An Injective Language for Reversible Computation

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi

Department of Information Engineering
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
{scm,hu,takeichi}@ipl.t.u-tokyo.ac.jp

Abstract. Erasure of information incurs an increase in entropy and dissipates heat. Therefore, information-preserving computation is essential for constructing computers that use energy more effectively. A more recent motivation to understand reversible transformations also comes from the design of editors where editing actions on a view need to be reflected back to the source data. In this paper we present a point-free functional language, with a relational semantics, in which the programmer is allowed to define injective functions only. Non-injective functions can be transformed into a program returning a history. The language is presented with many examples, and its relationship with Bennett's reversible Turing machine is explained. The language serves as a good model for program construction and reasoning for reversible computers, and hopefully for modelling bi-directional updating in an editor.

1 Introduction

The interest in reversible computation arose from the wish to build computers dissipating less heat. In his paper in 1961, Landauer [17] noted that it is not the computation, but the erasure of information, that generates an increase in entropy and thus dissipates heat. Since then, various models of computation that do not erase information, thus capable of reversely construct the input from the output, have been proposed. Lecerf [18] and Bennett [4] independently developed their reversible Turing machines. Toffoli [26] proposed an information preserving logic, in which traditional logic can be embedded. Fredkin and Toffoli [11] then presented their "ballistic computer", which dramatically diverts from typical computers and instead resembles movement of particles, yet computationally equivalently to reversible Turing machines. Recent research actually attempts to build VLSI chips that do not erase information [27]. Due to the interest in quantum computing, reversible computation has recently attracted a wide range of researchers [24].

As was pointed out by Baker [3], were it only a problem in the hardware, we could compile ordinary programs for reversible computers and hide the reality from the programmers. Yet it is actually the highest level of computation where the loss of information is the most difficult to deal with. It is thus desirable to have programming languages that have more structure — a language designed for reversible computation.

Our interest in reversible computation came from yet another area. In our Programmable Structured Documents project [25], we are developing a structural editor/viewer for XML documents with embedded computations. The source document is transformed into a *view* in ways defined by the document designer. When a user edits the view, the changes need to be reflected back to the original source. It is thus preferable that the transformation be programmed in a language where only reversible transformation is allowed. When non-reversible computation is needed, it should be made explicit what extra information needs to be remembered to perform the view-to-source transformation. Dependency among different parts of the view shall also be made explicit so the system knows how to maintain consistency when parts of the view are edited.

In this paper we will present a point-free functional language in which all functions definable are injective. Previous research was devoted to the design of models and languages for reversible computation [26, 4, 5, 19, 3, 10, 28]. Several features distinguish our work from previous results. Since it is a language in which we want to specify XML transformations, we prefer a high-level, possibly functional, programming language. It has a relational semantics, thus all programs have a relational interpretation. While some previous works focus more on the computational aspect, our language serves as a good model for program construction, derivation and reasoning of reversible programs. We have another target application in mind: to extend the language to model the bi-directional updating problem in an editor, studied by [21, 14]. We hope that the relational semantics can shed new light on some of the difficulties in this field.

In the next three sections to follow, we introduce the basic concepts of relations, a point-free functional language, and finally our injective language Inv, each of which is a refinement of the previous one. Some examples of useful injective functions defined in Inv are given in Section 5. We describe how non-injective functions can be "compiled" into Inv functions in Section 6, where we also discuss the relationship with Bennett's reversible Turing machine. Some implementation issues are discussed in Section 7.

2 Relations

Since around the 80's, the program derivation community started to realise that there are certain advantages for a theory of functional programming to base on relations [2, 6]. More recent research also argued that the relation is a suitable model to talk about program inversion because it is more symmetric [22]. In this section we will give a minimal introduction to relations.

A relation of type $A \to B$ is a set of pairs whose first component has type A and second component type B. When a pair (a, b) is a member of a relation R, we say that a is mapped to b by R. A (partial) function[1], under this interpretation, is a special case of a relation that is *simple* — a value in A is mapped to at most one value in B. That is, if $(a, b) \in R$ and $(a, b') \in R$, then $b = b'$. For example, the function $\mathit{fst} :: (A \times B) \to A$ extracting the first component of a pair, usually denoted pointwisely as $\mathit{fst}\,(a, b) = a$, is defined by the following set:

$$\mathit{fst} = \{((a, b), a) \mid a \in A \wedge b \in B\}$$

where (a, b) indeed uniquely determines a. The function $\mathit{snd} :: (A \times B) \to B$ is defined similarly.

The *domain* of a relation $R :: A \to B$ is the set $\{a \in A \mid \exists b \in B :: (a, b) \in R\}$. The *range* of R is defined symmetrically. The *converse* of a relation R, written R°, is obtained by swapping the pairs in R. That is,

$$(b, a) \in R^\circ \equiv (a, b) \in R$$

An injective function is one whose converse is also a function. In such cases a value in the domain uniquely defines its image in the range, and vice versa. The term *inverse of a function* is usually reserved to denote the converse of an injective

[1] For convenience, we refer to possibly partial functions when we say "functions". Other papers may adopt different conventions.

function. Given relations $R :: A \to B$ and $S :: B \to C$, their composition $R; S$ is defined by:

$$R; S = \{(a, c) \,|\, \exists b :: (a, b) \in R \wedge (b, c) \in S\}$$

The converse operator $^\circ$ distributes into composition contravariantly:

$$(R; S)^\circ = S^\circ; R^\circ$$

Given two relations R and S of the same type, one can take their union $R \cup S$ and intersection $R \cap S$. The union, when R and S have disjoint domains, usually corresponds to conditional branches in a programming language. The intersection is a very powerful mechanism for defining useful relations. For example, the function $dup\ a = (a, a)$, which duplicates its argument, can be defined by:

$$dup = fst^\circ \cap snd^\circ$$

Here the relation fst°, due to type constraints, has to have type $A \to (A \times A)$. It can therefore be written as the set $\{(a, (a, a')) \,|\, a \in A \wedge a' \in A\}$ — that is, given a, fst° maps it to (a, a') where a' is an arbitrary value in A. Similarly, snd° maps a to (a', a) for an arbitrary a'. The only point where they coincide is (a, a). That is, taking their intersection we get

$$dup = \{(a, (a, a)) \,|\, a \in A\}$$

which is what we expect.

The converse operator $^\circ$ distributes into union and intersection. If we take the converse of dup, we get:

$$dup^\circ = fst \cap snd$$

Given a pair, fst extracts its first component, while snd extracts the second. The intersection means that the results have to be equal. That is, dup° takes a pair and lets it go through only if the two components are equal. That explains the observation in [12] that to "undo" a duplication, we have to perform an equality test.

3 The Point-free Functional Language Fun

The intersection is a very powerful construct for specification — with it one can define undecidable specifications. In this section we will refine the relational constructs to a point-free functional language which is computationally equivalent to conventional programming languages we are familiar with. The syntax of Fun is defined by[2]:

$$\begin{aligned} F &::= C \mid C^\circ \\ &\quad \mid F; F \mid id \\ &\quad \mid \langle F, F \rangle \mid fst \mid snd \\ &\quad \mid F \cup F \\ &\quad \mid \mu(X\colon F_X) \\ C &::= nil \mid cons \mid zero \mid succ \end{aligned}$$

The base types of Fun are natural numbers, polymorphic lists, and $Unit$, the type containing only one element $()$. The function $nil :: B \to [A]$ is a constant function

[2] In Fun there are no primitive operators for equality or inequality check. They can be defined recursively on natural numbers, so we omitted it from the language to make it simpler. In Inv, however, we do include those checks as primitives since they are important for inversion.

always returning the empty list, while *cons* $:: (A \times [A]) \rightarrow [A]$ extends a list by the given element. Converses are applied to base constructors (denoted by the non-terminal C) only. We abuse the notation a bit by denoting the converse of all elements in C by C°, and by F_X we denote the union of F and the set of variable names X. The converse of *cons*, for example, decomposes a non-empty list into the head and the tail. The converse of *nil* matches only the empty list and maps it to anything. The result is usually thrown away. To avoid non-determinism in the language, we let the range type of nil° be *Unit*. Functions *zero* and *succ* are defined similarly. Presently we do not yet need a constructor for the type *Unit*.

The function *id* is the identity function, the unit of composition. Functions *fst* and *snd* extract the first and second components of a pair respectively. Those who are familiar with the "squiggle" style of program derivation would feel at home with the "split" construct, defined by:

$$\langle f, g \rangle\; a = (f\; a, g\; a)$$

The angle brackets on the left-hand side denote the split construct, while the parentheses on the right-hand side denote pairs. This definition, however, assumes that f and g be functional. Less well-known is its relational definition in terms of intersection:

$$\langle f, g \rangle = f; fst^{\circ}\ \cap\ g; snd^{\circ}$$

For example, the *dup* function in the previous section is defined by $\langle id, id \rangle$.

With *fst*, *snd* and the split we can define plenty of "piping functions" useful for point-free programming. For example, the function *swap* $:: (A \times B) \rightarrow (B \times A)$, swapping the components in a pair, and the function *assocr* $:: ((A \times B) \times C) \rightarrow (A \times (B \times C))$, are defined by:

$$\begin{aligned} swap &= \langle snd, fst \rangle \\ assocr &= \langle fst; fst, \langle fst; snd, snd \rangle \rangle \end{aligned}$$

The function *assocl* $:: (A \times (B \times C)) \rightarrow ((A \times B) \times C)$ can be defined similarly. The "product" functor $(f \times g)$, on the other hand, is defined by

$$(f \times g)\,(a, b) = (f\; a, g\; b)$$

Squigglists are more familiar with its point-free definition:

$$(f \times g) = \langle fst; f, snd; g \rangle$$

Union of functions is still defined as set union. To avoid non-determinism, however, we require in $f \cup g$ that f and g have disjoint domains. Arbitrary use of intersection, on the other hand, is restricted to its implicit occurrence in splits.

Finally, μF denotes the unique fixed-point of the Fun-valued function F, with which we can define recursive functions. The important issue whether a relation-valued function has an unique fixed-point shall not be overlooked. It was shown in [9] that the uniqueness of the fixed-point has close relationship with well-foundness and termination. All recursive definitions in this paper do have unique fixed-points, although it is out of the scope of this paper to verify them.

As an example, the concatenation of two cons-lists is usually defined recursively as below:

$$\begin{aligned} [\,] +\!\!+\, y &= y \\ (a : x) +\!\!+\, y &= a : (x +\!\!+\, y) \end{aligned}$$

Its curried variation, usually called *cat* $:: ([A] \times [A]) \rightarrow [A]$, can be written in point-free style in Fun as:

$$cat = \mu(X\colon (nil^{\circ} \times id); snd\ \cup\ (cons^{\circ} \times id); assocr; (id \times X); cons)$$

The two branches of $\cup$ correspond to the two clauses of $+\!\!+$, while $(nil^\circ \times id)$ and $(cons^\circ \times id)$ act as patterns. The term $(cons^\circ \times id)$ decomposes the first component of a pair into its head and tail, while $(nil^\circ \times id)$ checks whether that component is the empty list. The piping function *assocr* distributes the values to the right places before the recursive call.

We decided to make the language point-free because it is suitable for inversion — composition is simply run backward, and the need to use piping functions makes the control flow explicit. It is true, however, point-free programs are sometimes difficult to read. To aid understanding we will supply pointwise definition of complicated functions. The conversion between the point-free and pointwise style, however, will be dealt with loosely.

The language Fun is not closed under converse — we can define non-injective functions in Fun, such as *fst* and *snd*, whose converses are not functional. In other words, Fun is powerful enough that it allows the programmer to define functions "unhealthy" under converse. In the next section we will further refine Fun into an injective language.

4 The Injective Language Inv

In the previous section we defined a functional language Fun with a relational semantics. All constructs of Fun have relational interpretations and can thus be embedded in relations. In this section, we define a functional language Inv that allows only injective functions. All its constructs can be embedded in, and therefore Inv is strictly a subset of, Fun.

The problematic constructs in Fun include constant functions, *fst*, *snd*, and the split. Constant functions and projections lose information. The split duplicates information and, as a result, in inversion we need to take care of consistency of previously copied data. We wish to enforce constrained use of these problematic constructs by introducing more structured constructs in Inv, in pretty much the same spirit how we enforced constrained use of intersection by introducing the split in Fun. The language Inv is defined by:

$$\begin{aligned}
I \;&::= I^\circ \mid C \\
&\mid eq\,P \mid dup\,P \mid neq\,S\,S \\
&\mid I;I \mid id \\
&\mid (I \times I) \mid assocr \mid swap \\
&\mid (I \cup I) \\
&\mid \mu(X\colon I_X) \\
C \;&::= succ \mid cons \\
P \;&::= nil \mid zero \mid S \\
S \;&::= C^\circ \mid fst \mid snd \mid id \mid S;S
\end{aligned}$$

Each construct in Inv has its inverse in Inv. Constructors *cons* and *succ* have inverses $cons^\circ$ and $succ^\circ$. The function *swap*, now a primitive, is its own inverse. That is, $swap^\circ = swap$. The function *assocr* has inverse *assocl*, whose definition will be given later. The inverse operator promotes into composition, product, union and fixed-point operator by the following rules:

$$\begin{aligned}
(f;g)^\circ &= g^\circ;f^\circ \\
(f \times g)^\circ &= (f^\circ \times g^\circ) \\
(f \cup g)^\circ &= f^\circ \cup g^\circ \\
(\mu F)^\circ &= \mu(^\circ;F;^\circ)
\end{aligned}$$

In the last equation F is a function from Inv expressions to Inv expressions, and the composition ; is lifted. One might instead write $\mu(\lambda X \cdot (F\,X^\circ)^\circ)$ as the right-hand side. An extra restriction needs to be imposed on union. To preserve reversibility, in $f \cup g$ we require not only the domains, but the ranges of f and g, to be disjoint. The disjointness may be checked by a type system, but we have not explored this possibility.

The most interesting is the *dup*/*eq* pair of operators. Each of them takes an extra functional argument which is either *id*, a constant function, or a sequence of composition of *fst*s, *snd*s or constructors. They have types:

$$\begin{aligned} &dup :: (\mathsf{F}a \to a) \to \mathsf{F}a \to (\mathsf{F}a \times a) \\ &eq \;\;\, :: (\mathsf{F}a \to a) \to (\mathsf{F}a \times a) \to \mathsf{F}a \end{aligned}$$

where F is some type functor. A call $eq\, f\, (x, a)$ tests whether the field in x selected by f equals a. Conversely, $dup\, f\, x$ copies the selected field in x. They can be understood informally as

$$\begin{aligned} dup\, f\, x &= (x, f\, x) \\ eq\, f\, (x, a) &= x \equiv f\, x = a \end{aligned}$$

That they are inverses of each other can be seen from their relational definition:

$$\begin{aligned} dup\, f &= fst^{\circ} \;\cap\; f; snd^{\circ} \\ eq\, f &= fst \;\cap\; snd; f^{\circ} \end{aligned}$$

The definition of $dup\, f$ is similar to that in Section 2 apart from the presence of the argument f. Given the definitions it is clear that $(eq\, f)^{\circ} = dup\, f$ and vice versa.

According to the syntax, constant functions *zero* and *nil* can appear only as arguments to *dup*. For example, to introduce a fresh zero one has to call *dup zero*, which takes an input a and returns the pair $(a, 0)$. Therefore it is guaranteed that the input is not lost. An alternative design is to restrict the domain of *zero* and *nil* to the type *Unit* (therefore they do not lose information), while introducing a variation of the *dup* construct that creates fresh *Unit* values only. Considering their relational semantics, the two designs are interchangeable. For this paper we will mention only the first approach.

Some more words on the design decision of the *dup*/*eq* operators. Certainly the extra argument, if restricted to *fst*, *snd* and constructors, is only a syntactic sugar. We can always swap outside the element to be duplicated and use, for example $(id \times dup\; id)$. Nevertheless we find it quite convenient to have this argument. The natural extension to include constant functions unifies the two problematic elements for inversion, duplication and the constant function, into one language construct. For our future application about bi-directional editing, we further allow the argument to be any possibly non-injective functions (such as *sum*, *map fst*, etc), which turns out to be useful for specifying transformations from source to view. Allowing *dup*/*eq* to take two arguments, however, does not seem to be necessary, as *eq* is meant to be asymmetrical — after a successful equality check, *one* of the checked values has to go away. It is in contrast to the *neq* operator to be introduced below.

The $neq\, p_1\, p_2$ operator, where p_1 and p_2 are projections defined by *fst* and *snd*, is a partial function checking for inequality. It is defined by

$$neq\, p_1\, p_2\, (x, y) = (x, y) \equiv p_1\, x \neq p_2\, y$$

Otherwise (x, y) is not in its domain. The $neq\, p_1\, p_2$ operator is its own inverse. It is sometimes necessary for ensuring the disjointness of the two branches of a union.

Some more operators will be introduced in sections to come to deal with the sum type, trees, etc[3]. For now, these basic operators are enough for our purpose.

[3] Of course, Fun can be extended in the same way so these new operators can still be embedded in Fun.

Apparently every program in Inv is invertible, since no information is lost in any operation. Every operation has its inverse in Inv. The question, then, is can we actually define useful functions in this quite restrictive-looking language?

5 Examples of Injective Functions in Inv

In this section we give some examples of injective functions expressed in Inv.

5.1 Piping Functions

We loosely define "piping function" as functions that move around objects in pairs, copy them, or discard them, without checking their values – that is, "natural" functions on pairs. We choose not to include *assocl* as a primitive because it can be defined in terms of other primitives — in several ways, in fact. One is as below:

$$assocl = swap; (swap \times id); assocr; (id \times swap); swap$$

Another is:

$$assocl = swap; assocr; swap; assocr; swap$$

Alternatively, one might wish to make *assocl* a primitive, so that inverting *assocr* does not increase the size of a program.

These piping functions will turn out to be useful:

$$\begin{aligned} subr\,(a,(b,c)) &= (b,(a,c)) \\ trans\,((a,b),(c,d)) &= ((a,c),(b,d)) \\ distr\,(a,(b,c)) &= ((a,b),(a,c)) \end{aligned}$$

The function *subr* substitutes the first component in a pair to the right, *trans* transposes a pair of pairs, while *distr* distributes a value into a pair. They have point-free definitions in Inv, shown below[4]:

$$\begin{aligned} subr &= assocl; (swap \times id); assocr \\ trans &= assocr; (id \times subr); assocl \\ distr &= (dup\,id \times id); trans \end{aligned}$$

From the definitions it is immediate that *subr* and *trans* are their own inverses. The function *distr*, on the other hand, makes use of *dup id* to duplicate a before distribution. Its inverse, $trans; (eq\,id \times id)$ thus has to perform an equality check before joining the two as into one.

In Section 6.1 we will talk about automatic construction of piping functions.

5.2 Patterns

Patterns in Fun are written in terms of products, nil°, and $cons^\circ$. For example, $(cons^\circ \times id)$ decomposes the first component of a pair, while $(nil^\circ \times id)$ checks whether that component is the empty list. The latter is usually followed by a *snd* function to throw away the unit resulting from nil°.

The term $(cons^\circ \times id)$ is still a legal and useful pattern in Inv. However, we do not have nil° in Inv. Part of the reason is that we do not want to have to introduce *snd* into the language, which allows the programmer to throw arbitrary information away. Instead, to match the pattern $(x, [\,])$, we write *eq nil*, which throws $[\,]$ away and keeps x. It is $(id \times nil^\circ); fst$ packaged into one function. On the other hand, its inverse *dup nil* introduces a fresh empty list.

Similarly, $swap; eq\,nil$ is equivalent to $(nil^\circ \times id); snd$. We define

$$nl = swap; eq\,nil$$

because we will use it later.

[4] Another definition, $subr = swap; assocr; (id \times swap)$, is shorter if *assocl* is not a primitive.

5.3 Snoc, or Tail-cons

The function $wrap :: A \rightarrow [A]$, wrapping the input into a singleton list, can be defined in Fun as $wrap = \langle id, nil \rangle; cons$. It also has a definition in Inv:

$$wrap = dup\ nil; cons$$

Its converse is therefore $wrap^\circ = cons^\circ; eq\ nil$ – the input list is deconstructed and a nullity test is performed on the tail.

The function $snoc :: ([A], A) \rightarrow [A]$ appends an element to the right-end of a list. With $wrap$, we can define $snoc$ in Fun as

$$\begin{aligned} snoc = \mu(X\colon & (nil^\circ \times id); snd; wrap \cup \\ & (cons^\circ \times id); assocr; (id \times X); cons) \end{aligned}$$

To define it in Inv, we notice that $(nil^\circ \times id); snd$ is exactly nl defined above. Therefore we can rewrite $snoc$ as:

$$\begin{aligned} snoc = \mu(X\colon & nl; wrap \cup \\ & (cons^\circ \times id); assocr; (id \times X); cons) \end{aligned}$$

Its inverse $snoc^\circ :: [A] \rightarrow ([A], A)$ extracts the last element from a list, if the list is non-empty. The first branch of $snoc^\circ$, namely $wrap^\circ; nl$, extracts the only element from a singleton list, and pairs it with an empty list. The second branch, $cons^\circ; (id \times snoc^\circ); assocl; (cons \times id)$, deconstructs the input list, processes the tail with $snoc^\circ$, before assembling the result. The two branches have disjoint domains because the former takes only singleton lists while the second, due to the fact that $snoc^\circ$ takes only non-empty lists, accepts only lists with two or more elements.

5.4 Mirroring

Consider the function $mirror :: [A] \rightarrow [A]$ which takes a list and returns its mirror, for example, $mirror\,[1, 2, 3] = [1, 2, 3, 3, 2, 1]$. Assume that $snoc$ and its inverse exists. Its definition in Fun is given as below:

$$\begin{aligned} mirror = \mu(X\colon & nil^\circ; nil \cup \\ & cons^\circ; \langle fst, \langle snd, fst \rangle\rangle; (id \times (X \times id); snoc); cons) \end{aligned}$$

To rewrite $mirror$ in Inv, the tricky part is to convert $\langle fst, \langle snd, fst \rangle\rangle$ into something equivalent in Inv. It turns out that $\langle fst, \langle snd, fst \rangle\rangle = dup\,fst; assocr$. Also, since nil° is not available in Inv, we have to rewrite $nil^\circ; nil$ as $dup\ id; eq\ nil$. The function $dup\ id$ duplicates the input, before $eq\ nil$ checks whether the input is the empty list and eliminates the duplicated copy if the check succeeds. It will be discussed in Section 6.1 how such conversion can be done automatically. As a result we get:

$$\begin{aligned} mirror = \mu(X\colon & dup\ id; eq\ nil \cup \\ & cons^\circ; dup\,fst; assocr; (id \times (X \times id); snoc); cons) \end{aligned}$$

It is educational to look at its inverse. By distributing the converse operator inside, we get:

$$\begin{aligned} mirror^\circ = \mu(X\colon & dup\ nil; eq\ id \cup \\ & cons^\circ; (id \times snoc^\circ; (X \times id)); assocl; eq\,fst; cons) \end{aligned}$$

Note that $dup\,fst$ is inverted to an equality test $eq\,fst$. In the second branch, $cons^\circ; (id \times snoc^\circ; (X \times id))$ decomposes the given list into the head, the last element, and the list in-between. A recursive call then processes the list, before $assocl; eq\,fst$ checks that the first and the last elements are equal. The whole expression fails if the check fails, thus $mirror^\circ$ is a partial function. It is desirable to perform the equality check *before* making the recursive call. One can show, via algebraic reasoning, that the second branch equals

$$cons^{\circ}; (id \times snoc^{\circ}); assocl; eq\,fst; (id \times X); cons$$

which performs the check and rejects the illegal list earlier. This is one example showing that program inversion, even in this compositional style, is more than "running it backwards". To construct a program with desirable behaviour it takes some more transformations — some are easier to be performed mechanically than others.

5.5 Labelled Concatenation and Double Concatenation

List concatenation is not injective. However, the following function *lcat* (labelled concatenation)

$$lcat\,(a, (x, y)) = (a, x +\!\!+ [a] +\!\!+ y)$$

is injective if its domain is restricted to tuples where a does not appear in x. This way a acts as a marker telling us where to split $x +\!\!+ y$ into two. Its point-free definition can be written as:

$$lcat = nmem\,fst; \langle fst, subr; (id \times cons); cat\rangle$$

where $nmem\,p\,(a, x) = (a, x)$ if a is not a member of the list $p\,x$. To show that it is injective, it is sufficient to show an alternative definition of *lcat* in Inv:

$$\begin{aligned} lcat = \mu(X\colon\ & (dup\,id \times nl); assocr; (id \times cons) \cup \\ & (id \times (cons^{\circ} \times id); assocr); \\ & \ neq\,id\,fst; subr; (id \times X); subr; (id \times cons)) \end{aligned}$$

It takes a tedious but routine inductive proof, given in the appendix, to show that the two definitions are indeed equivalent. To aid understanding, the reader can translate it to its corresponding pointwise definition:

$$\begin{aligned} &lcat\,(a, ([\,], y)) &&= (a, a : y) \\ &lcat\,(a, (b : x, y)) &&= \textbf{let}\,(a', xy) = lcat\,(a, (x, y)) \\ & &&\quad \textbf{in}\,(a', b : xy) \qquad\qquad \text{if } a \neq b \end{aligned}$$

The presence of *neq* is necessary to guarantee the disjointness of the two branches — the first branch returns a list *starting with a*, while the second branch returns a list whose head is *not a*.

Similarly, the following function *dcat* (for "double" concatenation)

$$dcat\,(a, ((x, y), (u, v))) = (a, (x +\!\!+ y, u +\!\!+ [a] +\!\!+ v))$$

is injective if its domain is restricted to tuples where a does not appear in u, and x and u are equally long. Its point-free definition can be written as:

$$dcat = pred; distr; ((id \times cat) \times subr; (id \times cons); cat)$$

where *pred* is the predicate true of $(a, ((x, y), (u, v)))$ where a is not in u and $length\,x = length\,u$. To show that it is injective, it is sufficient to show an alternative definition of *dcat* in Inv:

$$\begin{aligned} dcat = \mu(X\colon\ & (dup\,id \times (nl \times nl)); trans; (id \times cons); assocr \cup \\ & (id \times ((cons^{\circ} \times id) \times (cons^{\circ} \times id))); neq\,id\,(snd; fst); \\ & \ pi; (id \times X); subr; (id \times trans; (cons \times cons))) \end{aligned}$$

where *pi* is a piping function defined by

$$pi = (id \times (assocr \times assocr); trans); subr$$

such that $pi\,(a, (((b, x), y), ((c, u), v))) = ((b, c), (a, ((x, y), (u, v))))$. The proof is similar to the proof for *lcat* and need not be spelt out in detail. To aid understanding, the above *dcat* is actually equivalent to the following pointwise definition:

$$\begin{aligned}
&dcat\,(a, (([\,], y), ([\,], v))) &&= (a, (y, a : v))\\
&dcat\,(a, ((b : x, y), (c : u, v))) &&= \mathbf{let}\ (a', (xy, uv)) \ = \ dcat\,(a, ((x, y), (u, v)))\\
& && \quad\ \mathbf{in}\ (a', (b : xy, c : uv))
\end{aligned}$$

5.6 Printing and Parsing XML Trees

Consider internally labelled binary trees:

$$\mathbf{data}\ Tree\ A = Null \mid Node\ A\ (Tree\ A)\ (Tree\ A)$$

To deal with trees, we extend Inv with a new primitive $node :: (A \times (Tree\ A \times Tree\ A)) \rightarrow Tree\ A$, the curried variation of *Node*, and extend *P* with a new constant function *null*. An XML tree is basically a rose tree – a tree where each node has a list of children. A *forest* of rose trees, however, can be represented as a binary tree by the child-sibling representation: the left child of the a node in the binary tree represents the leftmost child of the corresponding XML node, while the right child represents its next sibling. For example, the XML tree in Figure 1(a) can be represented as the binary tree in Figure 1(b).

```
<a>
  <b> <c></c>
      <d></d>
  </b>
  <e></e>
</a>
```

(a)

```
Node a (Node b (Node c Null
                        (Node d Null Null))
               (Node e Null Null))
      Null
```

(b)

Fig. 1. Child-sibling representation of an XML tree

To print a binary tree to its conventional serial representation, on the other hand, one has to print an opening tag (for example `<b>`), its left subtree (`<c></c><d></d>`), a closing tag (`</b>`), and then print its right subtree (`<e></e>`). That is similar to what *lcat* does! As a simplification, we define:

$$\begin{aligned}
&serialise :: Tree\ A \rightarrow [A]\\
&serialise = \mu(X\colon dup\ nil; swap; eq\ null \cup\\
&\qquad\qquad node^{\circ}; (id \times (X \times X)); lcat; cons)
\end{aligned}$$

The function *serialise* takes a binary tree whose values in each node do not occur in the right subtree, and flattens the tree into a list. To deal with XML trees in general, we will have to return a list of opening/closing tags and check, in *lcat*, that the tags are balanced. To perform the check, we have to maintain a stack of tags. For demonstration purpose, we deal with only the simpler case here. Its inverse, $serialise^{\circ}$, parses a stream of labels back to an XML tree. By implementing printing, we get parsing for free!

However, $serialise^{\circ}$ is a quadratic-time parsing algorithm. The reason is that *serialise*, due to repeated calls to *lcat*, is quadratic too, and the inverted program, without further optimising transformation, always has the same efficiency as the original one. To construct a linear-time parsing algorithm, we can try to construct a linear version of *serialise*. Alternatively, we can make use of well-known program derivation techniques to construct an algorithm performing $serialise^\circ$ in linear time. We define the function *pparse* (partial-parse) as below:

$$\begin{aligned}&pparse\,(a, x) = (a, (serialise^\circ\; y, z))\\&\quad \textbf{where}\;\; y +\!\!+ [a] +\!\!+ z = x\end{aligned}$$

In point-free style it is written

$$pparse = lcat^\circ; (id \times (serialise^\circ \times id))$$

To come up with a recursive definition, we notice that the identity function id, when instantiated to take value (a, x) of type $(A \times [A])$, can be factored into $id = (id \times dup\;nil; eq\;id) \cup (id \times cons^\circ; cons)$, corresponds to the case x being empty or non-empty respectively. We start derivation with $pparse = id; pparse$ and deal with each case separately. It will turn out that we need to further split the second case into two:

$$\begin{aligned}id = {}&(id \times dup\;nil; eq\;id) \cup\\&(id \times cons^\circ); swap; eq\;fst; dup\;fst; (cons \times id); swap \cup\\&(id \times cons^\circ; cons); neq\;id\;(cons; fst);\end{aligned}$$

When x is non-empty, the head of x may equal a or not. We prefix *pparse* with each of the cases, and try to simplify it using algebraic rules. It is basically a case-analysis in point-free style. Some of the branches may turn out to yield an empty relation.

We will demonstrate only the third branch. The derivation relies on the following associativity property:

$$a : ((b : (x +\!\!+ [b] +\!\!+ y)) +\!\!+ [a] +\!\!+ z) = a : (b : x +\!\!+ [b] +\!\!+ (y +\!\!+ [a] +\!\!+ z))$$

The point-free counterpart of the property, however, looks much more cumbersome. We define:

$$\begin{aligned}subrr &= subr; (id \times subr)\\subassoc &= (id \times assocr; (id \times assocr)); subrr\end{aligned}$$

The associativity property can be rewritten as:

$$\begin{aligned}&(id \times (lcat; cons \times id)); lcat\\&= subassoc; (id \times (id \times lcat)); subrr^\circ; (id \times lcat; cons); neq\;id\;(cons^\circ; fst) \qquad (1)\end{aligned}$$

The *neq* check needs to be there; otherwise only the inclusion $\subseteq$ holds. Given (1), derivation of *pparse* is long but trivial:

$$\begin{aligned}&\quad neq\;id\;(cons^\circ; fst); pparse\\&= \quad \{\text{definition of } pparse\}\\&\quad neq\;id\;(cons^\circ; fst); lcat^\circ; (id \times (serialise^\circ \times id))\\&\supseteq \quad \{\text{definition of } serialise\text{; we try its branches separately}\}\\&\quad neq\;id\;(cons^\circ; fst); lcat^\circ;\\&\quad (id \times (cons^\circ; lcat^\circ; (id \times (serialise^\circ \times serialise^\circ)); node \times id))\\&= \quad \{\text{products}\}\\&\quad neq\;id\;(cons^\circ; fst); lcat^\circ; (id \times (cons^\circ; lcat^\circ \times id));\\&\quad (id \times ((id \times (serialise^\circ \times serialise^\circ)); node \times id))\\&= \quad \{\text{by (1) and } neq\,f\;g; neq\,f\;g = neq\,f\;g\}\end{aligned}$$

$$
\begin{aligned}
& \mathit{neq}\ \mathit{id}\ (\mathit{cons}^\circ; \mathit{fst}); (\mathit{id} \times \mathit{cons}^\circ; \mathit{lcat}^\circ); \mathit{subrr}; (\mathit{id} \times (\mathit{id} \times \mathit{lcat}^\circ)); \mathit{subassoc}^\circ; \\
& (\mathit{id} \times ((\mathit{id} \times (\mathit{serialise}^\circ \times \mathit{serialise}^\circ)); \mathit{node} \times \mathit{id})) \\
= \quad & \{\text{naturalty of } \mathit{subassoc}\} \\
& \mathit{neq}\ \mathit{id}\ (\mathit{cons}^\circ; \mathit{fst}); (\mathit{id} \times \mathit{cons}^\circ; \mathit{lcat}^\circ); \mathit{subrr}; (\mathit{id} \times \\
& (\mathit{serialise}^\circ \times \mathit{lcat}^\circ; (\mathit{id} \times (\mathit{serialise}^\circ \times \mathit{id})))); \mathit{subassoc}^\circ; (\mathit{id} \times (\mathit{node} \times \mathit{id})) \\
= \quad & \{\text{definition of } \mathit{pparse}\} \\
& \mathit{neq}\ \mathit{id}\ (\mathit{cons}^\circ; \mathit{fst}); (\mathit{id} \times \mathit{cons}^\circ; \mathit{lcat}^\circ); \mathit{subrr}; \\
& (\mathit{id} \times (\mathit{serialise}^\circ \times \mathit{pparse})); \mathit{subassoc}^\circ; (\mathit{id} \times (\mathit{node} \times \mathit{id})) \\
= \quad & \{\text{naturalty of } \mathit{subrr}\} \\
& \mathit{neq}\ \mathit{id}\ (\mathit{cons}^\circ; \mathit{fst}); (\mathit{id} \times \mathit{cons}^\circ; \mathit{lcat}^\circ; (\mathit{id} \times (\mathit{serialise}^\circ \times \mathit{id}))); \mathit{subrr}; \\
& (\mathit{id} \times (\mathit{id} \times \mathit{pparse})); \mathit{subassoc}^\circ; (\mathit{id} \times (\mathit{node} \times \mathit{id})) \\
= \quad & \{\text{definition of } \mathit{pparse}\} \\
& \mathit{neq}\ \mathit{id}\ (\mathit{cons}^\circ; \mathit{fst}); (\mathit{id} \times \mathit{cons}^\circ; \mathit{pparse}); \mathit{subrr}; \\
& (\mathit{id} \times (\mathit{id} \times \mathit{pparse})); \mathit{subassoc}^\circ; (\mathit{id} \times (\mathit{node} \times \mathit{id})) \\
= \quad & \{\text{since } \mathit{neq}\, f\ (\mathit{cons}^\circ; g); (\mathit{id} \times \mathit{cons}^\circ) = (\mathit{id} \times \mathit{cons}^\circ); \mathit{neq}\, f\ g\} \\
& (\mathit{id} \times \mathit{cons}^\circ); \mathit{neq}\ \mathit{id}\, \mathit{fst}; (\mathit{id} \times \mathit{pparse}); \mathit{subrr}; \\
& (\mathit{id} \times (\mathit{id} \times \mathit{pparse})); \mathit{subassoc}^\circ; (\mathit{id} \times (\mathit{node} \times \mathit{id}))
\end{aligned}
$$

After some derivation, and a careful check that the recursive equation does yield unique fixed-point (see [9]), one will come up with the following definition of *pparse*:

$$
\begin{aligned}
\mathit{pparse} = \mu(X\colon & (\mathit{id} \times \mathit{cons}^\circ); \\
& (\mathit{swap}; \mathit{eq}\, \mathit{fst}; (\mathit{id} \times \mathit{dup}\ \mathit{null}; \mathit{swap}) \cup \\
& \mathit{neq}\ \mathit{id}\, \mathit{fst}; (\mathit{id} \times X); \mathit{subrr}; (\mathit{id} \times (\mathit{id} \times X)); \mathit{subassoc}^\circ; \\
& \quad (\mathit{id} \times (\mathit{node} \times \mathit{id}))))
\end{aligned}
$$

Now that we have *pparse*, we need to express $\mathit{serialise}^\circ$ in terms of *pparse*. Some derivation would show that:

$$
\begin{aligned}
\mathit{serialise}^\circ = \mu(X\colon & \mathit{dup}\ \mathit{null}; \mathit{swap}; \mathit{eq}\ \mathit{nil} \cup \\
& \mathit{cons}^\circ; \mathit{pparse}; (\mathit{id} \times (\mathit{id} \times X)); \mathit{node})
\end{aligned}
$$

The point-free definition of *pparse* and $\mathit{serialise}^\circ$ might be rather confusing to the reader. To aid understanding, their pointwise definition is given in Figure 2.

```
pparse (a, a : x) = (a, (Null, x))          serialise° []        = Null
pparse (a, b : x) =                         serialise° (a : x) =
   let  (b, (t, y))  = pparse (b, x)           let  (a, (t, y)) = pparse (a, x)
        (a, (u, z)) = pparse (a, y)                  u             = serialise° y
   in   (a, (Node b t u, z))                   in   Node a t u
```

Fig. 2. Pointwise definition of *pparse* and $\mathit{serialise}^\circ$.

5.7 Loops

An important feature is still missing in Inv — we can not define loops. Loops come handy when we want to show that Inv is computationally as powerful as Bennett's reversible Turing machine [4], since the simulation of a Turing machine is best described as a loop. In Fun, one can write a loop as a tail recursive function $\mu(X\colon \mathit{term} \cup \mathit{body}; X)$ where *term* and *body* have disjoint domains. However, the range of $\mathit{body}; X$ contains that of *term*, which is not allowed in Inv — when we ran the loop backwards we do not know whether to terminate the loop now or execute the body again.

Tail recursion is allowed in [13], where they resolve the non-determinism in a way similar to how left-recursive grammars are dealt with in LR parsing. Alternatively, we could introduce a special construct for loops, for example, $S; B^*; T$, where the initialisation S and loop body B have disjoint ranges, while B and the terminating condition T have disjoint domains. In [8, 9], the conditions for a loop to terminate, as well as guidelines for designing terminating loops, were discussed in a similar style. One of the earliest case study of inverting loops is [7]. Construction and reasoning of invertible loops in general has been discussed in [1].

Luckily, just to show that Inv is computationally equivalent to the reversible Turing machine, we do not need loops. One can code the reversible Turing machine as a function which returns the final state of the tapes together with an integer counting the number iterations executed. The count can then be eliminated in a clever way described by Bennett. More discussions will be given in Section 6.2.

6 Translating Non-injective Functions

Still, there are lots of things we cannot do in Inv. We cannot add two numbers, we cannot concatenate two lists. In short, we cannot construct non-injective functions. However, given a non-injective function $p :: A \to B$ in Fun, we can always construct a $p_I :: A \to (B \times H)$ in Inv such that $p_I; fst = p$. In other words, $p\ a = b$ if and only if there exists some h satisfying $p_I\ a = (b, h)$.

Such a p_I may not be unique, but always exists: you can always take $H = A$ and simply copy the input to the output. However, it is not immediately obvious how to construct such a $p_I :: A \to (B \times A)$ in Inv. Note that simply calling *dup* will not do, since not every function can be an argument to *dup*. Nor is it immediately obvious how to compose two transformed functions without throwing away the intermediate result. In Section 6.2, we will discuss the construction in more detail.

As another alternative, in Section 6.1 we will introduce what we call the "logging" translation, where a history of execution is recorded in H. The H resulting from the logging translation might not be the most interesting one, however. We can actually make p_I do different things by designing different H. We will see such an example in Section 6.3.

6.1 The Logging Translation

In this section we describe the logging translation from Fun functions to Inv. It basically works by pairing the result of the computation together with a history, where each choice of branch and each disposed piece of data is recorded, so one can always trace the computation back. It is similar to a translation for procedural languages described in [28].

Logging the History To represent the history, we introduce several new operators: *unit*, a constant function, like *zero* and *nil*, introduces the unit value (). Functions $inl :: A \to A + B$ and $inr :: B \to A + B$ wraps a value into a sum type. Finally, *in* builds recursive types.

The interesting fragments of the logging translation is summarised in Figure 3. The function *log* translates a Fun function into Inv, while returning a boolean value indicating whether it carries history or not. Using the associativity of composition and the following "split absorption" rule,

$$\langle f; h, g; k \rangle = \langle f, g \rangle; (h \times k)$$

```
log            :: Fun → (Inv, Bool)
log succ       = (succ, F)
                 :
log (f ∪ g)    = (h (log f); (id × inl)∪
                    h (log g); (id × inr), T)
    where h (f, F) = f; dup unit
          h (f, T) = f
log (f × g)    = case (log f, log g) of
   ((f′, F), (g′, F))  → ((f′ × g′), F)
   ((f′, T), (g′, F))  → ((f′ × g′); lsub, T)
   ((f′, F), (g′, T))  → ((f′ × g′); assocl, T)
   ((f′, T), (g′, T))  → ((f′ × g′); trans, T)
    where lsub = assocr; (id × swap); assocl

log μF     = (μ(X: fst (log (F(S X))); (id × in)), T)
log (S f)  = (f, T)
      — a placeholder for recursion
log fs | unsafe     =  compose (pipe hd) (log tl)
       | otherwise  =  compose (log hd) (log tl)
      — fs may be a split or a seq. of composition
     where (hd, unsafe, tl) = factor fs
             :

compose (f′, F) (g′, F)  = ((f′; g′), F)
compose (f′, T) (g′, F)  = (f′; (g′ × id), T)
compose (f′, F) (g′, T)  = (f′; g′, T)
compose (f′, T) (g′, T)  = (f′; (g′ × id); assocr, T)
```

Fig. 3. The logging translation

we can factor a sequence of composition into a head and a tail. The head segment uses only splits, *fst*, *snd*, converses, and constant functions. The tail segment, on the other hand, does not start with any of the constructs. For example, $\langle cons, fst\rangle$ is factored into $\langle id, fst\rangle; (cons \times id)$, while $nil^\circ; nil$ into $(nil^\circ; nil); id$. The factoring is done by the function *factor*. If the head does use one of the unsafe functions, it is compiled into Inv using the method to be discussed in the next section, implemented in the function *pipe*.

An expression $f; g$, where f and g have been translated separately, is translated into $f; (g \times id); assocr$ if both f and g carry history. If f carries history but g does not, it is translated into $f; (g \times id)$. A product $(f \times g)$, where f and g both carry history, is translated into $(f \times g); trans$, where *trans* couples the histories together. If, for example, only g carries history, it is translated into $(f \times g); assocl$.

A union $f \cup g$ is translated into $f; (id \times inl) \cup g; (id \times inr)$, where *inl* and *inr* guarantee that the ranges of the two branches are disjoint. We require both f and g to carry history. The branch not carrying history is postfixed with *dup unit* to create an empty history. Finally, the fixed-point $\mu(X\colon F(X))$ is translated to $\mu(X\colon F(X); (id \times in))$. Here we enforce that recursive functions always carry history. Known injective functions can be dealt with separately as primitives.

As an example, let us recall the function *cat* concatenating two lists. It is defined very similarly to *snoc*. The difference is that the two branches no longer have disjoint ranges.

$$\begin{aligned} cat = \mu(X\colon & (nil^\circ \times id); snd \cup \\ & (cons^\circ \times id); assocr; (id \times X); cons) \end{aligned}$$

The logging translation converts *cat* into

$$\begin{aligned} cat_I = \mu(X\colon & (nl; dup\ unit; (id \times inl) \cup \\ & (cons^\circ \times id); assocr; (id \times X); assocl; (cons \times inr)); \\ & (id \times in)) \end{aligned}$$

The function *pipe*, to be discussed in the next section, compiles expression $(nil^\circ \times id); snd$ into $swap; eq\ nil$. The first branch, however, does not carry history since no non-constant data is thrown away. We therefore make a call to *dup unit* to create an initial history. In the second branch, the recursive call is assumed to carry history. We therefore shift the history to the right position by *assocl*. The two branches are distinguished by *inl* and *inr*.

The history returned by cat_I is a sequence of *inr*s followed by *inl* — an encoding of natural numbers! It is the length of the first argument. In general, the history would be a tree reflecting the structure of the recursion.

Compiling Piping Functions Given a "natural" function defined in Fun in terms of splits, *fst*, *snd*, and constant functions, how does one find its equivalent, if any, in Inv? For mechanical construction, simple brute-force searching turned out to be satisfactory enough.

Take, for example, the expression $\langle\langle zero, snd; fst\rangle, \langle fst, snd; snd\rangle\rangle$. By a type inference on the expression, taking zero as a type of its own, we find that it transforms input of the form $(A,(B,C))$ to output $((0,B),(A,C))$. We can then start a breadth-first search, where the root is the input type $(A,(B,C))$, the edges are all applicable Inv formed by primitive operations and products, and the goal is the target type $((0,B),(A,C))$. To reduce the search space, we keep a count of copies of data and constants to be created. In this example, we need to create a fresh zero, so *dup zero* needs to be called once (and only once). Since no input data is duplicated, other calls to *dup* are not necessary. One possible result returned by the search is $swap; assocr; (dup\ zero \times id); (swap \times swap)$.

As another example, $nil^\circ; nil$, taking empty lists to empty lists, can be compiled into either $dup\ nil; eq\ id$ or $dup\ id; eq\ nil$. Some extra care is needed to distinguish input (whose domain is to be restricted) and generated constants, such that *id* and $dup\ id \cdot eq\ id$ are not legitimate answers. The search space is finite, therefore the search is bound to terminate.

When the Fun expression throws away some data, we compile it into an Inv expression that returns a pair whose first component is the output of the collects Fun expression. The forgotten bits are stored in the second component. For example, the Fun expression $\langle fst, snd; fst\rangle$ will be compiled into an Inv expression taking $(A,(B,C))$ to $((A,B),C)$, where C is the bit of data that is left out in the Fun expression. One possibility is simply *assocl*. The left components of the compiled Inv expressions constitute the "history" of the computation.

6.2 Relationship with the Reversible Turing Machine

The logging translation constructs, from a function $p :: A \rightarrow B$, a function $p_I :: A \rightarrow (B \times H)$, where H records the history of computation. The question, then, is what to do with the history? Throwing H away merely delays the loss of information and dissipation of heat. Then answer was given by Bennett in [4].

The basic configuration of Bennett's reversible Turing machine uses three tapes: one for input, one for output, and one for the history. Given a two-tape Turing machine accepting input A on the input tape and outputting B on the output tape, Bennett showed that one can always construct a three-tape reversible Turing machine which reads the input A, and terminates with A and B on the input and output tapes, while leaving the history tape *blank*. This is how it is done: in the first phase, the program is run forward, consuming the data on the input tape while writing to the output and history tapes. The output is then copied. In the third phase the program is run backwards, this time consuming the original output and history, while regenerating the input. This can be expressed in Inv by:

$$p_I; dup\,fst; ({p_I}^\circ \times id) :: A \rightarrow (A \times B)$$

We cannot entirely get rid of A, or some other sufficient information, if the computation is not injective. Otherwise we are losing information. When the computed function *is* injective, however, there is a way to erase both the history and input tapes empty. Bennett's method to do it can be expressed in our notation as the following. Assume that there exists a $q :: B \rightarrow A$, defined in Fun, serving as the inverse of p. The logging translation thus yields $q_I :: B \rightarrow (A, H')$ in Inv.

$$p_I; dup\,fst; ({p_I}^\circ \times id); swap; (q_I \times id); eq\,fst; {q_I}^\circ$$

The prefix $p_I; \mathit{dup}\, \mathit{fst}; (p_I{}^\circ \times \mathit{id})$, given input a, computes (a, b). The pair is swapped and b is passed though q_I, yielding $((a, h'), a)$. The duplicated a is removed by $\mathit{eq}\, \mathit{fst}$, and finally $q_I{}^\circ$ takes the remaining (a, h') and produces b.

The above discussion is relevant to us for another reason: it helps to show that Inv, even without an explicit looping construct, is computationally at least as powerful as the reversible Turing machine. Let p be a Fun function, defined tail-recursively, simulating a reversible Turing machine. The types A and B both represent states of the machine and the contents of the three tapes. The function q simulates the reversed Turing machine. They can be translated, via the logging translation, into Inv as functions that returns the final state together with an extra counter. The counter can then be eliminated using the above technique.

6.3 Preorder Traversal

To see the effect of the logging translation, and its alternatives, let us look at another example. Preorder traversal for binary trees can be defined in Fun as:

$$\begin{aligned} \mathit{pre} = \mu(X\colon & \mathit{null}^\circ; \mathit{nil} \cup \\ & \mathit{node}^\circ; (\mathit{id} \times (X \times X); \mathit{cat}_I); \mathit{cons}) \end{aligned}$$

The logging translation delivers the following program in Inv:

$$\begin{aligned} \mathit{pre}_I = \mu(X\colon & (\mathit{dup}\, \mathit{nil}; \mathit{eq}\, \mathit{null}; \mathit{dup}\, \mathit{unit}; (\mathit{id} \times \mathit{inl}) \cup \\ & \mathit{node}^\circ; (\mathit{id} \times (X \times X); \mathit{trans}; (\mathit{cat}_I \times \mathit{id}); \mathit{assocr}); (\mathit{cons} \times \mathit{inr})); \\ & (\mathit{id} \times \mathit{in})) \end{aligned}$$

which returns a tree as a history. In each node of the tree is a number, returned by cat_I, telling us where to split the list into two.

However, if we choose not to reply on the logging translation, we could have chosen $H = [A]$ and defined:

$$\begin{aligned} \mathit{prein} = \mu(X\colon & \mathit{dup}\, \mathit{nil}; \mathit{eq}\, \mathit{null}; \mathit{dup}\, \mathit{nil} \cup \\ & \mathit{node}^\circ; (\mathit{id} \times (X \times X); \mathit{trans}); \mathit{dcat}; \mathit{assocl}; (\mathit{cons} \times \mathit{id})) \end{aligned}$$

where dcat is as defined in Section 5.5. We recite its definition here:

$$\mathit{dcat}\, (a, ((x, y), (u, v))) = (a, (x +\!\!+\, y, u +\!\!+\, [a] +\!\!+\, v))$$

where a does not occur in x, and x and u have the same lengths. Since $(\mathit{id} \times \mathit{cat}); \mathit{cons} = \mathit{dcat}; \mathit{assocl}; (\mathit{cons} \times \mathit{id}); \mathit{fst}$, it is obvious that $\mathit{prein}; \mathit{fst}$ reduces to pre – with a restriction on its domain. The partial function prein accepts only trees with no duplicated labels. What about $\mathit{prein}; \mathit{snd}$? It reduces to inorder traversal of a binary tree. It is a known fact: we can reconstruct a binary tree having no duplicated labels from its preorder and inorder traversals [23].

7 Implementation

We have a prototype implementation of the logging translation and a simple, backtracking interpreter for Inv, both written in Haskell. The implementation of the logging translation, though tedious, poses no substantial difficulty. Producing an efficient, non-backtracking Inv interpreter, however, turned out to be more tricky than we expected.

Implementing a relational programming language has been discussed, for example, in [15] and [20]. Both considered implementing an executable subset of Ruby, a relational language for designing circuits [16]. A functional logic programming language was used for the implementation, which uses backtracking to collect all results of a relational program.

The problem we are dealing with here, however, is a different one. We *know* that legitimate programs in Inv are deterministic. But can we implement the language without the use of backtracking? Can we detect the error when the domains or ranges of the two branches of a union are not disjoint? Consider $snoc^\circ$, with the definition of *wrap* and *nl* expanded:

$$\begin{aligned} snoc^\circ = \mu(X : \, & cons^\circ; eq\ nil; dup\ nil; swap \cup \\ & cons^\circ; (id \times X); assocl; (cons \times id)) \end{aligned}$$

Both branches start with $cons^\circ$, and we cannot immediately decide which branch we should take.

A natural direction to go is to apply some form of domain analysis. Glück and Kawabe [13] recently suggested another approach. They observed that the problem is similar to parsing. A program is like a grammar, where the traces of a program constitute its language. Determining which branch to go is like determining which production rule to use to parse the input. In [13] they adopted the techniques used in LR parsing, such as building item sets and automatons, to construct non-backtracking programs. It is interesting to see whether the handling of parsing conflicts can be adapted to detect, report and resolve the non-disjointness of branches.

8 Conclusion and Related Work

We have presented a language, Inv, in which all functions definable are injective. It is a functional language with a relational semantics. Through examples, we find that many useful functions can be defined in Inv. In fact, it is computationally equivalent to Bennett's reversible Turing machines. Non-injective functions can be simulated in Inv via the logging translation which converts it to an Inv function returning both the result and a history.

A lot of previous work has been devoted into the design of programming languages and models for reversible computation. To the best of the authors' knowledge, they include Baker's PsiLisp [3], Lutz and Derby's JANUS [19], Frank's R [10], and Zuliani's model based on probabilistic guarded-command language [28]. Our work differs from the previous ones in several aspects: we base our model on a functional language; we do not rely on "hidden" features of the machine to record the history; and we highlight the importance of program derivation as well as mechanical inversion. We believe that Inv serves as a clean yet expressive model for the construction and reasoning of reversible programs.

The motivation to study languages for reversible programs traditionally comes from the thermodynamics view of computation. We were motivated for yet another reason — to build bi-directional editors. The source data is transformed to a view. The user can then edit the view, and the system has to work out how the source shall be updated correspondingly. Meertens [21] and Greenwald and Moore, et al. [14] independently developed their combinators for describing the source-to-view transformation, yet their results are strikingly similar. Both are combinator-like, functional languages allowing relatively unrestricted use of non-injective functions. Transformations are surjective functions/relations, and view-to-source updating is modelled by a function taking both the old source and the new view as arguments. Things get complicated when duplication is involved. We are currently exploring a slightly different approach, basing on Inv, in which the transformation is injective by default, and the use of non-injective functions is restricted to the *dup* operator. We hope this would make the model clearer, so that the difficulties can be better tackled.

Our work is complementary to Glück and Kawabe's recent work on automatic program inversion. While their focus was on automatic inversion of programs and ours on theory and language design, many of their results turned out to be highly relevant to our work. The stack-based intermediate language defined in [12] is actually an injective language. They also provided sufficient, though not necessary, conditions for the range-disjointness of branches. For a more precise check of disjointness they resort to the LR parsing technique [13]. Their insight that determining the choice of branches is like LR parsing is the key to build an efficient implementation of Inv. The authors are interested to see how conflict-handling can help to resolve the disjointness of branches.

Acknowledgements The authors would like to thank Robert Glück and Masahiko Kawabe for valuable discussions and encouragement. This research is partly supported by the e-Society Infrastructure Project of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

1. R. J. R. Back and J. von Wright. Statement inversion and strongest postcondition. *Science of Computer Programming*, 20:223–251, 1993.
2. R. C. Backhouse, P. de Bruin, G. Malcolm, E. Voermans, and J. van der Woude. Relational catamorphisms. In B. Möller, editor, *Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs*, pages 287–318. Elsevier Science Publishers, 1991.
3. H. G. Baker. NREVERSAL of fortune–the thermodynamics of garbage collection. In *Proc. Int'l Workshop on Memory Mgmt*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992.
4. C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.
5. C. H. Bennett. Thermodynamics of computation—a review. *International Journal of Theoretical Physics*, 21:905–940, 12 1982.
6. R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.
7. E. W. Dijkstra. Program inversion. Technical Report EWD671, Eindhoven University of Technology, 1978.
8. H. Doornbos and R. C. Backhouse. Induction and recursion on datatypes. In B. Möller, editor, *Mathematics of Program Construction, 3rd International Conference*, number 947 in Lecture Notes in Computer Science, pages 242–256. Springer-Verlag, July 1995.
9. H. Doornbos and R. C. Backhouse. Reductivity. *Science of Computer Programming*, 26:217–236, 1996.
10. M. P. Frank. The R programming language and compiler. MIT Reversible Computing Project Memo #M8, Massachusetts Institute of Technology, 1997. `http://www.ai.mit.edu/mpf/rc/memos/M08/M08_rdoc.html`.
11. E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982. MIT Report MIT/LCS/TM-197.
12. R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In A. Ohori, editor, *Programming Languages and Systems. Proceedings*, number 2895 in Lecture Notes in Computer Science, pages 246–264. Springer-Verlag, 2003.
13. R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on LR parsing (extended abstract). Submitted to the Seventh International Symposium on Functional and Logic Programming, 2004.
14. M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bidirectional tree transformations. University of Pennsylvania CIS Dept. Technical Report, MS-CIS-03-08, University of Pennsylvani, August 2003.

15. G. Hutton. The Ruby interpreter. Technical Report 72, Chalmers University of Technology, May 1993.
16. G. Jones and M. Sheeran. Circuit design in Ruby. In *Formal Methods for VLSI Design*. Elsevier Science Publishers, 1990.
17. R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.
18. Y. Lecerf. Machines de Turing réversibles. Récursive insolubilité en $n \in N$ de l'équation $u = \theta^n$, où θ est un "isomorphisme de codes". In *Comptes Rendus*, volume 257, pages 2597–2600, 1963.
19. C. Lutz and H. Derby. Janus: a time-reversible language. Caltech class project, California Institute of Technology, 1982. `http://www.cise.ufl.edu/~mpf/rc/janus.html`.
20. R. McPhee. Implementing Ruby in a higher-order logic programming language. Technical report, Oxford University Computing Laboratory, 1995.
21. L. Meertens. Designing constraint maintainers for user interaction. `ftp://ftp.kestrel.edu/ pub/papers/meertens/dcm.ps`, 1998.
22. S.-C. Mu and R. S. Bird. Inverting functions as folds. In E. Boiten and B. Möller, editors, *Sixth International Conference on Mathematics of Program Construction*, number 2386 in Lecture Notes in Computer Science, pages 209–232. Springer-Verlag, July 2002.
23. S.-C. Mu and R. S. Bird. Rebuilding a tree from its traversals: a case study of program inversion. In A. Ohori, editor, *Programming Languages and Systems. Proceedings*, number 2895 in Lecture Notes in Computer Science, pages 265–282. Springer-Verlag, 2003.
24. J. W. Sanders and P. Zuliani. Quantum programming . In R. C. Backhouse and J. N. F. d. Oliveira, editors, *Mathematics of Program Construction 2000*, number 1837 in Lecture Notes in Computer Science, pages 80–99. Springer-Verlag, 2000.
25. M. Takeichi, Z. Hu, K. Kakehi, Y. Hayashi, S.-C. Mu, and K. Nakano. TreeCalc:towards programmable structured documents. In *The 20th Conference of Japan Society for Software Science and Technology*, September 2003.
26. T. Toffoli. Reversible computing. In J. W. d. Bakker, editor, *Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.
27. S. G. Younis and T. F. Knight. Asymptotically zero energy split-level charge recovery logic. In *1994 International Workshop on Low Power Design*, page 114, 1994.
28. P. Zuliani. Logical reversibility. *IBM Journal of Research and Development*, 46(6):807–818, 2001. Available online at `http://www.research.ibm.com/journal/rd45-6.html`.

A Proof for the Labelled Concatenation

Let *lcat* be defined by

$$lcat = nmem\, fst \cdot \langle fst, id\rangle; (id \times subr; (id \times cons); cat)$$

where $nmem\, p\, (a, x) = (a, x)$ if a is not a member of the list $p\, x$, and $lcat_I$ be the fixed-point of *lcatF*, where

$$\begin{aligned} lcatF\ X = {} & (dup\, id \times nl); assocr; (id \times cons) \cup \\ & (id \times (cons^\circ \times id); assocr); \\ & \quad neq\, id\, fst; subr; (id \times X); subr; (id \times cons)) \end{aligned}$$

The aim is to show that *lcat* is also a fixed-point of *lcatF*. Starting with the more complicated branch of *lcatF*, we reason

$$\begin{aligned} & (id \times (cons^\circ \times id); assocr); neq\, id\, fst; subr; (id \times lcat); subr; (id \times cons) \\ = {} & \quad \{\text{definition of } lcat\} \\ & (id \times (cons^\circ \times id); assocr); neq\, id\, fst; subr; \\ & (id \times nmem\, fst; \langle fst, id\rangle; (id \times subr; (id \times cons); cat)); subr; (id \times cons) \\ = {} & \quad \{\text{since } (id \times (id \times R)); subr = subr; (id \times (id \times R))\} \end{aligned}$$

$(id \times (cons^\circ \times id); assocr); neq\ id\ fst; subr; (id\times$
$nmem\ fst; \langle fst, id\rangle; (id \times subr)); subr; (id \times (id \times (id \times cons); cat); cons)$

$=\quad$ {since $subr; (id \times nmem\ fst) = nmem\ (snd; fst); subr$}

$(id \times (cons^\circ \times id); assocr); neq\ id\ fst; nmem\ (snd; fst);$
$subr; (id \times \langle fst, id\rangle; (id \times subr)); subr; (id \times (id \times (id \times cons); cat); cons)$

$=\quad$ {expressing the piping in terms of splits}

$(id \times (cons^\circ \times id); assocr); neq\ id\ fst; nmem\ (snd; fst);$
$\langle fst, \langle snd; fst, \langle snd; snd; fst, \langle fst, snd; snd; snd\rangle\rangle\rangle\rangle;$
$(id \times (id \times (id \times cons); cat); cons)$

$=\quad$ {associativity: $(id \times cat); cat = assocl; (cons \times id); cat$,
and naturalty: $(id \times (id \times R)); assocl = assocl; (id \times R)$}

$(id \times (cons^\circ \times id); assocr); neq\ id\ fst; nmem\ (snd; fst);$
$\langle fst, \langle snd; fst, \langle snd; snd; fst, \langle fst, snd; snd; snd\rangle\rangle\rangle\rangle;$
$(id \times assocl); (id \times (cons \times cons); cat)$

$=\quad$ {move $(id \times assocr)$ rightwards}

$(id \times (cons^\circ \times id)); neq\ id\ (fst; fst); nmem\ (fst; snd); (id \times assocr)$
$\langle fst, \langle snd; fst, \langle snd; snd; fst, \langle fst, snd; snd; snd\rangle\rangle\rangle\rangle;$
$(id \times assocl); (id \times (cons \times cons); cat)$

$=\quad$ { cancelling $assocl$ and $assocr$ with splits}

$(id \times (cons^\circ \times id)); neq\ id\ (fst; fst); nmem\ (fst; snd);$
$\langle fst, \langle snd; fst, \langle fst, snd; snd\rangle\rangle\rangle; (id \times (cons \times cons); cat)$

$=\quad$ {split absorption}

$(id \times (cons^\circ \times id)); neq\ id\ (fst; fst); nmem\ (fst; snd);$
$\langle fst, \langle snd; fst; cons, \langle fst, snd; snd\rangle\rangle\rangle; (id \times (id \times cons); cat)$

$=\quad$ {products}

$(id \times (cons^\circ \times id)); neq\ id\ (fst; fst); nmem\ (fst; snd);$
$(id \times (cons \times id)); \langle fst, \langle snd; fst, \langle fst, snd; snd\rangle\rangle\rangle; (id \times (id \times cons); cat)$

$=\quad$ {since $neq\ id\ (fst; fst); nmem\ (fst; snd); (id \times (cons \times id))$
$= (id \times (cons \times id)); nmem\ fst$}

$(id \times (cons^\circ; cons \times id)); nmem\ fst;$
$\langle fst, \langle snd; fst, \langle fst, snd; snd\rangle\rangle\rangle; (id \times (id \times cons); cat)$

$=\quad$ {products}

$(id \times (cons^\circ; cons \times id)); nmem\ fst; \langle fst, id\rangle; (id \times subr);$
$(id \times (id \times cons); cat)$

$=\quad$ {folding $lcat$}

$(id \times (cons^\circ; cons \times id)); lcat$

For the other branch we reason:

$(id \times (nil^\circ; nil \times id)); lcat$

$=\quad$ {definition of $lcat$ and $(id \times (nil \times id)); nmem\ fst = (id \times (nil \times id))$}

$(id \times (nil^\circ; nil \times id)); \langle fst, id\rangle; (id \times subr; (id \times cons); cat)$

$=\quad$ {since $h; \langle f, g\rangle = \langle h; f, h; g\rangle$ for total h, f,g}

$(id \times (nil^\circ \times id)); \langle fst, (id \times (nil \times id))\rangle; (id \times subr; (id \times cons); cat)$

$=\quad$ {split absorption}

$(id \times (nil^\circ \times id)); \langle fst, (id \times (nil \times id)); subr; (id \times cons); cat\rangle$

$=\quad$ {since $(f \times (g \times h)); subr = subr; (g \times (f \times h))$}

$$
\begin{array}{cl}
& (id \times (nil^\circ \times id)); \langle fst, subr; (nil \times cons); cat\rangle \\
= & \quad \{\text{since } (nil \times id); cat = snd\} \\
& (id \times (nil^\circ \times id)); \langle fst, subr; (id \times cons); snd\rangle \\
= & \quad \{\text{since } (id \times f); snd = snd; f \text{ for total } f\} \\
& (id \times (nil^\circ \times id)); \langle fst, subr; snd; cons\rangle \\
= & \quad \{\text{split absorption, backwards}\} \\
& (id \times (nil^\circ \times id)); \langle fst, subr; snd\rangle; (id \times cons) \\
= & \quad \{\text{piping}\} \\
& (dup\ id \times swap; eq\ nil); assocr; (id \times cons)
\end{array}
$$

Therefore we conclude that

$$
\begin{array}{cl}
& lcatF\ lcat \\
= & \quad \{\text{with the reasoning above}\} \\
& (id \times (cons^\circ; cons \times id)); lcat \cup (id \times (nil^\circ; nil \times id)); lcat) \\
= & \quad \{\text{composition distributes into union}\} \\
& ((id \times (cons^\circ; cons \times id)) \cup (id \times (nil^\circ; nil \times id))); lcat \\
= & \quad \{\text{since } cons^\circ; cons \cup nil^\circ; nil = id\} \\
& lcat
\end{array}
$$