Emordnilap - A Language for Ambidirectional Programming and its Categorical Semantics

Stefan Bohne[1][0000−1111−2222−3333], Second Author[2,3][1111−2222−3333−4444], and Third Author[3][2222−−3333−4444−5555]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

Abstract. The abstract should briefly summarize the contents of the paper in 15–250 words.

Keywords: First keyword · Second keyword · Another keyword.

No Title Given

No Author Given

No Institute Given

1 Introduction

There are many examples of pairs of functions that have a tight relationship, where one can be seen as the 'kind of inverse' of the other.

- reading/writing data to/from a file (de-/serialization),
- parsing/unparsing,
- conversion between different data formats,
- model transformations,
- do/undo in an editor,
- buying/selling,
- photographing/printing,
- recording/playing media,
- machine translation.

Like monads a few decades ago, this is a software pattern that seems to permeate software engineering but has not much support from programming languages. Usually, the two functions are written separately. But, anyone who has hand-written a top-down parser and corresponding pretty-printer, for example, has noticed how similar the structure of both functions is.

Let's look at a relatively simple example. The following functions are parsing and pretty-printing numbers. For simplicity, the string to parse already consists of individual digits.

```
let parse = λ
    cons(0, nil). 0
  | cons(d, r).   parse(r) * 10 + d
  in
let unparse = λ
    0. cons(0, nil)
  | n. let d = mod(n, 10) in
       let r = unparse((n - d) / 10) in
       cons(d, r)
```

The correspondences between `parse(r) * 10 + d` and `unparse((n - d) / 10)` are striking.

The relationship between the two functions may not be that of perfect reversibility. A parser typically discards white space. The example discards leading zeros. The reverse function, the pretty-printer, then cannot recreate the original

white space. But in this case, the behavior is actually intended. For this reason we make a distinction between the words *reversible* and *invertible*.

In the next section, TODO. In section 3, we will then introduce a (mostly untyped) functional programming language that has a unified syntax and semantics for both irreversible and reversible programs. This means that irreversible programs are written in the style of the normal lambda calculus. The language is extended with reversible functions. Moreover, we will see that the semantics of the reversible sub-language is such that, when a reversible program is interpreted as an irreversible one, it has (almost) the same semantics. In other words, reversible programs should be a subset of irreversible ones. Section 4 is dedicated to formalizing the language. We present a type system that defines which programs are valid and we prove its soundness. This shows that well-typed programs adhere to our janus classes.

It follows that a reversible morphism $f\colon A \to B$ (in $\mathcal{D}$) has two associated irreversible functions $f\nearrow\colon A \to B$ and $f\searrow = (f^\dagger)\nearrow\colon B \to A$ (in $\mathcal{C}$). This observation leads us our first category of reversible functions.

The idea of a reversible functional language is that, instead of programming by composing functions, a program is built by composing reversible functions. This lends itself nicely to an interpretation in category theoretical terms. Reversible programs will be modeled by morphisms in a dagger category $\mathcal{D}$.

Definition 1. *A* dagger category *is a category equipped with a functor $-^\dagger\colon \mathcal{D}^{\mathrm{op}} \to \mathcal{D}$ which is the identity on objects, i.e., it assigns to every morphism $f\colon A \to B$ a morphism $f^\dagger\colon B \to A$ 'going the opposite direction'. The functor must adhere to the laws $f^{\dagger\dagger} = f$ and $(f\,;g)^\dagger = g^\dagger\,;f^\dagger$.*

Irreversible functions are understood as the morphisms of a category $\mathcal{C}$. We will choose $\mathcal{C}$ to be a Kleisli-category over some cartesian closed category in the spirit of Moggi's computational lambda calculus [CITE]. So $\mathcal{C}$ is the semantical framework in which we can write functional programs and which we are already familiar with. In order to make use of those reversible functions, we have to turn them into irreversible functions. This shall be accomplished with a projection functor $-\nearrow\colon \mathcal{D} \to \mathcal{C}$. For the seamless integration of the reversible sublanguage into the irreversible language it is necessary to require $-\nearrow$ to be the identity on objects, and so $\mathcal{C}$ and $\mathcal{D}$ have to have the same objects.

It follows that a reversible morphism $f\colon A \to B$ (in $\mathcal{D}$) has two associated irreversible functions $f\nearrow\colon A \to B$ and $f\searrow = (f^\dagger)\nearrow\colon B \to A$ (in $\mathcal{C}$). This observation leads us our first category of reversible functions.

Definition 2. *For a category $\mathcal{C}$ we call a pair of $\mathcal{C}$-morphisms $(f,g)\colon (A \to B) \times (B \to A)$ a* Janus.

The category $\mathsf{J}\mathcal{C}$ is called the Janus category *of $\mathcal{C}$. It has*

- *the same objects as $\mathcal{C}$,*
- *as morphisms all januses of $\mathcal{C}$ where we treat (f,g) as going the same direction as f, i.e., $(f,g)\colon A \to B$ in $\mathsf{J}\mathcal{C}$ if $f\colon A \to B$ in $\mathcal{C}$,*
- *$(f,g)^\dagger = (g,f)$ as the dagger, and*

– $(f, g)\nearrow = f$ *as the projection functor from* $\mathsf{J}\mathcal{C}$ *to* $\mathcal{C}$.

Proposition 1. $\mathsf{J}\mathcal{C}$ *is a dagger category.*
If $\mathcal{C}$ *is monoidal or symmetric monoidal then so is* $\mathsf{J}\mathcal{C}$.

The janus category will serve as our running example model for reversible functions as it can be seen as the most generic category of reversible functions for a given category of irreversible functions as shown by the following lemma.

Definition 3. *Let* $\mathsf{JC}\mathcal{C}$ *be the category which has*

– *as objects all the pairs* $(\mathcal{D}, -\nearrow_{\mathcal{D}} \colon \mathcal{D} \to \mathcal{C})$ *of dagger categories and projection functors, and*
– *as morphisms all projection dagger functors* $F \colon \mathcal{D} \to \mathcal{D}'$*, i.e., functors such that* $F\,;\, -\nearrow_{\mathcal{D}'} = -\nearrow_{\mathcal{D}}$ *and* $F\,;\, -^{\dagger^{\mathcal{D}'}} = -^{\dagger^{\mathcal{D}}}\,;\, F$.

Proposition 2. $\mathsf{JC}\mathcal{C}$ *is indeed a category.*

Lemma 1. $\mathsf{J}\mathcal{C}$ *is a terminal object of* $\mathsf{JC}\mathcal{C}$.

Proof. The unique morphism $F \colon (\mathcal{D}, -\nearrow_{\mathcal{D}}) \to (\mathsf{J}\mathcal{C}, -\nearrow_{\mathsf{J}\mathcal{C}})$ is dictated by the definition of $\mathsf{J}\mathcal{C}$ and the requirement that the morphisms must commute with daggers and projections.

$$F\,;\pi_1 = F\,;\, -\nearrow_{\mathsf{J}\mathcal{C}} = -\nearrow_{\mathcal{D}}$$
$$F\,;\pi_2 = F\,;\, -^{\dagger^{\mathsf{J}\mathcal{C}}}\,;\, -\nearrow_{\mathsf{J}\mathcal{C}} = -^{\dagger^{\mathcal{D}}}\,;F\,;\, -\nearrow_{\mathsf{J}\mathcal{C}} = -^{\dagger^{\mathcal{D}}}\,;\, -\nearrow_{\mathcal{D}}$$

Another example for a reversible setting is the core of $\mathcal{C}$ which is (isomorphic to) a subcategory of $\mathsf{J}\mathcal{C}$ and inherits the dagger category structure from it.

Definition 4. *The category* $\mathsf{core}\mathcal{C}$ *has the same objects as* $\mathcal{C}$ *and as morphisms all the isomorphisms of* $\mathcal{C}$*, i.e., the morphisms* $f \colon A \to B$ *for which there exists a morphism* $g \colon B \to A$ *such that* $f\,;g = \mathrm{id}_A$ *and* $g\,;f = \mathrm{id}_B$.

1.1 Categorical Semantics of an Irreversible Language

$$\begin{aligned}
E ::= {} & C \mid V \mid () \mid (E, E) \mid \\
& E\,E \mid \lambda V.E \mid \texttt{let}\ P = E\ \texttt{in}\ E \\
V ::= {} & V \mid () \mid (P, P) \\
V = {} & \text{set of variable names} \\
C = {} & \text{set of constant symbols}
\end{aligned}$$

Fig. 1. Syntax of a basic functional language

Name	Typing (Expressions)	Semantics ($[\![-]\!]_\lambda$)
Const	$\Gamma \vdash_\lambda c_i : \tau_{c_i}$	$= !_{[\![\Gamma]\!]} \,;\, [\![c_i]\!]$ where $[\![c_i]\!] : I \to [\![\tau_{c_i}]\!]$
Var	$v_1 \colon \tau_1, \ldots, v_n \colon \tau_n \vdash_\lambda v_i : \tau_i$	$= !_{[\![\tau_1 \times \ldots \times \tau_{i-1}]\!]} \otimes \mathrm{id}_{[\![\tau_i]\!]} \otimes !_{[\![\tau_{i+1} \times \ldots \times \tau_n]\!]}$
Unit	$\Gamma \vdash_\lambda () : 1$	$= !_{[\![\Gamma]\!]}$
Tup	$\Gamma \vdash_\lambda e_1 : \tau_1$	$= g_1$
	$\Gamma \vdash_\lambda e_2 : \tau_2$	$= g_2$
	$\Gamma \vdash_\lambda (e_1, e_2) : \tau_1 \times \tau_2$	$= \Delta \,;\, (g_1 \otimes g_2)$
App	$\Gamma \vdash_\lambda e_1 : \tau_1 \to \tau_2$	$= g_1$
	$\Gamma \vdash_\lambda e_2 : \tau_1$	$= g_2$
	$\Gamma \vdash_\lambda e_1\, e_2 : \tau_2$	$= \Delta_{[\![\Gamma]\!]} \,;\, (g_2 \otimes \mathrm{id}_{[\![\Gamma]\!]}) \,;\, \mathrm{V}\, g_1$
Lam	$\Gamma, v \colon \tau_1 \vdash_\lambda e : \tau_2$	$= g$
	$\Gamma \vdash_\lambda \lambda v.e : \tau_1 \to \tau_2$	$= \Lambda\, g$
Let	$\Gamma \vdash_\lambda e_1 : \tau_1$	$= g_1$
	$\Delta \vdash^{\mathrm{P}}_\lambda p_2 : \tau_1$	$= g_2$
	$\Gamma, \Delta \vdash_\lambda e_3 : \tau_3$	$= g_3$
	$\Gamma \vdash_\lambda \mathtt{let}\, p_2 = e_1\, \mathtt{in}\, e_3 : \tau_3$	$= \Delta_{[\![\Gamma]\!]} \,;\, (\mathrm{id}_{[\![\Gamma]\!]} \otimes (g_1 \,;\, g_2)) \,;\, g_3$
Proj	$\Gamma \vdash_\lambda e : A \leftrightarrows B$	$= g$
	$\Gamma \vdash_\lambda e : A \to B$	$= g \,;\, - \nearrow$

Name	Typing (Patterns)	Semantics ($[\![-]\!]^{\mathrm{P}}_\lambda$)
VarP	$v \colon \tau \vdash^{\mathrm{P}}_\lambda v : \tau$	$= \mathrm{id}_{[\![\tau]\!]}$
UnitP	$\emptyset \vdash^{\mathrm{P}}_\lambda () : 1$	$= \mathrm{id}_{[\![1]\!]}$
TupP	$\Delta_1 \vdash^{\mathrm{P}}_\lambda p_1 : \tau_1$	$= g_1$
	$\Delta_2 \vdash^{\mathrm{P}}_\lambda p_2 : \tau_2$	$= g_2$
	$\Delta_1, \Delta_2 \vdash^{\mathrm{P}}_\lambda (p_1, p_2) : \tau_1 \times \tau_2$	$= g_1 \otimes g_2$
SumLP	$\Delta \vdash^{\mathrm{P}}_\lambda p : \tau_1$	$= g_1$
	$\Delta \vdash^{\mathrm{P}}_\lambda \mathrm{inj}_1\, p : \tau_1 + \tau_2$	$= g_1 \otimes g_2$
SumRP	$\Delta \vdash^{\mathrm{P}}_\lambda p : \tau_2$	$= g_2$
	$\Delta \vdash^{\mathrm{P}}_\lambda \mathrm{inj}_2\, p : \tau_1 + \tau_2$	$= g_1 \otimes g_2$

Fig. 2. Basic functional typing rules and semantics

Lets take a quick recap on how the semantics of a functional programming language are captured in category $\mathcal{C}$. Figure 1 shows the syntax of a simple functional language. $[\![-]\!]$ maps the types of the language to corresponding objects of the category. The typing and categorical semantics in Figure 2 assign to every well-typing $\Gamma \vdash_\lambda e : \tau$ a morphism $[\![e]\!]_\lambda : [\![\Gamma]\!] \to [\![\tau]\!]$ of the category.

A context with multiple variables is represented as a tensor product, written $[\![A \times B]\!] = A \otimes B$, of the types of the variables. Since the order of the variables shouldn't matter we assume isomorphisms between the types $(A \otimes B) \otimes C \cong A \otimes (B \otimes C)$ and $A \otimes B \cong B \otimes A$. The empty context is represented by the type $[\![1]\!] = I$ that is the identity of the tensor product, i.e., there are isomorphisms $A \otimes I \cong A \cong I \otimes A$. Additionally these isomorphisms should be subject to certain coherence laws that are not interesting at this point. So far what we have described is a *symmetric monoidal category*. We are going to assume these isomorphisms throughout this paper so in order to simplify the presentation we omit them.

The sublanguage of patterns is very simple. It has only three rules and categorical the semantics are exactly the identities of the type of the pattern. Note that the typing judgment $\Delta \vdash^{\mathrm{P}}_\lambda p : A$ is linear, i.e., every variable in Δ appears exactly once in the pattern p.

A few symbols appear in the semantics that we haven't explained yet. These are categorical concepts that go beyond a symmetric monoidal category and we should spend some more time on why we need them here. Note that the presentation in Figure 2 is equivalent to what one finds in textbooks like [CITE] but it is not the same. We chose this formulation to specifically highlight the differences between irreversible and reversible functional languages.

The typing rules allow expressions not to use all the variables in the context. So we need a way to throw away the ones we don't need. This is the purpose of $!_A \colon A \to I$. It allows us to throw away information. It is obvious how this is a problem for a reversible language. The typing rules also allow us to use a variable more than once. So $\Delta_A \colon A \to A \otimes A$ is needed to duplicate information. We will later define a multiple versions of this function. A symmetric monoidal category with $!_A$ and Δ_A (together with certain coherence laws) is called a *cartesian category*.

We also want to have first class functions. The type of functions between types A and B is denoted $[\![A \to B]\!] = [[\![A]\!] \to [\![B]\!]]$. In order to create terms of function types, we need to be able to turn a term that evaluates a context to a value into a term that evaluates a smaller context into a function that will evaluate into the same value when applied to the rest of the context. This is called currying and written as Λ. It forms the well-known isomorphism $[A \otimes C \to B] \cong [C \to [A \to B]]$. The inverse of this isomorphism is written V and required to apply a function. Cartesian categories with this structure are called *cartesian closed categories*.

Finally we will employ some syntactic sugar for patterns in lambda- and let-expressions as given in Figure 3. First, any lambda-expression with an argument pattern other than a simple variable will be translated to a lambda with a fresh

$$
\begin{aligned}
[\![\lambda p.e]\!]_{\mathsf{S}} &= \lambda v.\, [\![\mathtt{let}\, p = v\, \mathtt{in}\, e]\!]_{\mathsf{S}} \\
[\![\mathtt{let}\, (p, p_2) = e_1\, \mathtt{in}\, e_2]\!]_{\mathsf{S}} &= [\![\mathtt{let}\, (v, p_2) = e_1\, \mathtt{in}\, \mathtt{let}\, p_1 = v\, \mathtt{in}\, e_2]\!]_{\mathsf{S}} \\
[\![\mathtt{let}\, (v_1, p) = e_1\, \mathtt{in}\, e_2]\!]_{\mathsf{S}} &= \mathtt{let}\, (v_1, v) = e_1\, \mathtt{in}\, \mathtt{let}\, p = v\, \mathtt{in}\, [\![e_2]\!]_{\mathsf{S}} \\
[\![e]\!]_{\mathsf{S}} &= e \quad \text{otherwise}
\end{aligned}
$$

with $v_1 \in V$, $e, e_1, e_2, p_2 \in E$, $p \in E \backslash V$ and v a fresh variable

Fig. 3. Syntactic sugar

variable argument and a let-expression. Then complex tuple patterns will be resolved to a series of let-expressions with simple pairs-of-variable-patterns. This allows us for example to write $\lambda(x, ()).x$ instead of $\lambda x_1.\mathtt{let}\, (x, x_3) = x_1\, \mathtt{in}\, \mathtt{let}\, () = x_3$

$$
\begin{aligned}
-\,;-\colon & [B \to C] \otimes [A \to B] \to [A \to C] \\
&= [\![f, g \vdash \lambda a.f\ (g\ a)]\!]_{\lambda} \\
-\otimes-\colon & [A \to C] \otimes [B \to D] \to [A \otimes B \to C \otimes D] \\
&= [\![f, g \vdash \lambda(a, b).(f\ a, g\ b)]\!]_{\lambda} \\
\Delta_A\colon & I \to [A \to A \otimes A] \\
&= [\![\emptyset \vdash \lambda a.(a, a)]\!]_{\lambda} \\
!_A\colon & I \to [A \to I] \\
&= [\![\emptyset \vdash \lambda a.()]\!]_{\lambda} \\
\Lambda_{A,B,C}\colon & [C \otimes A \to B] \to [C \to [A \to B]] \\
&= [\![f \vdash \lambda c.\lambda a.f\ (c, a)]\!]_{\lambda} \\
\mathrm{V}_{A,B,C}\colon & [C \to [A \to B]] \to [C \otimes A \to B] \\
&= [\![f \vdash \lambda(c, a).f\ c\ a]\!]_{\lambda}
\end{aligned}
$$

Fig. 4. Self-enrichment of $\mathcal{C}$

Cartesian closed categories have the property that they can be represented within themselves. Figure 4 shows the morphisms that exist in $\mathcal{C}$ that recapture the structure of the cartesian closed category. If we add these morphisms as constant symbols to the language (and add syntactic sugar for infix operators) then Figure 2 can be seen as an interpreter for the functional language written in itself. This situation is called enrichment. In general a category $\mathcal{D}$ is enriched over a (usually different) monoidal category $\mathcal{C}$ when $\mathcal{C}$ has a family of objects to represent the $\mathcal{D}$-morphisms between two $\mathcal{D}$-types and all the relevant $\mathcal{D}$-structure is present in the form of $\mathcal{C}$-morphisms similar to those in Figure 4. For example the enriched associativity law for morphism composition is

$$((-\,;-) \otimes \mathrm{id})\,;(-\,;-) = (\mathrm{id} \otimes (-\,;-))\,;(-\,;-)$$

or more directly and more intuitively

$$[\![f, g, h \vdash (f\,;g)\,;h]\!]_\lambda = [\![f, g, h \vdash f\,;(g\,;h)]\!]_\lambda \ .$$

In general for all assumptions $a = b$ in the metalanguage we have a law for the enrichment that can be stated as $[\![a]\!]_\lambda = [\![b]\!]_\lambda$ in the object language.

As the lambda-calculus is much more succinct and more familiar to many readers, we will often use $[\![-]\!]_\lambda$ to give $\mathcal{C}$-morphisms. We generally leave out all the type information if it is obvious or not important.

All the rules are standard except Proj. This rule is an extension to the lambda calculus and only necessary to simplify the coherence theorems later. With this language we can define and use reversible functions in a point-free manner already. Proj allows us to omit $-\nearrow$ when a reversible function is used where an irreversible function is expected. Function reverse can achieved by the constant symbol $-^\dagger_{\tau_1,\tau_2} \colon [\tau_1 \leftrightarrows \tau_2] \to [\tau_2 \leftrightarrows \tau_1]$.

Constant symbols also allow us to extend the language without affecting the language consistency. For example let $\mathcal{C}$ have and object 2 such that there is isomorphisms $A \otimes A \cong [2 \to A]$. Let's call the type $[\![\mathsf{Bool}]\!] = 2$ and let $\mathsf{sel}_\tau \colon \tau \times \tau \to \mathsf{Bool} \to \tau$ be the constant symbol for that isomorphism. This allows us to use syntactic sugar for the familiar if-then-else construct.

$$[\![\texttt{if}\, i\, \texttt{then}\, t\, \texttt{else}\, e]\!]_{\mathsf{S}} = \mathsf{sel}\,(\lambda().t, \lambda().e)\, i\, ()$$

1.2 Thoughts on Reversible Semantics

Dagger categories are a natural framework for reversible semantics. They come with a strong consequence though: For any commuting diagram in a dagger category it's dual diagram also commutes. For example if the dagger category $\mathcal{D}$ has products $A \times B$ with

- the projection morphisms $\pi_1 \colon A \times B \to A$ and $\pi_2 \colon A \times B \to B$, and
- the unique morphism $\langle f, g\rangle \colon C \to A \times B$

then $A \times B$ is necessarily also a coproduct with

- the injection morphisms $\pi_1^\dagger \colon A \to A \times B$ and $\pi_2^\dagger \colon B \to A \times B$, and
- the unique morphism $\langle f^\dagger, g^\dagger\rangle^\dagger \colon C \to A \times B$.

Therefore the structure that we will require of the category for reversible semantics has to be self-dual. The non-self-dual structure used in Figure 2 is that of the cartesian product and the cartesian closedness which we extracted into the symbols !, Δ, Λ and V.

Proposition 3. *A symmetric monoidal category is cartesian if and only if there are natural transformations $\Delta_A \colon A \to A \otimes A$ and $!_A \colon A \to I$ such that*

$$A \xrightarrow{\Delta_A} A \otimes A \xrightarrow{\mathrm{id}_A \otimes !_A} A \otimes I \xrightarrow{\cong} A = \mathrm{id}_A$$

$$A \xrightarrow{\Delta_A} A \otimes A \xrightarrow{!_A \otimes \mathrm{id}_A} I \otimes A \xrightarrow{\cong} A = \mathrm{id}_A$$

Proposition 4. *A cartesian category is cartesian closed if and only if there is a natural isomorphism* $\Lambda_{A,B,C} \colon [A \otimes C \to B] \to [C \to [A \to B]]$.

So a symmetric monoidal category is as close as we can get to a cartesian closed category whilst keeping all the required structure self-dual. But we can do better by requiring that our dagger category $\mathcal{D}$ is enriched over our cartesian closed category $\mathcal{C}$. We shall denote the external hom-object of $\mathcal{D}$ in $\mathcal{C}$ by $[A \leftrightarrows B]$.

Let's take a look at the evaluation morphism

$$\mathrm{ev}^{\mathcal{C}}_{A,B} \colon A \otimes [A \to B] \to B = \mathrm{V}^{\mathcal{C}}_{A,B,[A\to B]}\, \mathrm{id}_{[A\to B]} \,.$$

We have to find a a reversible version such that

$$\begin{aligned} \mathrm{ev}^{\mathcal{D}}_{A,B}\nearrow &\colon A \otimes [A \leftrightarrows B] \to B = \mathrm{ev}^{\mathcal{C}}_{A,B} \\ \mathrm{ev}^{\mathcal{D}}_{A,B}\searrow &\colon B \to A \otimes [A \leftrightarrows B] = ? \end{aligned}$$

but we would be hard pressed to find a non-trivial, cartesian closed category with a morphism for $\mathrm{ev}^{\mathcal{D}}_{A,B}\searrow$. The key idea is that ev's type could be made self-dual if it returned the function that was applied in addition to the value, i.e., $\mathrm{jev}_{A,B} \colon A \otimes [A \leftrightarrows B] \to B \otimes [A \leftrightarrows B]$. In the Janus category we can easily define a suitable definition.

$$\mathrm{jev}_{A,B} = \begin{pmatrix} [\![a, f \vdash (f \nearrow a, f)]\!]_\lambda \\ [\![b, f \vdash (f \searrow b, f)]\!]_\lambda \end{pmatrix}$$

This idea of retaining information that is used to reversibly transform other pieces of information is embodied in the following function.

$$\begin{aligned} \mathrm{jV}^{\mathsf{J}\mathcal{C}} &\colon [C \to [A \leftrightarrows B]] \to [A \otimes C \leftrightarrows B \otimes C] \\ \mathrm{jV}^{\mathsf{J}\mathcal{C}}\, f &= \begin{pmatrix} [\![a, c \vdash ((f\ c) \nearrow a, c)]\!]_\lambda \\ [\![b, c \vdash ((f\ c) \searrow b, c)]\!]_\lambda \end{pmatrix} \end{aligned}$$

So jV is a way to convert $\mathcal{C}$-morphisms that construct $\mathcal{D}$-morphisms into a proper $\mathcal{D}$-morphism. It allows us to create reversible functions in an irreversible manner. This is where most of the expressiveness of Emordnilap comes from. It looks similar to uncurrying and just as $\mathrm{V}\,\mathrm{id} = \mathrm{ev}$ we have $\mathrm{jV}\,\mathrm{id} = \mathrm{jev}$. It is weaker though, because jV is not necessarily an isomorphism.

Definition 5. *A* jV-structure *is given by the tuple* $(\mathcal{C}, \mathcal{D}, -\nearrow, \mathrm{jV}^{\mathcal{D}})$ *where*

- $\mathcal{C}$ *is a cartesian closed category with internal hom-object* $[A \to B]$,
- $\mathcal{D}$ *is a symmetric monoidal category* $\mathcal{D}$ *with the same objects as* $\mathcal{C}$ *and enriched over* $\mathcal{C}$ *with external hom-object* $[A \leftrightarrows B]$,
- $-\nearrow \colon \mathcal{D} \to \mathcal{C}$ *is an identity on objects* $\mathcal{C}$*-enriched, symmetric monoidal functor, and*
- *for every* $\mathcal{C}$*-morphism* $f \colon C \to [A \leftrightarrows B]$ *there is a* $\mathcal{D}$*-morphism* $\mathrm{jV}^{\mathcal{D}}\, f \colon A \otimes C \to B \otimes C$ *such that* $(\mathrm{jV}^{\mathcal{D}}\, f)\nearrow = \mathrm{jV}^{\mathcal{C}}\, (f\,; -\nearrow)$ *where* $\mathrm{jV}^{\mathcal{C}}\, f = [\![a, c \vdash (f\ c\ a, c)]\!]$

A †jV-structure *is a* jV*-structure* $(\mathcal{C}, \mathcal{D}, -\nearrow, \mathrm{jV}^{\mathcal{D}})$ *such that*

– $\mathcal{D}$ *is a dagger symmetric monoidal category, and*
– $-\nearrow$ *is a dagger functor.*

Example 1. $(\mathcal{C}, \mathsf{J}\mathcal{C}, \pi_1, \mathrm{jV}^{\mathsf{J}\mathcal{C}})$ is a †jV-structure.

Example 2. Let $\mathcal{D}$ be the wide subcategory of $\mathsf{J}\mathcal{C}$ where for all Januses f we have $f\searrow ; f\nearrow = \mathrm{id}$. We can retain jV from $\mathsf{J}\mathcal{C}$, because for any $g\colon C \to [A \leftrightarrows B]$ in $\mathcal{C}$ we have

$$\begin{aligned}
&(\mathrm{jV}^{\mathsf{J}\mathcal{C}}\, g)\searrow ; (\mathrm{jV}^{\mathsf{J}\mathcal{C}}\, g)\nearrow \\
=&[\![\mathtt{a}, \mathtt{c} \vdash ((g\ \mathtt{c})\searrow\ \mathtt{a}, \mathtt{c})]\!]_\lambda \ ; \ [\![\mathtt{b}, \mathtt{c} \vdash ((g\ \mathtt{c})\nearrow\ \mathtt{b}, \mathtt{c})]\!]_\lambda \\
=&[\![\mathtt{a}, \mathtt{c} \vdash ((g\ \mathtt{c})\nearrow ((g\ \mathtt{c})\searrow\ \mathtt{a}), \mathtt{c})]\!]_\lambda \\
=&[\![\mathtt{a}, \mathtt{c} \vdash (\mathtt{a}, \mathtt{c})]\!]_\lambda \\
=&\mathrm{id}
\end{aligned}$$

Unfortunately $\mathcal{D}$ is not self-dual, but $(\mathcal{C}, \mathcal{D}, -\nearrow, \mathrm{jV}^{\mathcal{D}})$ is still a jV-structure. Parser/pretty-printer pairs usually have this property.

Example 3. $\mathsf{core}\mathcal{C}$ has a canonical †jV-structure with $-\nearrow\colon \mathsf{core}\mathcal{C} \to \mathcal{C}$ the injection functor and $\mathrm{jV}^{\mathsf{core}\mathcal{C}}\, f = \mathrm{jV}^{\mathcal{C}} f$. Note that if $[\![\mathtt{a} \vdash f\, c\, \mathtt{a}]\!]_\lambda : A \to B$ is an isomorphism for all c then $[\![\mathtt{a}, \mathtt{c} \vdash (f\ \mathtt{c}\ \mathtt{a}, \mathtt{c})]\!]_\lambda : A \otimes C \to B \otimes C$ is an isomorphism too. This can be shown with an analogous argument as for Example 2 but for both $\mathsf{core}\mathcal{C}$ and $\mathsf{core}\mathcal{C}^{\mathrm{op}}$.

1.3 The Categorical Semantics of Emordnilap

The formal semantics of Emordnilap are given in Figure 5. The typing judgement $\Gamma\ ;\ \Delta \vdash_\dagger e : j\ A$ splits the context into a non-linear part Γ and a linear part Δ. Additionally we distinguish between reversible and irreversible expression with $j \in \{\to, \leftrightarrows\}$. Each well-typed expressions is a assigned a morphism $[\![e]\!]_\dagger : [\![\Gamma]\!] \to [[\![\Delta]\!]\ j\ [\![A]\!]]$ in $\mathcal{C}$. This uses the fact that both $\mathcal{C}$ and $\mathcal{D}$ are enriched over $\mathcal{C}$ and allows us to construct reversible expression in a non-reversible manner.

Lemma 2. *The following holds.*

(i) $\Gamma\ ;\ \Delta \vdash_\dagger e : \leftrightarrows A$ *implies* $\Gamma, \Delta\ ;\ \emptyset \vdash_\dagger e : \to A$
(ii) $\Gamma\ ;\ \emptyset \vdash_\dagger e : \to [A \leftrightarrows B]$ *implies* $\Gamma\ ;\ \emptyset \vdash_\dagger e : \to [A \to B]$

The proof requires weakening lemmas.

Lemma 3. $\Gamma \vdash_\lambda e : A$ *implies*

(i) $\Gamma, \Gamma' \vdash_\lambda e : A$ *and*
(ii) $[\![\Gamma, \Gamma' \vdash_\lambda e : A]\!]_\lambda = \pi_1\ ;\ [\![\Gamma \vdash_\lambda e : A]\!]_\lambda$.

Proof. Simple induction.

Lemma 4. $\Gamma\ ;\ \Delta \vdash_\dagger e : j\ A$ *implies*

Name	Typing	Semantics ($[\![-]\!]_\dagger$)
Const	$\Gamma \,;\, \emptyset \vdash_\dagger c_i : \to C_i$	$= [\![\gamma \vdash \lambda().c_i]\!]_\lambda$
Var	$v_1 \colon A_1, \ldots, v_n \colon A_n \,;\, \emptyset \vdash_\dagger v_i : \to A_i$	$= [\![v_1 \colon A_1, \ldots, v_n \colon A_n \vdash \lambda().v_i]\!]_\lambda$
RVar	$\Gamma \,;\, v \colon A \vdash_\dagger v : \leftrightarrows A$	$= [\![\gamma \vdash \mathrm{id}_{[\![A]\!]}]\!]_\lambda$
LetV	$\Gamma, \Delta_2 \,;\, \Delta_1 \vdash_\dagger e_1 : j\ A$	$= g_1$
	$\Gamma \,;\, v \colon A, \Delta_2 \vdash_\dagger e_2 : j\ B$	$= g_2$
	$\Gamma \,;\, \Delta_1, \Delta_2 \vdash_\dagger \mathtt{let}\, v = e_1\, \mathtt{in}\, e_2 : j\ B$	$= [\![\gamma \vdash (\mathrm{jV}\ \lambda\delta_2.g_1\,(\gamma, \delta_2)) \,;\, g_2\,\gamma]\!]_\lambda$
App	$\Gamma \,;\, \emptyset \vdash_\dagger e_1 : \to [A\ j\ B]$	$= g_1$
	$\Gamma \,;\, \Delta \vdash_\dagger e_2 : j\ A$	$= g_2$
	$\Gamma \,;\, \Delta \vdash_\dagger e_1\, e_2 : j\ B$	$= [\![\gamma \vdash g_2\,\gamma \,;\, g_1\,\gamma\,()]\!]_\lambda$
Lam	$\Gamma \,;\, v \colon A \vdash_\dagger e : j\ B$	$= g$
	$\Gamma \,;\, \emptyset \vdash_\dagger \lambda v.e : \to [A\ j\ B]$	$= [\![\gamma \vdash \lambda().g\,\gamma]\!]_\lambda$
Unit	$\Gamma \,;\, \emptyset \vdash_\dagger () : j\ 1$	$= [\![\gamma \vdash \mathrm{id}_{[\![1]\!]}]\!]_\lambda$
LetU	$\Gamma, \Delta_2 \,;\, \Delta_1 \vdash_\dagger e_1 : j\ 1$	$= g_1$
	$\Gamma \,;\, \Delta_2 \vdash_\dagger e_2 : j\ B$	$= g_2$
	$\Gamma \,;\, \Delta_1, \Delta_2 \vdash_\dagger \mathtt{let}\, () = e_1\, \mathtt{in}\, e_2 : j\ B$	$= [\![\gamma \vdash (\mathrm{jV}\ \lambda\delta_2.g_1\,(\gamma, \delta_2)) \,;\, g_2\,\gamma]\!]_\lambda$
Tup	$\Gamma \,;\, \Delta_1 \vdash_\dagger e_1 : j\ A_1$	$= g_1$
	$\Gamma \,;\, \Delta_2 \vdash_\dagger e_2 : j\ A_2$	$= g_2$
	$\Gamma \,;\, \Delta_1, \Delta_2 \vdash_\dagger (e_1, e_2) : j\ A_1 \times A_2$	$= [\![\gamma \vdash g_1\,\gamma \otimes g_2\,\gamma]\!]_\lambda$
LetT	$\Gamma, \Delta_2 \,;\, \Delta_1 \vdash_\dagger e_1 : j\ A_1 \times A_2$	$= g_1$
	$\Gamma \,;\, v_1 \colon A_1, v_2 \colon A_2, \Delta_1 \vdash_\dagger e_2 : j\ B$	$= g_2$
	$\Gamma; \Delta_1, \Delta_2 \vdash$ $\mathtt{let}\,(v_1, v_2) = e_1\, \mathtt{in}\, e_2 \colon j\ B$	$= [\![\gamma \vdash (\mathrm{jV}\ \lambda\delta_2.g_1\,(\gamma, \delta_2)) \,;\, g_2\,\gamma]\!]_\lambda$
Proj	$\Gamma \,;\, \emptyset \vdash_\dagger e : \to A \leftrightarrows B$	$= g$
	$\Gamma \,;\, \emptyset \vdash_\dagger e : \to A \to B$	$= [\![\gamma \vdash \lambda().(g\,\gamma)\nearrow]\!]_\lambda$
Dup	$\Gamma \,;\, \Delta, x : A, x : A \vdash_\dagger e : \leftrightarrows B$	$= g$
	$\Gamma \,;\, \Delta, x : A \vdash_\dagger e : \to B$	$= [\![\gamma \vdash (\mathrm{id}_{[\![\Delta]\!]} \otimes \Delta_A) \,;\, g\,\gamma]\!]_\lambda$
Term	$\Gamma \,;\, \Delta \vdash_\dagger e : \leftrightarrows B$	$= g$
	$\Gamma \,;\, \Delta, x : A \vdash_\dagger e : \to B$	$= [\![\gamma \vdash (\mathrm{id}_{[\![\Delta]\!]} \otimes !_A) \,;\, g\,\gamma]\!]_\lambda$

Fig. 5. Categorical semantics for Emordnilap

(i) $\Gamma, \Gamma' \,;\, \Delta \vdash_\dagger e : j\ A$ *and*
(ii) $[\![\Gamma, \Gamma' \,;\, \Delta \vdash_\dagger e : j\ A]\!]_\dagger = \pi_1 \,;\, [\![\Gamma \,;\, \Delta \vdash_\dagger e : j\ A]\!]_\dagger$.

Proof. 4.(i) Simple induction.

4.(ii) By case analysis and Lemma 3.

Proof (of Lemma 2). 2.(i) Simple induction and use of Lemma 4 to make the contexts agree. PROJ was specifically included in the irreversible language for this proof.

2.(ii) Via PROJ.

Lemma 5. *For well-typed terms the following holds.*

(i) $\mathrm{V}\,([\![\Gamma \,;\, \Delta \vdash_\dagger e : \leftrightarrows A]\!]_\dagger \,;\, - \nearrow) = \mathrm{V}\,[\![\Gamma, \Delta \,;\, \emptyset \vdash_\dagger e : \to A]\!]_\dagger$
(ii) $(\mathrm{V}\,[\![\Gamma \,;\, \emptyset \vdash_\dagger e : \to [A \leftrightarrows B]]\!]_\dagger) \,;\, - \nearrow = \mathrm{V}\,[\![\Gamma \,;\, \emptyset \vdash_\dagger e : \to [A \to B]]\!]_\dagger$

Proof. By simultaneous induction over typing derivation.

5.(i) Checking the base case RVAR is routine expansion. We assume that $- \nearrow$ commutes with all the structure used in the semantics. The only complication is APP where we have to use 5.(ii).

5.(ii) CONST and VAR By TODO. LAM By 5.(i). LETV, APP, LETU, LETT and CURRY by induction assumption.

Theorem 1. *The following holds.*

(i) $\Gamma \,;\, \emptyset \vdash_\dagger e : \to A$ *if and only if* $\Gamma \vdash_\lambda e : A$ *and*
(ii) $\mathrm{V}\,[\![\Gamma \,;\, \emptyset \vdash_\dagger e : \to A]\!]_\dagger = [\![\Gamma \vdash_\lambda e : A]\!]_\lambda$

Proof. 1.(i) Simple induction.

1.(ii) CONST, VAR, UNIT, TUP and PROJ are straight-forward.

Theorem 2. *For well-typed terms the following holds.*

(i) $\mathrm{V}\,([\![\Gamma \,;\, \Delta \vdash_\dagger e : \leftrightarrows A]\!]_\dagger \,;\, - \nearrow) = [\![\Gamma, \Delta \vdash_\lambda e : A]\!]_\lambda$ *and*
(ii) $(\mathrm{V}\,[\![\Gamma \,;\, \emptyset \vdash_\dagger e : \to A \leftrightarrows B]\!]_\dagger) \,;\, - \nearrow = [\![\Gamma \vdash_\lambda e : A \to B]\!]_\lambda$.

Proof. By Lemma 2, Lemma 5 and Theorem 1.

2 Extending the Language

We extend the language by using Moggi's computational lambda calculus. Instead of assuming our irreversible category to be cartesian closed we assume it to be the Kleisli-category $\mathcal{C}_T$ of a strong monad (T, μ, η, t) over a cartesian closed category $\mathcal{C}$. The typing and interpration rules of Figure 5 stay unchanged only the operators are to be read as stated in Figure 6. Of course we can retain the semantics from the previous section by using the identity monad.

$$
\begin{aligned}
f \,;^{\mathcal{C}_T}\, g &= f \,;^{\mathcal{C}}\, T\, g \,;^{\mathcal{C}}\, \mu \\
\mathrm{id}^{\mathcal{C}_T} &= \eta \\
f \otimes^{\mathcal{C}_T} g &= (f \otimes^{\mathcal{C}} g) \,;^{\mathcal{C}}\, t \,;^{\mathcal{C}}\, T\, t \,;^{\mathcal{C}}\, \mu \\
\Delta^{\mathcal{C}_T} &= \Delta^{\mathcal{C}} \,;^{\mathcal{C}}\, \eta \\
!^{\mathcal{C}_T} &= !^{\mathcal{C}} \,;^{\mathcal{C}}\, \eta \\
\Lambda^{\mathcal{C}_T}\, f &= (\Lambda^{\mathcal{C}}\, f) \,;^{\mathcal{C}}\, \eta \\
\mathrm{V}^{\mathcal{C}_T}\, f &= (\mathrm{id}^{\mathcal{C}_T} \otimes^{\mathcal{C}_T} f) \,;^{\mathcal{C}_T}\, (\mathrm{V}^{\mathcal{C}}\, \mathrm{id}) \\
[\![A \to B]\!] &= A \to T\, B
\end{aligned}
$$

Fig. 6. Computational Semantics

Additionally we will extend the syntactic sugar to allow let-expressions and (reversible) function application as pattern.

$$
\begin{aligned}
[\![\mathtt{let}\, e_1\, p = e_2\, \mathtt{in}\, e_3]\!]_{\mathsf{S}} &= \mathtt{let}\, p = e_1^{\dagger}\, e_2\, \mathtt{in}\, e_3 \\
[\![\mathtt{let}\, (\mathtt{let}\, e_1 = p_1\, \mathtt{in}\, p_2) = e_2\, \mathtt{in}\, e_3]\!]_{\mathsf{S}} &= \mathtt{let}\, p_2 = e_2\, \mathtt{in}\, \mathtt{let}\, p_1 = e_1\, \mathtt{in}\, e_3
\end{aligned}
$$

With this syntactic sugar the set of terms that can appear in pattern position is actually the same as the set of terms that can appear as the body of a reversible function. We even have $[\![\Gamma\, ;\, \emptyset \vdash_{\dagger} (\lambda p.e)^{\dagger} : \to A \leftrightarrows B]\!]_{\dagger} = [\![\Gamma\, ;\, \emptyset \vdash_{\dagger} \lambda e.p : \to B \leftrightarrows A$

2.1 Pattern Matching

Let's assume the category $\mathcal{C}$ representing the irreversible world has an operation to combine two morphisms $\oplus_A \colon A \otimes A \to A$ with an identity $\mathrm{fail}_A \colon I \to A$ such that

$$
\begin{aligned}
[\![\mathtt{a}, \mathtt{b}, \mathtt{c} \vdash (\mathtt{a} \oplus \mathtt{b}) \oplus \mathtt{c}]\!]_{\lambda} &= [\![\mathtt{a}, \mathtt{b}, \mathtt{c} \vdash \mathtt{a} \oplus (\mathtt{b} \oplus \mathtt{c})]\!]_{\lambda} \\
[\![\mathtt{a} \vdash \mathrm{fail} \oplus \mathtt{a}]\!]_{\lambda} &= [\![\mathtt{a} \vdash \mathtt{a}]\!]_{\lambda} = [\![\mathtt{a} \vdash a \oplus \mathrm{fail}]\!]_{\lambda}
\end{aligned}
$$

We can use this to extend the base language with a pattern matching construct by giving the following syntactic sugar.

$$
\begin{aligned}
[\![\mathtt{case}\, e\, \mathtt{of};]\!]_{\mathsf{S}} &= \mathtt{let}\, x = e\, \mathtt{in}\, \mathrm{fail} \\
[\![\mathtt{case}\, e_0\, \mathtt{of}\, p_1 \Rightarrow e_1; \ldots\, p_n \Rightarrow e_n;]\!]_{\mathsf{S}} &= \mathtt{let}\, x = [\![e_0]\!]_{\mathsf{S}}\, \mathtt{in}\, (\mathtt{let}\, [\![p_1]\!]_{\mathsf{S}} = x\, \mathtt{in}\, [\![e_1]\!]_{\mathsf{S}}) \oplus \\
&\qquad [\![\mathtt{case}\, x\, \mathtt{of}\, p_2 \Rightarrow e_2; \ldots\, p_n \Rightarrow e_n;]\!]_{\mathsf{S}}
\end{aligned}
$$

If the reversible category $\mathcal{D}$ has a similar operator such that $\oplus_A^{\mathcal{D}} \nearrow = \oplus_A^{\mathcal{C}}$ and $\mathrm{fail}_A^{\mathcal{D}} \nearrow = \mathrm{fail}_A^{\mathcal{C}}$ then we can use the above definition unchanged to define pattern matching ambidirectionally.

The Janus category $\mathsf{J}\mathcal{C}$ inherits $\oplus$ from $\mathcal{C}$ as follows.

$$\begin{aligned}\oplus^{\mathsf{J}\mathcal{C}} &= (f\nearrow \oplus^{\mathcal{C}} g\nearrow, f\searrow \oplus^{\mathcal{C}} g\searrow)\\ \mathrm{fail}^{\mathsf{J}\mathcal{C}}_{A,B} &= (\mathrm{fail}^{\mathcal{C}}_{A,B}, \mathrm{fail}^{\mathcal{C}}_{B,A})\end{aligned}$$

For other dagger categories an appropriate operator is harder to define. For $\mathsf{core}\mathcal{C}$ for example we need to ensure that $(f \oplus g)\nearrow ; (f \oplus g)\searrow = \mathrm{id}$. With the above laws we can show that

$$\begin{aligned}(f \oplus g)\nearrow ; (f \oplus g)\searrow &= (f\nearrow \oplus g\nearrow) ; (f\searrow \oplus g\searrow)\\ &= f\nearrow ; f\searrow \oplus f\nearrow ; g\searrow \oplus g\nearrow ; f\searrow \oplus g\nearrow ; g\searrow\end{aligned}$$

but f and g have no relationship with each other in general.

A strategy to deal with this problem is the so called symmetric first-match policy as proposed in CITE. Here we deal with partial function so the semantics lives in the Maybe-monad. We will write $\mathcal{C}_{+1}$ for the Kleisli-category of $\mathcal{C}$ over the Maybe-monad. Let $-\downarrow : A \to \mathsf{Bool}$ be the function that returns true for all pure values and false otherwise. With these ideas in mind we define a homset monoid in $\mathsf{J}\mathcal{C}_{+1}$ as follows.

$$f \oplus^{\mathsf{J}\mathcal{C}_{+1}} g = (\mathsf{dup}_{\mathsf{Bool}}\, \lambda a.(f\nearrow a)\downarrow)^{\dagger} ; (\mathrm{jV}\, \mathsf{sel}_{[AjB]}\, (f,g)) ; (\mathsf{dup}_{\mathsf{Bool}}\, \lambda b.(f\searrow b)\downarrow)$$

This generalized the approach in the paper from just certain parts of case-expressions to all expressions to allow for a more succinct presentation.

The problem here is that we have $(f \oplus^{\mathsf{J}\mathcal{C}_{+1}} g)\nearrow \neq f\nearrow \oplus^{\mathcal{C}_{+1}} g\nearrow$ which so far has been fundamental to our approach. And since the symmetric first-match policy must refer to the reverse direction of f there is little hope to find a compatible homset-monoid for $\mathcal{C}_{+1}$ where there is no reverse direction.

There is hope, though. Any idempotent monoid induces a partial order as follows.

$$f \leq g \iff f \oplus g = g = g \oplus f$$

Lemma 6. $(f \oplus^{\mathsf{J}\mathcal{C}_{+1}} g)\nearrow \leq f\nearrow \oplus^{\mathcal{C}_{+1}} g\nearrow$

Proof.

There have been several strategies proposed in the literature. One way is to ensure at compile-time that the mixed compositions cannot appear. Another strategy is to check at run-time that the result of the compound function will choose the same function in the opposite direction.

$$f \oplus_j g = (\mathsf{fgt}_{\mathsf{Bool}}\, f\nearrow \downarrow)^{\dagger} ; (\mathrm{jV}\, \mathsf{sel}_{[AjB]}\, (f,g)) ; (\mathsf{fgt}_{\mathsf{Bool}}\, f\searrow \downarrow)$$

Lemma 7. $(f \oplus g) \oplus h = f \oplus (g \oplus h)$

2.2 Duplicating and Discarding Information

Definition 6. *Monad* $T\,X = X+1$, $\eta_A = \iota_1$, $\mu_A = [id_{T\,A}, \iota_2]$, $t_{A,B} =$. $[\![\mathsf{fail}_{A,B}]\!] : [A \leftrightarrows B] = \iota_2$

$$\begin{aligned}
\mathsf{fgt}_A &: A \to [A \leftrightarrows I] \\
[\![a \vdash (\mathsf{fgt}_A\ a)\nearrow]\!]_\dagger &= [\![\emptyset \vdash \lambda a\text{'}.()]\!]_\lambda \\
[\![a \vdash (\mathsf{fgt}_A\ a)\searrow]\!]_\dagger &= [\![\emptyset \vdash \lambda().a]\!]_\lambda
\end{aligned}$$

$$\begin{aligned}
\mathsf{dup}_A &: A \to [I \leftrightarrows A] \\
[\![a \vdash (\mathsf{dup}_A\ a)\nearrow]\!]_\dagger &= [\![a \vdash \lambda a\text{'}.\mathtt{if}\ a\text{'} = a\ \mathtt{then}\ ()\ \mathtt{else}\ \mathsf{fail}\ ]\!]_\lambda \\
[\![a \vdash (\mathsf{dup}_A\ a)\searrow]\!]_\dagger &= [\![a \vdash \lambda().a]\!]_\lambda
\end{aligned}$$

2.3 Recursion

If $\mathcal{C}$ admits fixpoints $\mathsf{fix}_A : [A \to A] \to [I \to A]$ with $\mathsf{fix}_A\, f\,; f = \mathsf{fix}_A\, f$ then we can immediately define recursive reversible morphisms in $\mathcal{D}$ by setting $A = [B \leftrightarrows C]$.

3 Examples

3.1 Dagger Symmetric Monodial Isos

$$\begin{aligned}
\mathit{rev} &= \lambda f.\lambda f\ x.x \\
\mathit{assoc} &= \lambda((a,b),c).(a,(b,c)) \\
\mathit{swap} &= \lambda(a,b).(b,a) \\
\mathit{unitl} &= \lambda a.((),a) \\
\mathit{unitr} &= \lambda a.(a,())
\end{aligned}$$

3.2 Lists

Assume for every type τ there is a type τ^* with constant symbols $\mathsf{nil}_\tau : 1 \leftrightarrows \tau^*$ and $\mathsf{cons}_\tau : \tau \times \tau^* \leftrightarrows \tau^*$ and having computable equality if τ has computable equality. We can write functions for appending to a list and reversing a list.

$$\begin{aligned}
\mathit{append} = \ & \lambda(\mathsf{nil}\,(),x).\mathsf{cons}\,(x,\mathsf{nil}\,()) \\
& |\lambda(\mathsf{cons}\,(y,r),x).\mathsf{cons}\,(y,\mathit{append}\ (r,x)) \\
\mathit{reverse} = \ & \lambda\mathsf{nil}\,().\mathsf{nil}\,() \\
& |\lambda\mathsf{cons}\,(x,r).\mathit{append}\ (x,\mathit{reverse}\ r)
\end{aligned}$$

We can even define a right-fold.

$$\begin{aligned}\mathit{foldr} &= \lambda f.\ \lambda(\mathsf{cons}\,(y, r), x).\mathit{foldr}\ f\ (r, f\ (y, x))\\ &\quad |\lambda(\mathsf{nil}\,(), x).x\end{aligned}$$

$\mathit{foldr}\ f$ establishes a relationship between $A^* \times B$ and B using a suitable f. Through pattern matching the values $b\colon B$ such that $f^\dagger\, b$ is not defined are the ground values.

The naïve left-fold definition is not useful unfortunately. The reverse direction would recurse indefinitely.

It is possible a define $\mathit{foldl} = \lambda f.\lambda(x, l).\mathit{foldr}\,(\mathsf{swap}\,;f)\,(\mathit{reverse}\ l, x)$ though.

3.3 Integer parser

4 Related Work

4.1 Lenses

Definition 7. *A* lens l *between types* A *and* B *is a pair of functions* $l.\mathsf{get}\colon A \to B$ *and* $l.\mathsf{put}\colon B \times A \to A$*. It is called* well-behaved *if*

$$l.\mathsf{put}\,(l.\mathsf{get}\,a, a) = a \qquad\qquad l.\mathsf{get}\,(l.\mathsf{put}\,(a, b)) = b\ .$$

Lenses are closed under composition and we can define an identity lens so they form a category enriched over Set. Let $\mathsf{L}\mathcal{C}$ be the category of lenses formed by appropriate pairs of morphisms from a suitable category $\mathcal{C}$. Because any isomorphism in $\mathcal{C}$ naturally induces a lens we find that $\mathsf{L}\mathcal{C}$ inherits the symmetric monoidal structure from $\mathcal{C}$. Unfortunately lenses are asymmetrical so we don't have a dagger category.

Defining a jV is difficult. The problem is that when jV transforms a lens-builder $f\colon C \to [A \xrightarrow{\mathsf{L}\mathcal{C}} B] - \otimes C$ is applied to both A and B but A appears twice in the signature of put. So $(\mathrm{jV}\,f).\mathsf{put}$ is of type $[(B \otimes C) \otimes (A \otimes C) \to A \otimes C]$. We can remedy this by using the following isomorphic representation of well-behaved lenses over Set.

Lemma 8. *Existentially quantified pairs of inverse morphisms* $\exists R.(A \to B \times R) \times (B \times R \to A)$ *are isomorphic to well-behaved lenses between* A *and* B*.*

Proof. ($\Longrightarrow$) Let $(f{\nearrow}, f{\searrow})\colon \exists R.(A \to B \times R) \times (B \times R \to A)$ be such that $f{\nearrow}\,; f{\searrow} = \mathrm{id}$ and $f{\searrow}\,; f{\nearrow} = \mathrm{id}$. Then we define a lens l with

$$l.\mathsf{get}\,a = \pi_1\,(f{\nearrow}\,a) \qquad\qquad l.\mathsf{put}\,(b, a) = f{\searrow}\,(b, \pi_2\,(f{\nearrow}\,a))$$

Now

$$\begin{aligned}&l.\mathsf{put}\,(l.\mathsf{get}\,a, a)\\ =&f{\searrow}\,(\pi_1\,(f{\nearrow}\,a), \pi_2\,(f{\nearrow}\,a))\\ =&f{\searrow}\,(f{\nearrow}\,a)\\ =&a\end{aligned} \qquad \begin{aligned}&l.\mathsf{get}\,(l.\mathsf{put}\,(b, a))\\ =&\pi_1\,(f{\nearrow}\,(f{\searrow}\,(b, (\pi_2\,(f{\nearrow}\,a)))))\\ =&\pi_1\,(b, (\pi_2\,(f{\nearrow}\,a)))\\ =&b\end{aligned}$$

($\Longleftarrow$) Let l be a well-behaved lens between A and B. Let $R_b = l.\mathsf{put}\,(b, A) \subseteq A$, $p_b \colon A \to R_b = l.\mathsf{put}\,(b, -)$ and $q_b \colon R_b \to A$ the injection of R_b in A. Observe that $p_b\,;q_b = l.\mathsf{put}\,(b, -)$ and $q_b\,;p_b = \mathrm{id}_{R_b}$. Define $(f\nearrow, f\searrow) \colon \exists R.(A \to B \times R) \times (B \times R \to A)$ with

$$f\nearrow = \lambda a.\mathtt{let}\ b = l.\mathsf{get}\,a\ \mathtt{in}\ (b, p_b\,a)$$
$$f\searrow = \lambda(b, r).l.\mathsf{put}\,(b, q_b\,r)$$

Now

$$\begin{aligned} & f\searrow(f\nearrow a) \\ =& \mathtt{let}\ b = l.\mathsf{get}\,a\ \mathtt{in}\ l.\mathsf{put}\,(b, q_b\,(p_b\,a)) \\ =& \mathtt{let}\ b = l.\mathsf{get}\,a\ \mathtt{in}\ l.\mathsf{put}\,(b, l.\mathsf{put}\,(b, a)) \\ =& \mathtt{let}\ b = l.\mathsf{get}\,a\ \mathtt{in}\ l.\mathsf{put}\,(b, a) \\ =& a \end{aligned}$$

$$\begin{aligned} & f\nearrow(f\searrow(b, r)) \\ =& \mathtt{let}\ a = l.\mathsf{put}\,(b, q_b\,r)\ \mathtt{in} \\ & \quad \mathtt{let}\ b' = l.\mathsf{get}\,a\ \mathtt{in}\ (b', p_{b'}\,a) \\ =& \mathtt{let}\ b' = b\ \mathtt{in}\ (b', p_{b'}\,(l.\mathsf{put}\,(b, q_b\,r))) \\ =& (b, p_b\,(l.\mathsf{put}\,(b, q_b\,r))) \\ =& (b, p_b\,(q_b\,(p_b\,(q_b\,r)))) \\ =& (b, r) \end{aligned}$$

Remark 1. The above proof works for any category that has existential quantification and is *idempotent complete*. A category is idempotent complete if for all morphisms $f \colon A \to A$ with $f\,;f = f$ there exists an object R so that f can be split into two morphisms $g \colon A \to R$ and $h \colon R \to A$ such that $g\,;h = f$ and $h\,;g = \mathrm{id}_R$.

Lemma 9. *Morphism pairs of the form $\exists R.(A \to B \times R) \times (B \times R \to A)$ form a symmetric monoidal category.*

Proof. Composition is given by a morphism in the janus category

$$\begin{aligned} -\,;- &= [\![\boldsymbol{f}, \boldsymbol{g}; \boldsymbol{a} \vdash \mathtt{let}\,(\boldsymbol{b}, \boldsymbol{rf}) = \boldsymbol{f}\ \boldsymbol{a}\ \mathtt{in}\ \mathtt{let}\,(\boldsymbol{c}, \boldsymbol{rg}) = \boldsymbol{g}\ \boldsymbol{b}\ \mathtt{in}\,(\boldsymbol{c}, (\boldsymbol{rf}, \boldsymbol{rg}))]\!]_\dagger \\ &= \lambda(f, g).f\,;^{\mathsf{J}\mathcal{C}}\,(g \otimes^{\mathsf{J}\mathcal{C}} \mathrm{id}) \\ &= \lambda(f, g).\begin{pmatrix} \lambda a.\mathsf{let}\,(b, r_f) = f\nearrow a\ \mathsf{in}\ \mathsf{let}\,(c, r_g) = g\nearrow b\ \mathsf{in}\,(c, (r_f, r_g)) \\ \lambda(c, (r_f, r_g)).f\searrow(g\searrow(c, r_g), r_f) \end{pmatrix} \end{aligned}$$

so $R_{f\,;g} = R_f \otimes R_g$. Associativity is proven by associativity of $-\,;-$ and $-\otimes-$.

For isomorphisms in $\mathcal{C}$ we find that we can set $R = I$. So the identities and the symmetric monoidal structure is isomorphic to that in $\mathsf{J}\mathcal{C}$.

Definition 8. *Define the category $\mathsf{L}_\exists\mathcal{C}$ having the same objects as $\mathcal{C}$ and as pairs of inverse morphisms of type $\exists R.(A \to B \times R) \times (B \times R \to A)$ as in Lemma 9.*

Define $-.\mathsf{get} \colon \mathsf{L}_\exists\mathcal{C} \to \mathcal{C}$ as an identity on objects functor with $(f\nearrow, f\searrow).\mathsf{get} = \pi_1\,(f\nearrow a)$ as in Lemma 8.

Define $\mathrm{jV}^{\mathsf{L}_\exists\mathcal{C}}\,f = \begin{pmatrix} \lambda(a, c).\mathsf{let}\,(b, r) = (f\,c)\nearrow a\ \mathsf{in}\ ((b, c), r) \\ \lambda((b, c), r).(f\,c)\searrow(b, r) \end{pmatrix}$.

Theorem 3. $(\mathcal{C}, \mathsf{L}_\exists\mathcal{C}, -.\mathsf{get}, \mathrm{jV}^{\mathsf{L}_\exists\mathcal{C}})$ *is a* jV*-structure.*

Proof. That $-.\mathsf{get}$ is a symmetric monoidal functor is immediate from Lemma 9.

To show that $\mathrm{jV}^{\mathsf{L}_\exists \mathcal{C}}\, f$ is indeed a morphism of $\mathsf{L}_\exists \mathcal{C}$ observe that $\mathrm{jV}^{\mathsf{L}_\exists \mathcal{C}}\, f = (\mathrm{jV}^{\mathsf{J}\mathcal{C}}\, f)\,;^{\mathsf{J}\mathcal{C}}\, (\mathrm{id} \otimes \mathrm{swap})$. Now with the same argument as in Example 3 we have

$$\begin{aligned}
&((\mathrm{jV}^{\mathsf{L}_\exists \mathcal{C}}\, f)\nearrow\,;^{\mathcal{C}}\, (\mathrm{jV}^{\mathsf{L}_\exists \mathcal{C}}\, f)\searrow, (\mathrm{jV}^{\mathsf{L}_\exists \mathcal{C}}\, f)\searrow\,;^{\mathcal{C}}\, (\mathrm{jV}^{\mathsf{L}_\exists \mathcal{C}}\, f)\nearrow)\\
=&(\mathrm{jV}^{\mathsf{J}\mathcal{C}}\, f)\,;^{\mathsf{J}\mathcal{C}}\, (\mathrm{id} \otimes \mathrm{swap})\,;^{\mathsf{J}\mathcal{C}}\, (\mathrm{id} \otimes \mathrm{swap})^\dagger\,;^{\mathsf{J}\mathcal{C}}\, (\mathrm{jV}^{\mathsf{J}\mathcal{C}}\, f)^\dagger\\
=&\mathrm{id}
\end{aligned}$$

Finally we show that $\mathrm{jV}^{\mathsf{L}_\exists \mathsf{Set}}$ commutes with $-.\mathsf{get}$.

$$\begin{aligned}
(\mathrm{jV}^{\mathsf{L}_\exists \mathcal{C}}\, f).\mathsf{get} &= (\lambda(a,c).\mathsf{let}\,(b,r) = (f\,c)\nearrow a\,\mathsf{in}\,((b,c),r))\,;\pi_2\\
&= (\lambda(a,c).((f\,c)\nearrow, c)\\
&= \mathrm{jV}^{\mathcal{C}}\,(f\,;\,-\nearrow)
\end{aligned}$$

So with Emordnilap we can write lenses in a point-full style and by Theorem 2 the resulting get-function will behave exactly as if the program was interpreted as a normal irreversible functional program. We can even extend the language to allow the discarding of variables as shown in Figure 7.

Name	Typing	Semantics ($[\![-]\!]_\dagger$)
Term	$\Gamma\,;\,\Delta \vdash_\dagger e : \leftrightarrows A$	$= g$
	$\Gamma\,;\,\Delta, \Delta' \vdash_\dagger e : \leftrightarrows A$	$= [\![\gamma \vdash \mathrm{id}_{[\![\Delta]\!]} \otimes\, !_{[\![\Delta']\!]}]\!]_\lambda$
		where $!_A = \begin{pmatrix} \lambda a.((), a) \\ \lambda((), a).a \end{pmatrix}$

Fig. 7. Typing and semantics of discarding information for lenses

4.2 Traces, Hylomorphisms

4.3 RFun, Janus, Applicative

5 Conclusion and Future Work