathsf{N}\ a\ l\ r)\ = \mathbf{let}\ (\mathit{pr}, \mathit{ir}) = \mathit{pi}\ r;\ (\mathit{pl}, \mathit{il}) = \mathit{piAS}\ a\ l\ \mathit{pr}\ \mathit{ir}\ \mathbf{in}\ (a : \mathit{pl}, \mathit{il})
\end{array}
$$

5.1.2 Making Partial-Invertibility Explicit

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

$$
\begin{array}{l}
\mathit{new} : (a \to a \to \mathsf{Bool}) \to a \to a^{\bullet} \\
\mathit{new}\ \mathit{eq}\ c = \mathbf{lift}\ (\lambda\_.c)\ (\lambda c'.\ \mathbf{case}\ \mathit{eq}\ c\ c'\ \mathbf{of}\ \{\mathsf{True} \to (\,)\})\ (\,)^{\bullet}
\end{array}
$$

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 *piR* has the same behavior as that Mu and Bird (2003, Section 3) derived. The partial bijection that *piASR* 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.

$$
\begin{array}{l}
\mathit{checkSum} : \mathsf{List\ Int}^{\bullet} \multimap \mathsf{List\ Int}^{\bullet} \\
\mathit{checkSum}\ \mathit{xs} = \\
\quad \mathbf{let}\ (\mathit{xs}, s) = \mathbf{pin}\ \mathit{xs}\ (\lambda \mathit{xs}'.\mathit{new}\ \mathit{eqInt}\ (\mathit{sum}\ \mathit{xs}'))\ \mathbf{in} \quad \text{-- } \mathit{sum} : \mathsf{List\ Int} \to \mathsf{Int} \\
\quad \mathsf{Cons}^{\bullet}\ s\ \mathit{xs}
\end{array}
$$

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 *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 $\lambda x.\mathbf{pin}\ x\ (\mathit{new}\ \mathit{eq}) : A^{\bullet} \multimap (A \otimes A)^{\bullet}$, assuming appropriate $\mathit{eq} : A \to A \to \mathsf{Bool}$.

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

$$
\begin{array}{l}
\mathit{delete} : (a \to a \to \mathsf{Bool}) \to a \to a^{\bullet} \multimap (\,)^{\bullet} \\
\mathit{delete}\ \mathit{eq}\ c\ a = \mathbf{lift}\ (\lambda c'.\mathbf{case}\ \mathit{eq}\ c\ c'\ \mathbf{of}\ \{\mathsf{True} \to (\,)\})\ (\lambda\_.c)\ a
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{safeLift} : (a \to a \to \mathsf{Bool}) \to (b \to b \to \mathsf{Bool}) \to (a \to b) \to (b \to a) \to a^{\bullet} \multimap b^{\bullet} \\
\mathit{safeLift}\ \mathit{eqA}\ \mathit{eqB}\ f\ g\ a = \mathbf{let}\ (a, b)^{\bullet} \ \ = \mathbf{pin}\ a\ (\lambda a'.\mathit{new}\ \mathit{eqB}\ (f\ a'))\ \mathbf{in} \\
\phantom{\mathit{safeLift}\ \mathit{eqA}\ \mathit{eqB}\ f\ g\ a = } \mathbf{let}\ (b, (\,))^{\bullet} = \mathbf{pin}\ b\ (\lambda b'.\mathit{delete}\ \mathit{eqA}\ (g\ b')\ a)\ \mathbf{in} \\
\phantom{\mathit{safeLift}\ \mathit{eqA}\ \mathit{eqB}\ f\ g\ a = } b
\end{array}
$$

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.

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 $\underbrace{\texttt{0}}_{\texttt{a}}\ \underbrace{\texttt{0}}_{\texttt{a}}\ \underbrace{\texttt{10}}_{\texttt{b}}\ \underbrace{\texttt{0}}_{\texttt{a}}\ \underbrace{\texttt{110}}_{\texttt{c}}\ \underbrace{\texttt{0}}_{\texttt{a}}\ \underbrace{\texttt{10}}_{\texttt{b}}\ \underbrace{\texttt{111}}_{\texttt{d}}\ \underbrace{\texttt{0}}_{\texttt{a}}\ \underbrace{\texttt{0}}_{\texttt{a}}$, which is smaller than the 20-bit code obtained under the naive encoding that assigns two bits for each symbol.

$$
\begin{array}{l}
\mathit{huffCompress} : (\mathsf{List\ Symbol})^{\bullet} \multimap (\mathsf{Huff} \otimes \mathsf{List\ Bit})^{\bullet} \\
\mathit{huffCompress}\ s = \\
\quad \mathbf{let}\ (s, h)^{\bullet} = \mathbf{pin}\ s\ (\lambda s'.\mathit{new\ eqHuff}\ (\mathit{makeHuff}\ s'))\ \mathbf{in} \\
\quad \mathbf{pin}\ h\ (\lambda h'.\mathit{encode}\ h'\ s) \\[4pt]
\mathit{encode} : \mathsf{Huff} \to (\mathsf{List\ Symbol})^{\bullet} \multimap (\mathsf{List\ Bit})^{\bullet} \\
\mathit{encode}\ h\ \mathsf{Nil}^{\bullet} \qquad\qquad = \mathsf{Nil}^{\bullet}\ \mathbf{with}\ \mathit{null} \\
\mathit{encode}\ h\ (\mathsf{Cons}\ s\ ss)^{\bullet} = \mathit{encR}\ h\ s\ (\mathit{encode}\ h\ ss)
\end{array}
$$

Fig. 6: Two-pass Huffman coding in SPARCL

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.

$$
\begin{array}{l}
\mathit{makeHuff} : \mathsf{List\ Symbol} \to \mathsf{Huff} \\
\mathit{enc} : \mathsf{Huff} \to \mathsf{Symbol} \to \mathsf{List\ Bit} \\
\mathit{dec} : \mathsf{Huff} \to \mathsf{List\ Bit} \to \mathsf{Symbol} \otimes \mathsf{List\ Bit}
\end{array}
$$

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

Then, by *enc* and *dec*, we can define an bijective version *encR* as below.

$$
\begin{array}{l}
\mathit{encR} : \mathsf{Huff} \to \mathsf{Symbol}^{\bullet} \multimap (\mathsf{List\ Bit})^{\bullet} \multimap (\mathsf{List\ Bit})^{\bullet} \\
\mathit{encR}\ h\ s\ r = \mathbf{lift}\ (\lambda(s, \mathit{ys}).\ \mathit{enc}\ h\ s \mathbin{+\!\!+} \mathit{ys})\ (\lambda \mathit{ys}.\ \mathit{dec}\ h\ \mathit{ys})\ (s, r)^{\bullet}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{unsafeNew} : a \to a^{\bullet} \\
\mathit{unsafeNew}\ a = \mathbf{lift}\ (\lambda(\,).a)\ (\lambda a'.(\,)) \quad \text{-- assuming } a = a'
\end{array}
$$

The use of *unsafeNew a* is safe only when its backward execution always receives *a*. Replacing *new* with *unsafeNew* violates this assumption, but for this case, the replacement just widens the domain of **bwd** *huffCompress*, which is acceptable even though

fwd *huffCompress* and **bwd** *huffCompress* do not form a bijection due to *unsafeNew*. But in general this outcome is unreliable, unless the condition above can be guaranteed.

5.2.2 Concrete Representation of Huffman Tree in SPARCL

In the above we have modelled the case where different data structures are used for encoding and decoding, which demands the use of abstract type and consequently the use of **lift**ing. In this section, we define *encR* directly in 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 Huff.

$$\textbf{data}\ \mathsf{Huff} = \mathsf{Lf}\ \mathsf{Symbol} \mid \mathsf{Br}\ \mathsf{Huff}\ \mathsf{Huff}$$

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

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

$$\begin{array}{l}
\mathit{decR} : \mathsf{Huff} \to (\mathsf{List\ Bit})^{\bullet} \multimap (\mathsf{Symbol} \otimes \mathsf{List\ Bit})^{\bullet} \\
\mathit{decR}\ (\mathsf{Lf}\ s)\quad \mathit{ys} = (\mathit{new}\ \mathit{eqSym}\ s, \mathit{ys})^{\bullet} \\
\mathit{decR}\ (\mathsf{Br}\ l\ r)\ \mathit{ys} = \textbf{case}\ \mathit{ys}\ \textbf{of}\ (\mathsf{Cons}\ 0\ \mathit{ys}')^{\bullet} \to \mathit{decR}\ l\ \mathit{ys}'\ \textbf{with}\ \lambda(s, _).\mathit{member}\ s\ l \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{Cons}\ 1\ \mathit{ys}')^{\bullet} \to \mathit{decR}\ r\ \mathit{ys}' \\
\mathit{encR}\ h\ s\ \mathit{ys} = \mathit{invert}\ (\mathit{decR}\ h)\ (s, \mathit{ys})^{\bullet}
\end{array}$$

Here, $\mathit{member} : \mathsf{Symbol} \to \mathsf{Huff} \to \mathsf{Bool}$ is a membership test function. Recall that *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 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.

5.2.3 Adaptive Huffman Coding

In the above *huffCompress*, a Huffman coding table is fixed during compression which requires the preprocessing *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.

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

$$\mathit{initHuff} : \mathsf{Huff} \qquad\qquad \mathit{updHuff} : \mathsf{Symbol} \to \mathsf{Huff} \to \mathsf{Huff}$$

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 (Fig. 7). This is another demonstration of the SPARCL's strength in partial invertibility. Programming the same bijection in a

$$
\begin{aligned}
&\mathit{huffCompress} : (\mathsf{List\ Symbol})^{\bullet} \multimap (\mathsf{Huff} \otimes \mathsf{List\ Bit})^{\bullet} \\
&\mathit{huffCompress} = \mathit{encode}\ \mathit{initHuff} \\
&\mathit{encode} : \mathsf{Huff} \to (\mathsf{List\ Symbol})^{\bullet} \multimap (\mathsf{List\ Bit})^{\bullet} \\
&\mathit{encode}\ h\ \mathsf{Nil}^{\bullet} \qquad\quad = \mathsf{Nil}^{\bullet} \quad \mathbf{with}\ \mathit{null} \\
&\mathit{encode}\ h\ (\mathsf{Cons}\ s\ ss)^{\bullet} = \\
&\quad \mathbf{let}\ (s, r)^{\bullet} = \mathbf{pin}\ s\ (\lambda s'.\mathit{encode}\ (\mathit{updHuff}\ s'\ h)\ ss)\ \mathbf{in} \\
&\quad \mathit{encR}\ h\ s\ r
\end{aligned}
$$

Fig. 7: Adaptive Huffman coding in SPARCL

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 $[l + (r - l)l_s, l + (r - l)r_s)$ when it reads a symbol s to which $[l_s, r_s)$ is associated, and finally yields a real in $[l, r)$. For example, reading a text `aabacabdaa`, 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.

$$
\begin{aligned}
&\mathbf{type}\ \mathsf{Range} = (\mathbb{Q}, \mathbb{Q}) \\
&\mathit{rangeOf} : \mathsf{Symbol} \to \mathsf{Range} \\
&\mathit{find} \quad\ \ : \mathsf{Range} \to \mathbb{Q} \to \mathsf{Symbol}
\end{aligned}
$$

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.

$$
\begin{aligned}
&\mathit{narrow} : \mathsf{Range} \to \mathsf{Range} \to \mathsf{Range} \\
&\mathit{narrow}\ (l, r)\ (l_s, r_s) = (l + (r - l) * l_s, l + (r - l) * r_s)
\end{aligned}
$$

In what follows, *narrow* is used only with *rangeOf*. So, we define the following function for convenience.

$$
\begin{aligned}
&\mathit{narrowBySym} : \mathsf{Range} \to \mathsf{Symbol} \to \mathsf{Range} \\
&\mathit{narrowBySym}\ \mathit{ran}\ s = \mathit{narrow}\ \mathit{ran}\ (\mathit{rangeOf}\ s)
\end{aligned}
$$

These functions satisfy the following property.

$$\begin{aligned} &\mathit{narrowBySym}\ (l,r)\ s = (l',r') \\ &\quad\Rightarrow\quad (l \le l' \wedge r' \le r) \wedge (\forall q \in \mathbb{Q}. l' \le n < r' \Rightarrow \mathit{find}\ (l,r)\ q = s) \end{aligned}$$

Although $\lambda s.\mathit{narrow}\ (l,r)\ (\mathit{rangeOf}\ s)$ 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.

$$\begin{aligned} &\mathit{arithComp} : (\mathsf{List\ Symbol}) \to \mathbb{Q} \\ &\mathit{arithComp} = \mathit{encode}\ (0,1) \\ &\mathit{encode} : \mathsf{Range} \to (\mathsf{List\ Symbol}) \to \mathbb{Q} \\ &\mathit{encode}\ (l,r)\ \mathsf{Nil} \qquad\quad = l \\ &\mathit{encode}\ (l,r)\ (\mathsf{Cons}\ s\ ss) = \mathit{encode}\ (\mathit{narrowBySym}\ (l,r)\ s)\ ss \end{aligned}$$

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 *encode* meats 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 *arithComp*, which means that the range should be treated as invertible. The **pin** operator could be a solution to the issue. Since we want to use the unidirectional function *narrowBySym*, it is natural to **pin** the symbol s to narrow the range, which belongs to the unidirectional world. However, there is a problem. Using **pin** produces an invertible product $(\mathsf{Symbol} \otimes \mathbb{Q})^\bullet$ 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 *delete* function in Section 5.1.3. For this particular case of the arithmetic coding, the following derived version is more convenient.

$$\begin{aligned} &\mathit{deleteBy} : (b \to b \to \mathsf{Bool}) \to (a \to b) \to a^\bullet \multimap b^\bullet \multimap a^\bullet \\ &\mathit{deleteBy}\ \mathit{eq}\ f\ a\ b = \mathbf{let}\ (a,())^\bullet = \mathbf{pin}\ a\ (\lambda a'.\mathit{delete}\ \mathit{eq}\ (f\ a')\ b)\ \mathbf{in} \\ &\qquad\qquad\qquad\qquad a \end{aligned}$$

By using *deleteBy* with *find* (part of the arithmetic encoding API), we can write an invertible version as below.

$$\begin{aligned} &\mathit{arithComp} : (\mathsf{List\ Symbol})^\bullet \multimap \mathbb{Q}^\bullet \\ &\mathit{arithComp} = \mathit{encode}\ (0,1) \\ &\mathit{encode} : \mathsf{Range} \to (\mathsf{List\ Symbol})^\bullet \multimap \mathbb{Q}^\bullet \\ &\mathit{encode}\ (l,r)\ (\mathsf{Nil})^\bullet \qquad\quad = \mathit{new}\ \mathit{eqQ}\ l\ \mathbf{with}\ \mathit{eqQ}\ l \\ &\mathit{encode}\ (l,r)\ (\mathsf{Cons}\ s\ ss)^\bullet = \\ &\quad \mathbf{let}\ (s,q)^\bullet = \mathbf{pin}\ s\ \$\ \lambda s'.\ \mathit{encode}\ (\mathit{narrowBySym}\ (l,r)\ s')\ ss\ \mathbf{in} \\ &\quad \mathit{deleteBy}\ \mathit{eqSym}\ (\mathit{find}\ (l,r))\ q\ s \end{aligned}$$

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 meats 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 and 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[16]. 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 string `abda`, the algorithm yields $(3,2)$`da`, as below

$$\begin{array}{rll} \boxed{\texttt{dabc}}\boxed{\texttt{abd}}\texttt{a} & \text{emits} & (3,2) \\ \texttt{da}\boxed{\texttt{bcab}}\boxed{\texttt{da}\phantom{\texttt{a}}} & \text{emits} & \texttt{d} \\ \texttt{dab}\boxed{\texttt{cabd}}\boxed{\texttt{a}\phantom{\texttt{aa}}} & \text{emits} & \texttt{a} \end{array}$$

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 $\boxed{\texttt{bbba}}\boxed{\texttt{aaa}}$ the algorithm emits $(1,4)$.

The basic idea of our implementation is to use **pin** to convert the input string from invertible to unidirectional to allow overlapping in searching. Hence, we prepare the following

[16] 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

```
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 (λinp′. new eqMatchRes (findMatch w inp)) in
  case matchRes of
    Nothing• →
      let (Cons s ss)• = inp in
      let (s, r)• = pin s (λs′. 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 (λl′. split l inp) in
      let (mstr, r)• = pin mstr (λmstr′. encode (extendWindow w mstr′) rest) in
      let (c, ())• = pin (p, l)• (λc. 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

unidirectional functions for the manipulation of the window, which is an abstract type Window.

$$
\begin{aligned}
&\mathit{emptyWindow} : \mathsf{Window}\\
&\mathit{extendWindow} : \mathsf{Window} \to \mathsf{List\ Symbol} \to \mathsf{Window}\\
&\mathit{findMatch} : \mathsf{Window} \to \mathsf{List\ Symbol} \to \mathsf{Maybe}\ (\mathsf{Int} \otimes \mathsf{Int})\\
&\mathit{takeMatch} : \mathsf{Window} \to (\mathsf{Int} \otimes \mathsf{Int}) \to \mathsf{List\ Symbol}
\end{aligned}
$$

Here, the last two functions satisfy the following property.

$$\mathit{findMatch}\ w\ s = \mathsf{Just}\ (p, l) \Longrightarrow \mathit{takeMatch}\ w\ (p, l) = \mathit{take}\ l\ s$$

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

$$\mathbf{data}\ \mathsf{LZCode} = \mathsf{Lit\ Symbol} \mid \mathsf{Entry}\ (\mathsf{Int} \otimes \mathsf{Int})$$

We do not need to represent the look-ahead buffer explicitly, as it is hidden in the *findMatch* function. Instead of using custom-sized integers, we use 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.

Fig. 8 shows an implementation of an invertible LZ77 compression in SPARCL. We omit the definition of $\mathit{eqMatchRes} : \mathsf{Maybe}\ (\mathsf{Int} \otimes \mathsf{Int}) \to \mathsf{Maybe}\ (\mathsf{Int} \otimes \mathsf{Int}) \to \mathsf{Bool}$ and $\mathit{eqStr} : \mathsf{List\ Symbol} \to \mathsf{List\ Symbol} \to \mathsf{Bool}$. Similarly to the arithmetic coding example, we also use the *new*/*delete* trick here. The property above of *findMatch* and *takeMatch* ensures that the *delete* in *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 *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 *new* with *unsafeNew* is widen the domain of the backward execution, without risking the expected behavior that decompression after compression should yield the original data.

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 (Jones et al. 1993; Gomard and Jones 1991) 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 and Vidal 2006; Kirkeby and 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 *goSubs* succeeds when it inverts $x - n$ first, but fails if it tried to invert *goSubs* x xs first. The design of SPARCL is inspired by these partial inversion methods: we use $(-)^{\bullet}$-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 and Glück 2020) unifies the partial and semi inversion methods based on the authors' reformulation (Kirkeby and 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. 2004; Rendel and Ostermann 2010; Kennedy and Vytiniotis 2012; Wang et al. 2013), in which users program by composing bijections through designated combinators. To the best of our knowledge, only (Kennedy and Vytiniotis 2012) has an operator like $\mathbf{pin} : A^{\bullet} \multimap (A \to B^{\bullet}) \multimap A \otimes B^{\bullet}$, which is key to partial invertibility. More specifically, Kennedy and Vytiniotis (2012) has an operator *depGame* $:: \mathsf{Game}\ a \to (a \to \mathsf{Game}\ b) \to \mathsf{Game}\ (a, b)$. The types suggest that Game and

$(-)^{\bullet}$ 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 $A^{\bullet}$ represents a bijection whose range is A but domain is determined when **unlift** is applied. One consequence of this difference is that, in their domain-specific system, there is no restriction of using a value v :: Game a linearly, because there is no problem of using an encoder/decoder pair for type a multiple times, even though nonlinear use of $v : A^{\bullet}$, 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 $a \to$ Game b, whereas we can obtain a (not-necessarily-total) bijections between A and B from any function of type $A^{\bullet} \multimap B^{\bullet}$ that does not contain linear free variables.

The **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; Yokoyama et al. 2008; Glück and Yokoyama 2016; Frank 1997), of which the inverse is given by $x \ominus= e$ with $\ominus$ satisfying $(x \oplus y) \ominus y = x$ for any y; examples of $\oplus$ (and $\ominus$) include addition, subtraction, bitwise XOR, and replacement of `nil` (Glück and Yokoyama 2016) as a form of reversible copying (Glück and Kawabe 2003). Having $(x \oplus y) \ominus y$ means that $\oplus$ and $\ominus$ are partially invertible, and indicates that they correspond to the second argument of **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 **pin**, which leads to concise function definitions as demonstrated in *goSub* in Section 2 and the examples in Section 5.

Most of the existing reversible programming languages (Yokoyama et al. 2008; Lutz 1986; Frank 1997; Baker 1992; Yokoyama et al. 2011; Mu et al. 2004; 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 λ-calculus concerning $\multimap$ 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 and 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 $A \oplus B \leftrightarrow C \simeq A \leftrightarrow -B \oplus C$, 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 *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,[17] a typed variant of RFun (Yokoyama et al. 2011) with Haskell-like syntax, allows a function to take additional parameters called ancilla parameters. 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 operations 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.[18] The system is overly conservative: all the functions are (partially) reversible, and thus functions themselves used in the ancilla positions must obey the linearity restriction. Jeopardy (Kristensen et al. 2022)[19] is a work-in-progress reversible language, which plans to support partial invertibility via program analysis. The implicit argument analysis (Kristensen et al. 2022), which Jeopardy uses, identifies which arguments 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 crucially, 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 λ-abstraction), which limits flexible programming. In contrast, $A^\bullet$ is an ordinary type in SPARCL, and there is no syntactic restriction on expressions of type $A^\bullet$. This feature, combined with the higher-orderness, gives extra flexibility in mixing unidirectional and invertible programming. For example, 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).

6.2 Functional Quantum Programming Languages

In quantum programming many operation are reversible, and there are a few higher-order quantum programming languages (Selinger and Valiron 2006; Rios and Selinger 2017). Among them, the type system of Proto-Quipper-M (Rios and Selinger 2017) is similar to $\lambda^{\mathrm{PI}}_{\rightarrow}$ in the sense that it also uses a linear-type system and distinguishes two sorts of variable environments as we do with Γ and Θ, 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 Θ), 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 and Valiron 2006) and some allow contraction (i.e., copying) (Altenkirch and 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

[17] https://github.com/kirkedal/rfun-interp
[18] A correction to Jacobsen et al. (2018) (personal communication with Michael-Kirkedal Thomsen, Jun 2020).
[19] Don't confuse it with the program inversion method with the same name (Dershowitz and Mitra 1999).

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 Θ (notice that multiple uses of a variable in Γ will be reduced to multiple uses of variables in Θ (Matsuda and Wang 2018)). This requires the type system to distinguish types that can be in Θ from general ones, as types such as $A \multimap B$ 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 $get : a \to b$ and $put : a \to b \to a$ (Foster et al. 2007). Similarly to ours, in the bidirectional language HOBiT (Matsuda and Wang 2018), a bidirectional transformation between a and b is represented by a function from $\mathbf{B}\ a$ to $\mathbf{B}\ b$, and top-level functions of type $\mathbf{B}\ a \to \mathbf{B}\ b$ 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 **pin** as a solution. Another difference is that 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 and Wang (2018) can only interpret *top-level* functions of type $\mathbf{B}\ a \to \mathbf{B}\ b$ as bidirectional transformations between a and b, yet we can interpret functions of type $A^{\bullet} \multimap B^{\bullet}$ *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 $A^{\bullet} \multimap (A \to B^{\bullet}) \multimap (A \otimes B)^{\bullet}$ of **pin** clarifies that the function argument of **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 Θ 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 and Wang 2015, 2018), which is an embedded domain specific language in Haskell, provides a function *lift* that converts a bidirectional transformation $(a \to b, a \to b \to a)$ to a function of type $\mathsf{L}\ s\ a \to \mathsf{L}\ s\ b$ 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 and Paterson 2008; Paterson 2012) but not endo to be an Applicative instance (Matsuda and Wang 2018).) The type parameter s has a similar role to the s of the $\mathsf{ST}\ s$ monad (Launchbury and Jones 1994), which enables the *unlift*ing that converts a polymorphic function $\forall s.\mathsf{L}\ s\ a \to \mathsf{L}\ s\ b$ 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 **case**s 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. 2004; Hidaka et al. 2010), are not preserved in the presence of unrestricted duplication (Matsuda and Wang 2015).

6.4 Linear Type Systems

SPARCL is based on $\lambda^q_\to$, 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 $(-)^\bullet$-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 ($\circ$ and $*$) to distinguish types that are treated in a linear way ($\circ$) from those that are not ($*$). Thanks to the subkinding $* \preceq \circ$, no syntactic annotations are required to convert the unrestricted values to linear ones. Their system has two sort of function types: $\xrightarrow{\circ}$ for the functions that themselves are treated in the linear way and $\xrightarrow{*}$ 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 and 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 OUTSIDEIN(X) (Vytiniotis et al. 2011), also uses qualified typing with inequality constraints for $\lambda^q_\to$, 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 $(-)^{\bullet}$ 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 and 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 and Wang 2018). 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. 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. Ershov Memorial Conference. 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. *Mathematical Structures in Computer Science*. **12**(5), 625–665.

Almendros-Jiménez, J. M. & Vidal, G. (2006) Automatic partial inversion of inductively sequential functions. Implementation and Application of Functional Languages, 18th International Symp osium, 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. 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. 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. 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. 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 Journal of Research and Development*. **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 in Computer Science*. **1**(2).

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

Dershowitz, N. & Mitra, S. (1999) Jeopardy. 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. 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).

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). vol. 2297 of *Lecture Notes in Computer Science*. 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. APLAS. Springer. pp. 246–264.

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

Glück, R. & Yokoyama, T. (2016) A linear-time self-interpreter of a reversible imperative language. *Computer Software*. **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. 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., Takeichi, M. & Takano, A. (1997) Tupling calculation eliminates multiple data traversals. 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. & Takeichi, M. (2004) A programmable editor for developing structured documents based on bidirectional transformations. 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. 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. 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.

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

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. *Mathematical Proceedings of the Cambridge Philosophical Society*. **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. 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. 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. (2022) Branching execution symmetry in jeopardy by available implicit arguments analysis. *CoRR*. **abs/2212.03161**.

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

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

Landauer, R. (1961) Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*. **5**(3), 183–191.

Launchbury, J. & Jones, S. L. P. (1994) Lazy functional state threads. 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*. vol. 5 of *Graduate Texts in Matheematics*. Springer. second edition edition.

Matsuda, K. (2020) Modular inference of linear types for multiplicity-annotated arrows. 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. 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 and Symbolic Computation*. **25**(1), 3–38.

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

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

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

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

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

Matsuda, K. & Wang, M. (2018) Embedding invertible languages with binders: a case of the FliPpr language. 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. (2018) HOBiT: Programming lenses without using lens combinators. 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. 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. 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. Ershov Memorial Conference. Springer. pp. 322–334.

Mogensen, T. Æ. (2008) Semi-inversion of functional parameters. 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. 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. 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.

Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings. Springer. pp. 265–282.

Mu, S., Hu, Z. & Takeichi, M. (2004) An algebraic approach to bi-directional updating. 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. (2004) An injective language for reversible computation. Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings. Springer. pp. 289–313.

Nielson, F. & Nielson, H. R. (1992) *Two-Level Functional Languages*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Nishida, N., Sakai, M. & Sakabe, T. (2005) Partial inversion of constructor term rewriting systems. 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. RTA. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. pp. 283–298.

Paterson, R. (2012) Constructing applicative functors. MPC. Springer. pp. 300–323.

Rendel, T. & Ostermann, K. (2010) Invertible syntax descriptions: unifying parsing and pretty printing. Haskell. ACM. pp. 1–12.

Reynolds, J. C. (1998) Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*. **11**(4), 363–397.

Rios, F. & Selinger, P. (2017) A categorical model for a quantum circuit description language. 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. 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. 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. *Mathematical Structures in Computer Science*. **16**(3), 527–552.

Srivastava, S., Gulwani, S., Chaudhuri, S. & Foster, J. S. (2011) Path-based inductive synthesis for program inversion. 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*. vol. 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Tov, J. A. & Pucella, R. (2011) Practical affine types. 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.

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. 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) Substractural type systems. In *Advanced Topics in Types and Programming Languages*, Pierce, B. C. (eds). MIT Press. pp. 3–43.

Wang, M., Gibbons, J., Matsuda, K. & Hu, Z. (2013) Refactoring pattern matching. *Science of Computer Programming*. **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. 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. 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. RC. Springer. pp. 14–29.
Yokoyama, T., Axelsen, H. B. & Glück, R. (2012) Optimizing reversible simulation of injective functions. *Multiple-Valued Logic and Soft Computing*. **18**(1), 5–24.
Ziv, J. & Lempel, A. (1977) A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*. **23**(3), 337–343.

1 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 $\lambda^{\mathrm{PI}}_{\rightarrow}$ for construction, but the discussions in this section can be adapted straightforwardly to $\lambda^{\mathrm{PI}}_{\rightarrow}$.

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).

1.1 Reversible Turing Machines

We start with reviewing ordinary Turing machines.

Definition 1.1 (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*, Σ is a finite set of *symbols*, δ 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 Σ 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 δ 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 σ_1, then it writes σ_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 1.2 (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 (σ_1, σ_2) and (σ_1', σ_2'), respectively, and $\sigma_1 \neq \sigma_1'$.
- If $q_2 = q_2'$, then a and a' must have the forms (σ_1, σ_2) and (σ_1', σ_2'), respectively, and $\sigma_2 \neq \sigma_2'$. ■

1.2 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_Σ for states and symbols, and $\mathsf{Q}_q : \mathsf{T}_Q$ and $\mathsf{S}_\sigma : \mathsf{T}_\Sigma$ for constructors corresponding to $q \in Q$ and $\sigma \in \Sigma$, respectively. Then, a type for tapes is give by a product $\mathsf{Tape} = \mathsf{List}\ \mathsf{T}_\Sigma \otimes \mathsf{T}_\Sigma \otimes \mathsf{List}\ \mathsf{T}_\Sigma$, where a triple $(l, a, r) : \mathsf{Tape}$ 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 $\mathsf{S}_\sqcup$ if they are not empty.

Then, we prepare the function *moveR* below that moves the head to the right.

$$
\begin{array}{ll}
\mathit{moveR} : \mathsf{Tape}^\bullet \multimap \mathsf{Tape}^\bullet & \mathit{push} : (\mathsf{T}_\Sigma \otimes \mathsf{List}\ \mathsf{T}_\Sigma)^\bullet \multimap (\mathsf{List}\ \mathsf{T}_\Sigma)^\bullet \\
\mathit{moveR}\ (l, a, r)^\bullet = \mathbf{let}\ (a', l')^\bullet = \mathit{invert}\ \mathit{push}\ l\ \mathbf{in} & \mathit{push}\ (\mathsf{S}_\sqcup, \mathsf{Nil})^\bullet = \mathsf{Nil}^\bullet\ \mathbf{with}\ \mathit{null} \\
\qquad\qquad\qquad\qquad (l', a', \mathit{push}\ (a, r)^\bullet)^\bullet & \mathit{push}\ (a, \mathit{xs})^\bullet \quad = \mathsf{Cons}^\bullet\ a\ \mathit{xs}
\end{array}
$$

Here, $(-, \ldots, -)^\bullet$ is a lifted version of the tuple constructor $(-, \ldots, -)$, $\mathbf{let}\ p^\bullet = e\ \mathbf{in}\ e'$ is a shorthand notation for $\mathbf{case}\ e\ \mathbf{of}\ \{p^\bullet \to e'\ \mathbf{with}\ \lambda\_.\mathsf{True}\}$, and the function $\mathit{invert} : (A^\bullet \multimap B^\bullet) \to B^\bullet \multimap A^\bullet$ implements the inversion of a invertible function (Section 2).

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

$$
\begin{array}{l}
\mathit{step} : (\mathsf{T}_Q \otimes \mathsf{Tape})^\bullet \multimap (\mathsf{T}_Q \otimes \mathsf{Tape})^\bullet \\
\mathit{step}\ t = \mathbf{case}\ t\ \mathbf{of}\ \{[\![r]\!]\}_{r \in \delta}
\end{array}
$$

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

$$
\begin{array}{rcll}
[\![(q_1, (\sigma_1, \sigma_2), q_2)]\!] & = & (\mathsf{Q}_{q_1}, (l, \mathsf{S}_{\sigma_1}, r))^\bullet \to (\mathsf{Q}_{q_2}, (l, \mathsf{S}_{\sigma_2}, r))^\bullet & \\
& & \qquad\qquad \mathbf{with}\ \lambda(q, (\_, s, \_)).\mathit{isQ}_{q_2}\ q\ \&\&\ \mathit{isS}_{\sigma_2}\ s & \\
[\![(q_1, 0, q_2)]\!] & = & (\mathsf{Q}_{q_1}, t)^\bullet \to (\mathsf{Q}_{q_2}{}^\bullet, t)^\bullet & \mathbf{with}\ \lambda(q, \_).\mathit{isQ}_{q_2}\ q \\
[\![(q_1, 1, q_2)]\!] & = & (\mathsf{Q}_{q_1}, t)^\bullet \to (\mathsf{Q}_{q_2}{}^\bullet, \mathit{moveR}\ t)^\bullet & \mathbf{with}\ \lambda(q, \_).\mathit{isQ}_{q_2}\ q \\
[\![(q_1, -1, q_2)]\!] & = & (\mathsf{Q}_{q_1}, t)^\bullet \to (\mathsf{Q}_{q_2}{}^\bullet, \mathit{invert}\ \mathit{moveR}\ t)^\bullet & \mathbf{with}\ \lambda(q, \_).\mathit{isQ}_{q_2}\ q
\end{array}
$$

Here, $\mathit{isQ}_q : \mathsf{T}_Q \to \mathsf{Bool}$ is a function that returns True for Q_q and False otherwise, and $\mathit{isS}_\sigma : \mathsf{T}_\Sigma \to \mathsf{Bool}$ is similar but defined for symbols. Notice that, by the reversibility of the Turing machine, patterns are non-overlapping 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 (Nishida and Vidal 2011; Mogensen 2006; Matsuda et al. 2010; Glück and Kawabe 2004). Roughly speaking, for a tail recursion (such as $g\ x = \textbf{case}\ x\ \textbf{of}\{p \to g\ e;\ p' \to e'\}$), **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 $\lambda^{\mathrm{PI}}_{\to}$) is useful here, as the effort can be made once for all. Specifically, we prepare the following higher-order function implementing general loops.

$$
\begin{array}{l}
\mathit{trace} : (a^\bullet \multimap (a \otimes a)^\bullet) \to (b^\bullet \multimap (b \otimes b)^\bullet) \to ((a \oplus x)^\bullet \multimap (b \oplus x)^\bullet) \to a^\bullet \multimap b^\bullet \\
\mathit{trace}\ \mathit{dupA}\ \mathit{dupB}\ h\ a = \textbf{let}\ (a_1, a_2)^\bullet = \mathit{dupA}\ a\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ (b_1, n)^\bullet = \mathit{go}\ (h\ (\mathsf{InL}^\bullet\ a_1))\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ (\mathsf{InL}\ b_2)^\bullet = h\ (\mathit{goN}\ a_2\ n)\ \textbf{in} \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{invert}\ \mathit{dupB}\ (b_1, b_2)^\bullet \\
\quad \textbf{where}\ \mathit{go} : (b \oplus x)^\bullet \multimap (b \otimes \mathsf{Nat})^\bullet \\
\qquad\qquad \mathit{go}\ (\mathsf{InL}\ b)^\bullet\ = (b, \mathsf{Z}^\bullet)^\bullet\ \textbf{with}\ \mathit{isZ} \circ \mathit{snd} \\
\qquad\qquad \mathit{go}\ (\mathsf{InR}\ x)^\bullet = \textbf{let}\ (b, n)^\bullet = \mathit{go}\ (h\ (\mathsf{InR}^\bullet\ x))\ \textbf{in}\ (b, \mathsf{S}^\bullet\ n)^\bullet \\
\qquad\qquad \mathit{goN} : a^\bullet \multimap \mathsf{Nat}^\bullet \multimap (a \oplus x)^\bullet \\
\qquad\qquad \mathit{goN}\ a\ \mathsf{Z}^\bullet \qquad = \mathsf{InL}^\bullet\ a\ \textbf{with}\ \mathit{isInL} \\
\qquad\qquad \mathit{goN}\ a\ (\mathsf{S}\ n)^\bullet = \textbf{let}\ (\mathsf{InR}\ x)^\bullet = h\ (\mathit{goN}\ a\ n)\ \textbf{in}\ \mathsf{InR}^\bullet\ x
\end{array}
$$

The $\mathit{trace}\ \mathit{dupA}\ \mathit{dupB}\ h\ a$ applies the forward/backward computation of h repeatedly to $\mathsf{InL}\ a$; it returns b if h returns $\mathsf{InL}\ b$, and otherwise (if h returns $\mathsf{InR}\ x$) it applies the same computation again for $h\ (\mathsf{InR}\ x)$. Here, dupA and dupB are supposed to be the reversible duplication (Glück and Kawabe 2003). This implementation essentially uses Yokoyama et al. (2012)'s optimized version of Bennett (1973)'s encoding. That is, if we have an injective $f : A \multimap B$ of which invertibility is made evident (i.e., locally reversible) by outputting and consuming the *same* trace (or, history (Bennett 1973)) of type H as $f_1 : A \multimap B \otimes H$ and $f_2 : A \otimes H \multimap B$, respectively, then we can implement the version $f' : A \multimap B$ of which invertibility is evident by (1) copying the input a as (a_1, a_2), (2) applying f_1 to a_1 to obtain (b_1, h), (3) applying f_2 to a_2 and h to obtain b_2, and (4) applying the inverse of copying (i.e., equivalence check (Glück and Kawabe 2003)) to (b_1, b_2) to obtain $b\ (= b_1 = b_2)$. Note that the roles of f_1 and f_2 are swapped in the backward execution. Above, we use loop counts as the trace H, and go and goN correspond to f_1 and f_2, respectively. The construction implies that the inverse of copying must always succeeds, and thus we can safely replace dupA by unsafe copying $\lambda a.\textbf{pin}\ a\ \mathit{unsafeNew}$ and dupB by $\mathit{invert}\ (\lambda b.\textbf{pin}\ b\ \mathit{unsafeNew})$. The version presented in the main body of this paper assumes this optimization.

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

$$
\begin{array}{l}
\mathit{rtm} : \mathsf{Tape}^\bullet \multimap \mathsf{Tape}^\bullet \\
\mathit{rtm} = \mathit{trace}\ \mathit{dupTape}\ \mathit{dupTape}\ (\mathit{checkFinal} \circ \mathit{step} \circ \mathit{assertInit}) \\
\mathit{assertInit} : (\mathsf{Tape} \oplus (\mathsf{T}_Q \otimes \mathsf{Tape}))^\bullet \to (\mathsf{T}_Q \otimes \mathsf{Tape})^\bullet \\
\mathit{assertInit}\ (\mathsf{InL}\ t)^\bullet \qquad = (\mathsf{Q}_{q_0}{}^\bullet, t)^\bullet\ \textbf{with}\ \mathit{isQ}_{q_0} \circ \mathit{fst} \\
\mathit{assertInit}\ (\mathsf{InR}\ (q, t))^\bullet = (q, t)^\bullet \\
\mathit{checkFinal} : (\mathsf{T}_Q \otimes \mathsf{Tape})^\bullet \to (\mathsf{Tape} \oplus (\mathsf{T}_Q \otimes \mathsf{Tape}))^\bullet
\end{array}
$$

$$\begin{aligned}
&\mathit{checkFinal}\ (\mathsf{Q}_{q_f}, t)^\bullet = \mathsf{InL}^\bullet\ t\ \textbf{with}\ \mathit{isL} \\
&\mathit{checkFinal}\ (q, t)^\bullet \quad\ = \mathsf{InR}^\bullet\ (q, t)^\bullet
\end{aligned}$$

Here, $\mathit{dupTape} : \mathsf{Tape}^\bullet \multimap (\mathsf{Tape} \otimes \mathsf{Tape})^\bullet$ is the reversible duplication of tapes. Recall that q_0 cannot be the destination of a transition and q_f cannot be the source. Note that, thanks to *trace*, the above definition of *rtm* is more straightforward than Yokoyama et al. (2008) in which *rtm* is defined by forward and backward simulations of a reversible Turing machine with step counting.