An Approach for Generalized Reversible Functional Programming

Stefan Bohne[1]
Baltasar Trancón Widemann[2]

[1] senTec Elektronik GmbH, Ilmenau
[2] Fakultät für Informatik und Automatisierung, TU Ilmenau

In software engineering, one often comes across pairs of functions which look "something-like-inverse" to each other; reading and writing a file, sending and receiving data, parsing and pretty-printing. Modern programming languages provide almost no support for keeping these function pairs consistent. The concepts bijection and injection more often than not do not apply and there is no other concept readily available for what it means for these function pairs to be consistent. We make an attempt to categorize different classes of such function pairs and investigate their properties whilst providing the basis for a functional programming language with some yet unseen advantages.

1 Introduction

It is not rare to come across pairs of functions that look like inverses of each other. Parsing a grammar and pretty-printing the corresponding tree is one example. These two functions are usually written independently of each other. A close inspection often reveals that their source code shares a lot of structure. For every case distinction in the parser there tends to be a corresponding case in the pretty-printer. The first mathematical concept that comes to mind here are bijective functions, or injective functions if we allow partiality. Unfortunately, the parser/pretty-printer pair typically is not a partial injection. Whitespace and superfluous parentheses are often dropped from the parsed tree. Yet one can still formulate a consistency requirement: pretty-printing a tree and then parsing the result should always yield exactly the same tree. Reading a file while allowing many old versions of the file format and writing the file in the latest version is another function pair that falls into this category. More examples are constructor/pattern pairs of algebraic data types, operations together with their undo-operation in user interfaces, and conversion between different file formats. Such a function pair is the type of mathematical object which we begin to study in this paper.

Keeping both functions consistent with each other by hand can be very error-prone in a large software project. Thus, it is not surprising that research in the area of reversible computation and bidirectional transformations has already investigated several solutions. In [6], simple bidirectional isomorphisms are composed to construct more complex parser/pretty-printer pairs. Logic programming and Definite Clause Grammars are used in [4] to define reversible grammars. The problem of whitespace is not addressed in these frameworks. The programming languages INV [5] for writing injective programs and Boomerang [1] for writing lenses both use a point-free style to compose complex function pairs, which is unfamiliar to many programmers and not always the best paradigm for a given problem. The programming language Janus [9] is an applicative, imperative, reversible programming language. RFun [10] is an applicative, functional, reversible programming language. Janus and RFun only deal with injective functions.

In this paper, we sketch a programming language for reversible programs, that allows a point-free and an applicative style. To make best use of the point-free style, the programming language also includes syntactical constructions common in functional languages. Moreover, we require that the semantics of the reversible sub-language are such that, when a reversible program is interpreted as an irreversible one, it has exactly the same behavior. In other words, reversible programs should be a subset of irreversible ones. Additionally, we will try to find suitable concepts of programs beyond injective functions that are appropriate for the examples given above.

Note that, we want to specify two functions, as opposed to finding an arbitrary reverse function. A comparison between other techniques and the combinator approach that we employ, can be found in [2]. As this is still much work in progress, our results must be taken tentatively.

1.1 Notation and Assumptions

We write function composition using ";" in 'computer science' order, i.e., $f;g = \lambda x \bullet g\,(f\,x)$. Function application binds strongest, followed by function composition. The identity function on A is denoted by $\mathrm{id}_A : A \to A$ with $\mathrm{id}_A\ x = x$ for all $x \in A$. We will drop type subscripts most of the time. When talking about operators, we use dots to denote the operator itself. For example, $\cdot;\cdot : (A \to B) \times (B \to C) \to A \to C$ is the function composition operator.

We use a partial lambda term of the form $\lambda x \mid P(x) \bullet E(x)$ that defines a function only on those values x where the predicate $P(x)$ is true. The function is undefined on other values, including values on which P is undefined. This notation is inspired by the Z-Notation [8].

We assume all functions to be continuous in the domain theoretic sense. Thus, we also assume a complete partial order ($\sqsubseteq$) with bottom ($\perp$) to exist on all types. Top and Bot denote the type of all values and the type of no values respectively.

We will use bold font to denote reversible functions and operators on reversible functions.

2 Structure of a Reversible Functional Program

As the title of this paper says, we are going to describe an approach for a programming language that is both *functional* and *reversible*. By functional, we mean that a program is a function. By functional programming we mean that a program is constructed by composing functions. By reversible we mean programs can be run in two directions: the normal direction, that assigns an output value to an input value, and the reverse direction, that assigns an input value to an output value.

Remark 1. What we mean by reversible here is different from what is typically referred to as reversible. A reversible function can be applied in reverse on its own output to produce *some* input value, not necessarily the original value. The reverse of functions that do have such a behavior are typically called *inverse*. A better terminology for such functions would be *invertible*.

We always treat both directions in parallel resulting in the following definition. This is also called the combinator approach [2].

Definition 1. *A pair of functions* $t = (\overrightarrow{t}, \overleftarrow{t})$ *with opposing types, i.e.,* $\overrightarrow{t} : A \to B$ *and* $\overleftarrow{t} : B \to A$*, is called a* janus*.* $\overrightarrow{t}$ *is called the* normal direction *and* $\overleftarrow{t}$ *the* reverse direction *of* t*. We will sometimes write* $t = \begin{pmatrix} \overrightarrow{t} \\ \overleftarrow{t} \end{pmatrix}$ *where it improves readability. We write* $A \rightleftarrows B$ *as an abbreviation for* $(A \to B) \times (B \to A)$*. The* composition*,* $t_1 ; t_2 : A \rightleftarrows C$*, of two januses* $t_1 : A \rightleftarrows B$ *and* $t_2 : B \rightleftarrows C$ *is defined as* $t_1 ; t_2 = (\overrightarrow{t_1} ; \overrightarrow{t_2}, \overleftarrow{t_2} ; \overleftarrow{t_1})$*. The* reverse*,* $t^\dagger : B \rightleftarrows A$*, of a janus* $t : A \rightleftarrows B$ *is defined as* $t^\dagger = (\overleftarrow{t}, \overrightarrow{t})$*.*

Our approach can now be phrased: whereas functional programming consists of composing functions, generalized reversible functional programming consists of composing januses.

Remark 2. It is easy to see that janus composition is associative, $(t^\dagger)^\dagger = t$ and $(t_1 ; t_2)^\dagger = t_2^\dagger ; t_1^\dagger$. Let $\mathbf{id}_A = (\mathrm{id}_A, \mathrm{id}_A)$, then januses form a dagger category – thus, the choice of symbol for janus reverse. Together with

$$\begin{aligned}
A \otimes B &= A \times B \\
t_1 \otimes t_2 &= (\lambda(a, b) \bullet (\overrightarrow{t_1}\, a, \overrightarrow{t_2}\, b), \lambda(c, d) \bullet (\overleftarrow{t_1}\, c, \overleftarrow{t_2}\, d)) \\
\mathbf{swap}_{A,B} &= (\lambda(a, b) \bullet (b, a), \lambda(b, a) \bullet (a, b)) \\
\mathbf{assoc}_{A,B,C} &= (\lambda((a, b), c) \bullet (a, (b, c)), \lambda(a, (b, c)) \bullet ((a, b), c)) \\
I &= \{()\} \\
\mathbf{right}_A &= (\lambda a \bullet (a, ()), \lambda(a, ()) \bullet a)
\end{aligned}$$

januses become a dagger symmetric monoidal category.

Even though this paper is about generalizing reversible programming beyond injective functions, they are a useful object of investigation for finding what makes reversible programs different from irreversible ones. One important observation is that injective functions cannot throw away information. The prime example for functions that throw away information are the projection functions, specifically $\pi_1 : A \times B \to A$ with $\pi_1\,(a, b) = a$. How could a janus look like whose normal direction is that of a projection function? There does not seem to be a generic way how the B-component can be recomputed from the A-component. Thus, we provide one ourselves.

Definition 2. *The janus constructor* $\mathbf{forget}_{A,B} : (A \to B) \to (A \times B \rightleftarrows A)$ *is defined as* $\mathbf{forget}_{A,B}\, f = (\pi_1, \lambda a \bullet (a, f\, a))$.

Related to not using a variable at all is the issue of using the value of a variable more than once. In the irreversible world, this is represented by the function $\delta_A : A \to A \times A$ with $\delta_A\, a = (a, a)$. In the reverse direction, it is possible that the two copies of the variable have different values. Which do we choose? Should we combine them? Disallow such combinations? And again, it is up to the programmer to decide. $(\mathbf{forget}_{A,A}\ \mathrm{id}_A)^\dagger$ is one way – keep the first value and ignore the second. Keeping the second value can be achieved by using $\mathbf{swap};(\mathbf{forget}\ \mathrm{id})^\dagger$. There also exists a canonical solution if the data type admits equality testing.

Definition 3. *Let A be a type equipped with a binary function* $\cdot == \cdot : A \times A \to \mathrm{Bool}$ *with* $(a_1 == a_2) \sqsubseteq (a_1 = a_2)$. *Then the janus* $\mathbf{dup}_A : A \rightleftarrows A \times A$ *is defined as* $\mathbf{dup}_A = (\delta_A, \lambda(a_1, a_2) \mid a_1 == a_2 \bullet a_1)$.

2.1 Janus Classes

What we have described so far, are just arbitrary pairs of functions with reverse type signatures. And actually, **forget** is sufficient to construct any janus from its two directions.

Corollary 1. *Any $t : A \rightleftarrows B$ can be decomposed as*

$$t = (\mathbf{forget}_{A,B}\ \overrightarrow{t}\,)^\dagger;\mathbf{swap}_{A,B};\mathbf{forget}_{B,A}\ \overleftarrow{t}\ \ .$$

If we assume that the relationship between normal and reverse direction in simple januses is always useful, then the mere way in which complex januses are constructed from simpler januses will likely result in a useful janus. Nonetheless, it is possible – and interesting – to ensure certain consistency conditions.

Figure 1 shows the janus subsets, which we call *janus classes*, that are going to be used in the remainder of this paper.

Class's name	Condition	Abbreviation
inverse	$\overrightarrow{t}\,;\overleftarrow{t} \sqsubseteq \mathrm{id} \wedge \overleftarrow{t}\,;\overrightarrow{t} \sqsubseteq \mathrm{id}$	in
semi-inverse	$\overrightarrow{t}\,;\overleftarrow{t} \sqsubseteq \mathrm{id}$	si
reverse semi-inverse	$\overleftarrow{t}\,;\overrightarrow{t} \sqsubseteq \mathrm{id}$	rs
pseudoinverse	$\overrightarrow{t}\,;\overleftarrow{t}\,;\overrightarrow{t} \sqsubseteq \overrightarrow{t} \wedge \overleftarrow{t}\,;\overrightarrow{t}\,;\overleftarrow{t} \sqsubseteq \overleftarrow{t}$	pi
generic	–	gj
irreversible	$(\overleftarrow{t} = \bot)$	ir

Fig. 1: Janus classes

Inverse janus es are similar to partial injective functions. The condition states that, if both directions are defined at a point, information is never lost. The difference to injective functions is that one direction may be defined and returns a value at which the other direction is undefined.

Semi-inverse and reverse semi-inverse januses can be seen as transformations that may lose information only in one direction. It is easy to see that **forget** f is reverse semi-inverse for any f, since the information that is computed by f the reverse direction is simply thrown away in the normal direction. All of the example function pairs given in the introduction are actually either semi-inverse or reverse semi-inverse.

Pseudoinverse januses are named so, because they have a lot in common with Moore–Penrose pseudoinverses of matrices. The condition on pseudoinverse januses can be rewritten into $t;t^\dagger;t \sqsubseteq t$, from which $t^\dagger;t;t^\dagger \sqsubseteq t^\dagger$ follows. Also, the januses $t;t^\dagger$ and $t^\dagger;t$ are (partial) idempotent and self-reverse, which loosely corresponds to being hermitian. Pseudoinverse januses can be seen as those losing information in both directions, but only during the first pass. After a pseudoinverse janus has been applied to a value in one direction, applying it again backward and again forward will yield the same value or will be undefined.

Irreversible 'januses' represent normal functions. They are, on one hand, isomorphic to the januses class given by the condition $\overleftarrow{t} = \bot$, but, on the other hand, they can be seen as the type $A \to B \times \text{Top} \to \text{Bot}$. Thus, they aren't really januses from the typing perspective, but this view will be useful when we define the semantics. They basically are januses of which we promise never to invoke the reverse direction.

In Figs. 2a and 2b we overload $\cdot;\cdot$ and $\cdot^\dagger$ on janus classes, such that, if t_i is in janus class J_i for $i \in \{1,2\}$, then $t_1;t_2$ is in janus class $J_1;J_2$ and $t_1^\dagger$ is in janus class $J_1^\dagger$. It is worth noting that inverse, semi-inverse and reverse semi-inverse januses are closed under composition, but pseudoinverse januses are not. Nonetheless, we can still compose pseudoinverse januses with semi-inverse januses from the right and reverse semi-inverse januses from the left.

;	in	si	rs	pi	gj	ir
in	in	si	rs	pi	gj	ir
si	si	si	gj	gj	gj	ir
rs	rs	pi	rs	pi	gj	ir
pi	pi	pi	gj	gj	gj	ir
gj	gj	gj	gj	gj	gj	ir
ir	ir	ir	ir	ir	ir	ir

(a) Composition

J	$J^\dagger$
in	in
si	rs
rs	si
pi	pi
gj	gj
ir	–

(b) Reverse

$$\begin{tikzcd} & \text{in} \arrow[dl] \arrow[dr] & \\ \text{si} \arrow[dr] & & \text{rs} \arrow[dl] \\ & \text{pi} \arrow[d] & \\ & \text{gj} \arrow[d] & \\ & \text{ir} & \end{tikzcd}$$

(c) Inclusion lattice

Fig. 2: Relations between janus classes

The correctness proof for $\cdot^\dagger$ is straight-forward. The correctness proof for $\cdot;\cdot$ requires many case distinctions and is not very enlightening. We will only prove one case here as an example.

Lemma 1. *Let $t_1 : A \rightleftarrows B$ be reverse semi-inverse and $t_2 : B \rightleftarrows C$ be pseudoinverse. Then $t_1;t_2 : A \rightleftarrows C$ is pseudoinverse.*

Proof. First, we have to prove $\overrightarrow{(t_1;t_2)};\overleftarrow{(t_1;t_2)};\overrightarrow{(t_1;t_2)} \sqsubseteq \overrightarrow{(t_1;t_2)}$. The left-hand side simplifies by definition 1 to

$$\overrightarrow{(t_1;t_2)};\overleftarrow{(t_1;t_2)};\overrightarrow{(t_1;t_2)} = \overrightarrow{t_1};\overrightarrow{t_2};\overleftarrow{t_2};\overleftarrow{t_1};\overrightarrow{t_1};\overrightarrow{t_2} \ .$$

Since $\overleftarrow{t_1};\overrightarrow{t_1} \sqsubseteq \mathrm{id}$ by assumption, and $\overrightarrow{t_2}$ is continuous and thus monotone, we have

$$\overrightarrow{(t_1;t_2)};\overleftarrow{(t_1;t_2)};\overrightarrow{(t_1;t_2)} \sqsubseteq \overrightarrow{t_1};\overrightarrow{t_2};\overleftarrow{t_2};\overrightarrow{t_2} \ .$$

We also have $\overrightarrow{t_2};\overleftarrow{t_2};\overrightarrow{t_2} \sqsubseteq \overrightarrow{t_2}$ by assumption and can conclude

$$\overrightarrow{(t_1;t_2)};\overleftarrow{(t_1;t_2)};\overrightarrow{(t_1;t_2)} \sqsubseteq \overrightarrow{t_1};\overrightarrow{t_2} = \overrightarrow{(t_1;t_2)} \ .$$

$\overleftarrow{(t_1;t_2)};\overrightarrow{(t_1;t_2)};\overleftarrow{(t_1;t_2)} \sqsubseteq \overleftarrow{(t_1;t_2)}$ is proven analogously, but using the assumption $\overleftarrow{t_2};\overrightarrow{t_2};\overleftarrow{t_2} \sqsubseteq \overleftarrow{t_2}$. Thus, $t_1;t_2$ is pseudoinverse. □

Figure 2c shows how janus classes are included in each other. This complete lattice defines a partial order, $\leq$, that we can use to broaden a janus type.

Finally, we look at how janus classes compose in parallel. We can prove in general that they are closed under any bifunctor.

Definition 4. *A bifunctor F is a mapping from pairs of types to types and also a (continuous) mapping from pairs of functions to functions, such that*

1. $f : A \to B \wedge g : C \to D \implies F\,(f,g) : F\,(A,C) \to F\,(B,D)$,
2. $F\,(\mathrm{id}_A, \mathrm{id}_B) = \mathrm{id}_{F\,(A,B)}$, *and*
3. $F\,(f_1;f_2, g_1;g_2) = (F\,(f_1,g_1));(F\,(f_2,g_2))$.

We extend a bifunctor F to januses, such that for any two januses $t_1 : A \rightleftarrows B$ and $t_2 : C \rightleftarrows D$, $F\,(t_1,t_2) = (F\,(\overrightarrow{t_1}, \overrightarrow{t_2}), F\,(\overleftarrow{t_1}, \overleftarrow{t_2}))$. Thus, we have $F\,(\mathbf{id},\mathbf{id}) = \mathbf{id}$, $F\,(t_1;t_2, t_3;t_4) = (F\,(t_1,t_3));(F\,(t_2,t_4))$ and $F\,(t_1^\dagger, t_2^\dagger) = (F\,(t_1,t_2))^\dagger$ from the functor laws.

Remark 3. The parallel composition, $\otimes$, from remark 2 is actually a bifunctor extended to januses. The bifunctor maps a pair of functions, (f,g), to the function $\lambda(a,b).(f\,a, g\,b)$.

Again, we shall show proof for only one janus class as an example.

Lemma 2. *Let $t_1 : A \rightleftarrows B$ and $t_2 : C \rightleftarrows D$ be both semi-inverse and F be a bifunctor, then $F\,(t_1,t_2) : F\,(A,C) \rightleftarrows F\,(B,D)$ is also semi-inverse.*

Proof. We have to prove $\overrightarrow{(F\,(t_1,t_2))};\overleftarrow{(F\,(t_1,t_2))} \sqsubseteq \mathrm{id}$. The left-hand side simplifies as follows

$$\overrightarrow{(F\,(t_1,t_2))};\overleftarrow{(F\,(t_1,t_2))} = F\,(\overrightarrow{t_1},\overrightarrow{t_2});F\,(\overleftarrow{t_1},\overleftarrow{t_2}) = F\,(\overrightarrow{t_1};\overleftarrow{t_1},\overrightarrow{t_2};\overleftarrow{t_2})$$

by definition 4. Because F is monotone and $\overrightarrow{t_i};\overleftarrow{t_i} \sqsubseteq \mathrm{id}$ for $i \in \{1,2\}$, we have

$$\overrightarrow{(F\,(t_1,t_2))};\overleftarrow{(F\,(t_1,t_2))} \sqsubseteq F\,(\mathrm{id},\mathrm{id}) = \mathrm{id}\ \ .$$

Thus, $F\,(t_1,t_2)$ is semi-inverse. □

2.2 Choice

With $\otimes$ it is possible to compose reversible functions in parallel. We now define the functor $\oplus$ that composes januses as alternatives of each other.

Definition 5. *Let $\oplus$ be the bifunctor with*

$$\begin{aligned} A \oplus B &= A + B \\ t_1 \oplus t_2 &= \lambda x \mid \begin{cases} x = \mathrm{inj}_1\, a \bullet \mathrm{inj}_1(\overrightarrow{t_1}\, a) \\ x = \mathrm{inj}_2\, b \bullet \mathrm{inj}_2(\overrightarrow{t_2}\, b) \end{cases} \end{aligned}$$

where $A + B$ is the sum type, and $\mathrm{inj}_1 : A \to A + B$ and $\mathrm{inj}_2 : B \to A + B$ are the corresponding injections.

To actually make the choice, we use the following janus constructor.

Definition 6. *Let $\mathbf{if}_{\mathrm{si}} : (A \to \mathrm{Bool}) \to (A \rightleftarrows A + A)$ with*

$$\mathbf{if}_{\mathrm{si}}\, c = \begin{pmatrix} \lambda x \mid \begin{cases} c\,x \bullet \mathrm{inj}_1\, x \\ \neg c\,x \bullet \mathrm{inj}_2\, x \end{cases} \\ \lambda y \mid \begin{cases} y = \mathrm{inj}_1\, a \bullet a \\ y = \mathrm{inj}_2\, a \bullet a \end{cases} \end{pmatrix} \ .$$

Since there is an isomorphism between $A + A$ and $A \times \mathrm{Bool}$, $\mathbf{if}_{\mathrm{si}}$ is just a special form of $\mathbf{forget}^{\dagger}$. Thus, $\mathbf{if}_{\mathrm{si}}$ is not inverse, but only semi-inverse. It discards one bit of information in the reverse direction. Therefore, we cannot use it to construct inverse, reverse semi-inverse, or pseudoinverse januses. We define an inverse version that checks whether the correct alternative was used in the reverse direction.

Definition 7. *Let $\mathbf{if}_{\mathrm{in}} : (A \to \mathrm{Bool}) \to (A \rightleftarrows A + A)$ with*

$$\mathbf{if}_{\mathrm{in}}\, c = \begin{pmatrix} \lambda x \mid \begin{cases} c\,x \bullet \mathrm{inj}_1\, x \\ \neg c\,x \bullet \mathrm{inj}_2\, x \end{cases} \\ \lambda y \mid \begin{cases} y = \mathrm{inj}_1\, a \wedge c\,a \bullet a \\ y = \mathrm{inj}_2\, a \wedge \neg c\,a \bullet a \end{cases} \end{pmatrix} \ .$$

With these janus constructors as building blocks, it is possible to define a janus from alternative januses depending on a condition and an assertion. For example, the janus $\mathbf{if}_{\mathrm{si}}\, c\mathbf{;}\,(t \oplus e)\,\mathbf{;}(\mathbf{if}_{\mathrm{in}}\, a)^{\dagger}$ is semi-inverse if t and e are semi-inverse. For the compound janus to be defined at all, c and a should be related predicates. One typically expresses the same condition as the other, but in terms of a different data type.

2.3 Higher-Order Januses

Since we want to create a reversible language that has as many features of irreversible languages as possible, we have to look at januses over januses. The first non-trivial higher-order janus that comes to mind, is the janus that reverses other januses.

Definition 8. *Let* $\mathbf{rev}_{A,B} : (A \rightleftarrows B) \rightleftarrows (B \rightleftarrows A)$ *with* $\mathbf{rev}_{A,B} = (\cdot^\dagger, \cdot^\dagger)$.

rev is a bijection and thus inverse.

A core concept of functional programming is the existence of a function $\mathrm{eval} : \mathrm{A} \times (\mathrm{A} \to \mathrm{B}) \to \mathrm{B}$, that applies a function to a value and returns the result. Turning this signature into a janus, like $\mathbf{eval}' : A \times (A \rightleftarrows B) \rightleftarrows B$, is not going to work. How are we supposed to compute from just a value the function and its argument from which the value originated? Instead, we use a trick. If not only the result of the janus application is returned, but also the janus, then we can define a useful, higher-order janus.

Definition 9. *Let* $\mathbf{jeval}_{A,B} : A \times (A \rightleftarrows B) \rightleftarrows B \times (A \rightleftarrows B)$ *with*

$$\begin{aligned}\overrightarrow{\mathbf{jeval}_{A,B}}\,(a,t) &= (\overrightarrow{t}\;a, t)\\ \overleftarrow{\mathbf{jeval}_{A,B}}\,(b,t) &= (\overleftarrow{t}\;b, t)\;.\end{aligned}$$

Currying is another core concept of functional programming. Since currying is a bijection, we can define it as a janus.

Definition 10. *Let* $\mathbf{curry}_{A,B,C} : (C \to A \to B) \rightleftarrows (A \times C \to B)$ *with*

$$\begin{aligned}\overrightarrow{\mathbf{curry}_{A,B,C}}\,f &= \lambda(a,c) \bullet f\,c\,a\\ \overleftarrow{\mathbf{curry}_{A,B,C}}\,f &= \lambda c \bullet \lambda a \bullet f\,(a,c)\;.\end{aligned}$$

Again, there is no obvious correspondent in the janus world. Using a similar trick as above we can find the following janus.

Definition 11. *Let* $\mathbf{jcurry}_{A,B,C} : (C \to (A \rightleftarrows B)) \rightleftarrows (A \times C \rightleftarrows B \times C)$ *with*

$$\begin{aligned}\overrightarrow{\mathbf{jcurry}_{A,B,C}}\,f &= (\lambda(a,c) \bullet (\overrightarrow{(f\,c)}\,a, c), \lambda(b,c) \bullet (\overleftarrow{(f\,c)}\,b, c)\\ \overleftarrow{\mathbf{jcurry}_{A,B,C}}\,t &= \lambda c \bullet (\lambda a \bullet \pi_1\,(\overrightarrow{t}\;(a,c)), \lambda b \bullet \pi_1\,(\overleftarrow{t}\;(b,c)))\;.\end{aligned}$$

The argument of type C acts like a context in which the transformation between A and B is performed (explaining our unusual choice of type variable names).

Remark 4. An interesting fact is that **jeval** can be derived from **jcurry** by $\mathbf{jeval}_{A,B} = \overrightarrow{\mathbf{jcurry}_{A,B,A\rightleftarrows B}}\;\mathrm{id}_{A\rightleftarrows B}$.

$\overrightarrow{\textbf{jcurry}}$ suggests an idea how to bridge the irreversible and reversible world. A function $f : C \to (A \rightleftarrows B)$ could be any *irreversible* function that computes a janus. $\overrightarrow{\textbf{jcurry}}$ turns this into a janus, that we can use as a building block to compose more complex januses from. One variable of type A (or more than one, if A is a tuple) is consumed to produce a new variable of type B, while using – but not consuming – a variable of type C in that transformation. What this also suggests is that, when some expression is applied to a janus, the janus itself may be computed in an irreversible fashion from all variables that are not consumed in that expression.

2.4 Recursion

Since januses are just pairs of functions, defining a generic janus recursively by taking the fixpoint, fix F, of a function $F : (A \rightleftarrows B) \to (A \rightleftarrows B)$ just works. For recursion to make sense for other janus classes, the janus class has to be ω-complete and contain $\bot$. In this case, we can apply fixpoint induction. Again, we prove this for semi-inverse januses as an example.

Lemma 3. *Let* $F : (A \rightleftarrows B) \to (A \rightleftarrows B)$ *preserve semi-inverse januses. Then* fix F *is semi-inverse.*

Proof. By fixpoint induction.

1. Since $\overrightarrow{\bot};\overleftarrow{\bot} = \bot \sqsubseteq \mathrm{id}$, $\bot$ is semi-inverse.
2. F preserves semi-inverse januses by assumption.
3. Let $t_1 \sqsubseteq t_2 \sqsubseteq \ldots$ be an ω-chain of semi-inverse januses, and let $t = \bigsqcup_i t_i$. Then we have

$$\overrightarrow{t};\overleftarrow{t} = \left(\bigsqcup_{i\in\omega} \overrightarrow{t_i}\right);\left(\bigsqcup_{i\in\omega} \overleftarrow{t_i}\right) = \bigsqcup_{i_1\in\omega}\bigsqcup_{i_2\in\omega} \overrightarrow{t_{i_1}};\overleftarrow{t_{i_2}} = \bigsqcup_{i\in\omega} \overrightarrow{t_i};\overleftarrow{t_i} \sqsubseteq \bigsqcup_{i\in\omega} \mathrm{id} = \mathrm{id}\ .$$

Thus, t is semi-inverse and the set of semi-inverse januses is ω-complete. □

3 Generalized Reversible Functional Programming

In the previous chapter, we have effectively defined a point-free language for generalized reversible functional programming. This chapter will do the same in an applicative style. The syntax for this language is given in Fig. 3.

$$
\begin{aligned}
J &::= \mathrm{in}|\mathrm{si}|\mathrm{rs}|\mathrm{pi}|\mathrm{gj}|\mathrm{ir} \\
E &::= V|\texttt{?}V \\
&\ \mid\ (E,\dots,E) \\
&\ \mid\ E\,E \\
&\ \mid\ \lambda_J E \Rightarrow E \mid \dots \mid E \Rightarrow E \\
&\ \mid\ S\texttt{;}E
\end{aligned}
\qquad
\begin{aligned}
T &::= \mathrm{Top}|\mathrm{Bot}|\mathrm{Equ}|\mathrm{Bool}|\mathrm{Int}|\mathrm{List}| \\
&\ \mid\ T \times \cdots \times T \\
&\ \mid\ T \leftarrow J \rightarrow T \\
S &::= S\texttt{;}S \\
&\ \mid\ \texttt{let}\, E \Leftarrow E \\
&\ \mid\ \texttt{forget}\, E \Leftarrow E \\
&\ \mid\ \texttt{remember}\, E \Leftarrow E
\end{aligned}
$$

Fig. 3: Syntax of the reversible language

Not surprisingly we generalized function types ($A \to B$) to janus types ($A \leftarrow J \rightarrow B$) by including the janus class. We also distinguish types with equality and without equality. Those with equality are a subtype of Equ. As expected from the type signature $(A \to B) \times (B \to A)$, reversible januses are invariant in their type arguments. But they are covariant in their janus class, i.e., a janus class $A \leftarrow J_1 \rightarrow B$ is considered a subclass of $A \leftarrow J_2 \rightarrow B$ if and only if $J_1 \leq J_2$.

What probably is most unusual is the omission of a sub-language for patterns. In this language all expressions can be used as a pattern. A definition involving a function application, `let` $x = f\,e$ for example, can be easily reversed if f is a reversible function. The reverse is `let` $e = f^\dagger\,x$. This is equivalent to writing `let` $f\,e = x$ in our language. Thus, when the term $f\,e$ is used to pattern match a value v, the reverse direction of the value of f is applied to v and this transformed value is then pattern matched against e.

The only exception is that λ-constructs cannot be used as a pattern. Matching against a λ-construct would mean finding the values of the free variables in the λ-construct that make it equal to the function matched against. This is undecidable in general. Thus, λ-expressions have to be restricted to the body of irreversible functions. Here, we do not enforce this restriction in the syntax to simplify the denotational semantics.

The denotational semantics, $[\![\cdot]\!]_E$, in Fig. 4 assigns each expression a janus of type $\Gamma \rightleftarrows \mathrm{Top} \times \Gamma$, where $\Gamma = V \to \mathrm{Top}$ is the type of variable assignments. The normal direction defines the semantics when the expression is used as a value. The reverse direction defines the semantics when the expression is used as a pattern.

Let-expressions are generalized to statements and the scoping expression, s`;`e. The `forget`- and `remember`-statements allow the explicit discarding and reconstruction of information. A statement's denotation, $[\![\cdot]\!]_S : \Gamma \rightleftarrows \Gamma$, is simply a janus between environments, as it can only produce and consume variables.

Tuple expressions evaluate their components from left to right. Thus, values consumed in the right sub-expression are available to the left sub-expression, but not vice versa. Pattern matching of tuples happens necessarily in the opposite direction, from right to left.

$$
\begin{aligned}
[\![v]\!]_E &= \begin{pmatrix} \lambda\gamma \bullet (\gamma\, v, \gamma) \\ \lambda(x,\gamma) \mid \gamma\, v == x \bullet \gamma \end{pmatrix} \\
[\![\mathtt{?}v]\!]_E &= \begin{pmatrix} \lambda\gamma \bullet (\gamma\, v, \gamma \backslash v) \\ \lambda(x,\gamma) | v \notin \mathrm{dom} \bullet \gamma \cup (v \mapsto x) \end{pmatrix} \\
[\![(e_1, \ldots, e_n)]\!]_E &= [\![e_1]\!] \,\hat{\otimes} (\ldots \hat{\otimes}\, [\![e_n]\!]) \\
&\ \text{where } a \hat{\otimes} b = a\mathbf{;}(\mathbf{id} \otimes b)\mathbf{;}\mathbf{assoc}^\dagger \\
[\![f\ e]\!]_E &= [\![e]\!]_E \,\overrightarrow{\mathbf{;jcurry}}\, (\overrightarrow{[\![f]\!]_E};\pi_1) \\
[\![\lambda_J p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n]\!]_E &= \begin{pmatrix} \lambda\gamma \bullet \big((\mathbf{forget}\ \lambda x.\gamma)^\dagger\mathbf{;}\beta\mathbf{;}\mathbf{forget}\ \lambda y.\gamma, \gamma\big) \\ - \end{pmatrix} \\
&\ \text{where } \beta = [\![p_1]\!]_E^\dagger \,\mathbf{;}\, [\![e_1]\!]_E \,\hat{\oplus}^J (\ldots \hat{\oplus}^J\, [\![p_n]\!]_E^\dagger \,\mathbf{;}\, [\![e_n]\!]_E) \\
&\ \text{where } a\, \hat{\oplus}^J\, b = \mathbf{if}_J\, (\overrightarrow{a} \cdot \neq \mathfrak{U})\mathbf{;}\, (a \oplus b)\,\mathbf{;}(\mathbf{if}_{J^\dagger}\, (\overleftarrow{a} \cdot \neq \mathfrak{U}))^\dagger \\
&\ \text{where } \mathbf{if}_{\mathrm{in}} = \mathbf{if}_{\mathrm{rs}} = \mathbf{if}_{\mathrm{pi}} \text{ and } \mathbf{if}_{\mathrm{si}} = \mathbf{if}_{\mathrm{gj}} = \mathbf{if}_{\mathrm{ir}} = \mathbf{if}_{\mathrm{ir}^\dagger} \\
[\![s\,\mathbf{;}\,e]\!]_E &= [\![s]\!]_S \,\mathbf{;}\, [\![e]\!]_E \\
[\![s_1\,\mathbf{;}\,s_2]\!]_S &= [\![s_1]\!]_S \,\mathbf{;}\, [\![s_2]\!]_S \\
[\![\mathtt{let}\ p \Leftarrow e]\!]_S &= [\![e]\!]_E \,\mathbf{;}\, [\![p]\!]_E^\dagger \\
[\![\mathtt{forget}\ p \Leftarrow e]\!]_S &= [\![p]\!]_E \,\mathbf{;swap;forget}\, (\overrightarrow{[\![e]\!]_E};\pi_1) \\
[\![\mathtt{remember}\ p \Leftarrow e]\!]_S &= [\![\mathtt{forget}\ p \Leftarrow e]\!]_S^\dagger
\end{aligned}
$$

Fig. 4: Denotational semantics of the reversible language

From the discussions about **dup** and **jcurry** we derive the following rules regarding variables:

1. Variables of types with equality may be used multiple times. They act as duplicates, when used as a value, or equality checks, when used as a pattern. This allows us to omit literal values in the syntax. Literal values can be emulated by defining them as variables in the outermost scope.
2. If the current scope is that of an irreversible function, variable usage is loosened to the normal define-before-use rule. There, even types without equality may be duplicated.
3. The janus sub-expression of a janus application is an irreversible scope and has access to all variables that are not consumed or produced in the argument sub-expression. The same is true for the right sub-expression of a `forget`- and `remember`-statement.
4. Otherwise, variables must be defined exactly once and then consumed exactly once. In order to differentiate between a definition/consumption use and a copy/equality test use of a variable, we introduce the `?`V form. This is only necessary in order to keep the semantics simple and compositional. Every variable must be defined using the `?`V form. In a reversible scope, the last usage of every variable must also be a `?`V form.

The other peculiarity of these semantics is how we treat pattern matching. Januses defined in this system are implicitly using the Maybe (or Option) monad. The special semantic value $\mathfrak{U}$ is used to denote when a janus is undefined at the given argument, the None (or Nothing) case. This includes the conditional lambda expression, that we have used until now, i.e., $\neg P(x) \implies (\lambda y \mid P(y) \bullet E(y))\, x = \mathfrak{U}$. All functions are implicitly strict in $\mathfrak{U}$, i.e., $f\,\mathfrak{U} = \mathfrak{U}$. The lambda expression is the only place where we treat $\mathfrak{U}$ in a special way and this is where the pattern matching happens. Also, note that $\mathfrak{U}$ is different from $\perp$. We leave $\perp$ as the semantic value for non-termination. Especially, non-termination in a predicate still leads to a non-terminating function, i.e., $P(x) = \perp \implies (\lambda y \mid P(y) \bullet E(y))\, x = \perp$.

Remark 5. This treatement is similar to exceptions in the programming language Haskell [3]. $\mathfrak{U}$ can be emulated by a special exception and pattern matching is then just syntactic sugar for handling this exception.

Pattern matching generally happens in a similar way to other functional languages. The sub-cases of a function are tried in order. The first case that matches, i.e., is not $\mathfrak{U}$, is the one that determines the output. But from the discussions about $\mathbf{if}_{\mathrm{si}}$ and $\mathbf{if}_{\mathrm{in}}$, we know that, depending on the janus class, we have to perform some consistency checking afterwards. For inverse, reverse semi-inverse, and pseudoinverse januses, we have to ensure that none of the cases, that were undefined in the normal direction, are defined in the reverse direction. The same is true when going backwards for inverse, semi-inverse, and pseudoinverse januses.

3.1 An Example

As an example, we will define the janus *parseInt* : List $\leftarrow\!\mathrm{rs}\!\rightarrow$ Int that computes from a list of digits the corresponding number and vice versa. *parseInt* shall discard leading zeros and, therefore, is reverse semi-inverse. For this example, we assume the constants true : Bool, false : Bool, nil : List and cons : Int $\times$ List $\leftarrow\!\mathrm{in}\!\rightarrow$ List are already defined in the global context with their usual meaning. These are usually defined as type constructors in irreversible languages. Since type constructors are always injective, they extend naturally to januses. We assume the binary operators $+$ and $*$ of type Int $\leftarrow\!\mathrm{ir}\!\rightarrow$ (Int $\leftarrow\!\mathrm{in}\!\rightarrow$ Int) which perform addition/subtraction and multiplication/partially defined division respectively. The syntactic sugar for these operators swaps the arguments, i.e., $l + r \equiv (\cdot + \cdot)\, r\, l$. Hence, it is the left operand which is consumed, and the right operand stays untouched. We also assume // : Int $\leftarrow\!\mathrm{ir}\!\rightarrow$ (Int $\leftarrow\!\mathrm{ir}\!\rightarrow$ Int) which performs integer division with rounding towards negative infinity.

The janus $d2n$ in Fig. 5 is a first step. It uses the janus constructor muladd : Int $\leftarrow\!\mathrm{ir}\!\rightarrow$ (Int $\times$ Int $\leftarrow\!\mathrm{rs}\!\rightarrow$ Int) with the following behavior:

$$\mathrm{muladd}\, k = \begin{pmatrix} \lambda(a, b) \bullet a * k + b \\ \lambda y \bullet (\lfloor y/k \rfloor, y \bmod k) \end{pmatrix} .$$

```
let muladd ⇐ λ_ir ?k ⇒
  λ_rs (?a, ?b) ⇒
      let ?w ⇐ ?a * k;
      let ?y ⇐ ?b + w;
      forget ?w ⇐ y // k * k;
      ?y;
let d2n ⇐ λ_rs
      nil ⇒ 0
    | cons(?d, ?l) ⇒
        muladd 10 (d2n ?l, ?d);
let append ⇐ λ_in
      (nil, ?x) ⇒ cons(?x, nil)
    | (cons(?y, ?l), ?x) ⇒
        cons(?y, append(?l, ?x));
let reverse ⇐ λ_in
      nil ⇒ nil
    | cons(?x, ?l) ⇒
        append(reverse ?l, ?x);
let compose_rs ⇐ λ_ir ?f ⇒ λ_ir
  ?g ⇒ λ_rs ?x ⇒ g (f ?x);
let parseInt ⇐
  compose_rs reverse d2n;
```

```
let divmod ⇐ λ_ir ?k ⇒
  λ_si ?y ⇒
      remember ?w ⇐ y // k * k;
      let ?b ⇐ ?y − w;
      let ?a ⇐ ?w / k;
      (?a, ?b);
let n2d ⇐ λ_si
      0 ⇒ nil
    | ?x ⇒ let (?n, ?d) ⇐ divmod 10 ?x;
        cons (?d, n2d ?n);
let unappend ⇐ λ_in
      cons(?x, nil) ⇒ (nil, ?x)
    | cons(?y, unappend† (?l, ?x)) ⇒
        (cons(?y, ?l), ?x);
let unreverse ⇐ λ_in
      nil ⇒ nil
    | ?y ⇒ let (?r, ?x) ⇐ unappend ?y;
        cons (?x, unreverse ?r);
let compose_si ⇐ λ_ir ?f ⇒ λ_ir
  ?g ⇒ λ_si ?x ⇒ g (f ?x);
let prettyPrintInt ⇐
  compose_si n2d unreverse;
```

Fig. 5: Semi-reversible number parser and its reverse

The statement `let` $?y \Leftarrow ?b + w$ is where we use $\overrightarrow{\textbf{jcurry}}$ to its full extent. The variable w is used here in an irreversible context, even though it is defined in a reversible context. This is sound, because w does not appear in the argument to the janus $\cdot + w$, which is just b.

The implementation of *d2n* is straight-forward, but it has a problem: It reads the digits in the wrong order. We want the most significant digit to be the first in the list. This is solved by the janus *reverse*, which is written in terms of the janus *append*. Their definition is no different from that in an irreversible, functional language. The definition just happens to be reversible.

Finally, we define *parseInt* in a point-free style, to show-case this possibility.

Figure 5 also shows an implementation of the reverse for each of the above januses and janus constructors. The reverse of any lambda expression, $\lambda_J p \Rightarrow b$, is simply $\lambda_{J^\dagger} e \Rightarrow p$. This does not help much in understanding what is going on. From the denotational semantics (Fig. 4) we can derive the simplification rules in Fig. 6. The implementations utilizes those simplifications, in order to make the source code easier to understand.

$$\begin{aligned}
[\![\texttt{let}\ e_1\, e_2 \Leftarrow e_3]\!]_S &= \left[\!\!\left[\texttt{let}\ e_2 \Leftarrow e_1^\dagger\, e_3\right]\!\!\right]_S \\
[\![\lambda_J \texttt{let}\ e_1 \Leftarrow e_2; e_3 \Rightarrow e_4]\!]_E &= [\![\lambda_J e_3 \Rightarrow \texttt{let}\ e_2 \Leftarrow e_1; e_4]\!]_E \\
[\![\lambda_J e_1\, e_2 \Rightarrow e_3]\!]_E &= [\![\lambda_J e_1\, ?x \Rightarrow \texttt{let}\ e_2 \Leftarrow ?x; e_2]\!]_E \quad \text{with } x \text{ new variable} \\
[\![\lambda_J \texttt{forget}\ e_1 \Leftarrow e_2; e_3 \Rightarrow e_4]\!]_E &= [\![\lambda_J e_3 \Rightarrow \texttt{remember}\ e_1 \Leftarrow e_2; e_4]\!]_E
\end{aligned}$$

Fig. 6: Useful equivalences

4 Related Work

The observation that duplication in one direction requires equality testing in the other direction – as in **dup** – has been noted before [5,10].

The construction $\mathbf{if}_{\text{in}}\, c\mathbf{;}\, (t \oplus e)\, \mathbf{;}(\mathbf{if}_{\text{in}}\, a)^{\dagger}$ is inspired and closely related to the `if`-statement in the programming language Janus. Our pattern matching algorithm is an instance of that construction. It actually generalizes the symmetric first-match policy from RFun.

Our treatment of januses as irreversible expressions – as motivated by $\overrightarrow{\mathbf{jcurry}}$ – is a generalization of reversible updates from [9].

Lenses are isomorphic to a subset of inverse januses. A lens l consists of two functions $l.get : A \to B$ and $l.put : B \times A \to A$ adhering to the lens laws

$$\begin{aligned} l.get(l.put(b, a)) &\sqsubseteq b \\ l.put(l.get(a), a) &\sqsubseteq a\ . \end{aligned}$$

These laws are equivalent to requiring the janus $t : A \rightleftarrows B \times A$ to be inverse and have $\overrightarrow{t};\pi_2 = \mathrm{id}_A$.

5 Future Work

A more thorough domain theoretic treatment of the reversible language is needed. Issues like recursion and loops will then have a stronger basis.

A proper type system should prove that defined januses actually belong to their respective janus class. The janus class of a reversible λ-expression is almost derivable from the denotational semantics from Fig. 4 using the foundations that were laid out in Section 2. We must additionally prove that $[\![v]\!]_E$ and $[\![?v]\!]_E$ are inverse, and that $\overrightarrow{\mathbf{jcurry}}\,(\overrightarrow{[\![f]\!]_E};\pi_1)$ has the same janus class as $\overrightarrow{[\![f]\!]_E}$.

A category theoretic treatment might also be enlightening. Especially in regards to quantum computation, which shares many properties with reversible programming and in recent years got their share of category theory in [7].

The same holds for linear type systems. The language proposed here has a lot in common with linear languages, only we do not consume variables in the function part of a function application. This suggests a modification of the modus ponens in linear logic to $\dfrac{A \to B \quad A}{A \to B \quad B}$, where the implication may be reused.

A usability issue comes up when we define binary operators like $\cdot + \cdot : \mathrm{Int} \to (\mathrm{Int} \rightleftarrows \mathrm{Int})$. Only one of its arguments can be consumed. The other one must be constant in the current scope. But which one? Is it necessary to have two versions of each binary operator? Is it useful to have typing rules that can deal with overloaded operators? Or is our current approach, where the left operand is always consumed, sufficient?

6 Introduction[†]

We have defined januses as the core concept of a reversible program along with other concepts to compose large januses from smaller ones. The functors $\otimes$ and $\oplus$ for parallel and alternative execution, **if** as basis for choice and pattern matching, **forget** as means to discard information and **jcurry** as the bridge between the irreversible and reversible world. One fundamental observation is that, unlike injective functions, general januses *can* discard information, but unlike irreversible functions, they cannot discard information *implicitly*.

We have also given the syntax and semantics for a programming language that allows to write reversible and irreversible januses. Recursion, pattern matching, and higher order programming are possible and even look like functional programming. The language allows to use irreversible functions to compute reversible januses and invoke them in even in a reversible context. As a side effect, this language generalizes pattern matching to all expressions except lambda abstractions. It especially allows to pattern-match against a janus application.

Januses with certain consistency requirements have been identified. These janus classes compose with relatively simple rules and form a complete lattice. Our example program shows how a very simplified parser can be written as a reverse semi-inverse janus. Its reverse, the pretty-printer, is implicitly specified by the same source code.

References

1. Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful Lenses for String Data. In *ACM SIGPLAN Notices*, pages 407–419, 2008.
2. Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three Complementary Approaches to Bidirectional Programming. *Generic and Indexed Programming*, pages 1–46, 2012.
3. Simon Peyton Jones, Alastair Reid, Tony Hoare, and Fergus Henderson. A Semantics for Imprecise Exceptions. In *ACM SIGPLAN Notices*, pages 25–36, 1999.
4. Peter Kourzanov. Bidirectional Parsing – A Functional/Logic Perspective. *International Symposia on Implementation and Application of Functional Languages (IFL 2014)*, 2014.
5. Shin-cheng Mu, Zhenjiang Hu, and Masato Takeichi. An Algebraic Approach to Bi-directional Updating. In *Programming Languages and Systems*, pages 2–20, 2004.
6. Tillmann Rendel and Klaus Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *ACM Haskell Symposium*, pages 1–12, 2010.
7. Peter Selinger. Dagger compact closed categories and completely positive maps (extended abstract). pages 1–23, 2005.
8. John Michael Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 2nd edition, 1998.
9. Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a Reversible Programming Language. In *Computing Frontiers*, pages 43–54, 2008.
10. Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a Reversible Functional Language. In *Reversible Computation*, pages 14–29, 2012.