There and Back Again *

Arrows for Invertible Programming

Artem Alimarine Sjaak Smetsers Arjen van Weelden
Marko van Eekelen Rinus Plasmeijer

Institute for Computing and Information Sciences,
Radboud University Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands.

A.Alimarine@cs.ru.nl S.Smetsers@cs.ru.nl A.vanWeelden@cs.ru.nl
M.vanEekelen@cs.ru.nl R.Plasmeijer@cs.ru.nl

Abstract

Invertible programming occurs in the area of data conversion where it is required that the conversion in one direction is the inverse of the other. For that purpose, we introduce bidirectional arrows (*bi-arrows*). The bi-arrow class is an extension of Haskell's arrow class with an extra combinator that changes the direction of computation.

The advantage of the use of bi-arrows for invertible programming is the preservation of invertibility properties using the bi-arrow combinators. Programming with bi-arrows in a polytypic or generic way exploits this the most. Besides bidirectional polytypic examples, including invertible serialization, we give the definition of a monadic bi-arrow transformer, which we use to construct a bidirectional parser/pretty printer.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Algorithms

Keywords Haskell, Arrows, Invertible program construction, Polytypic programming.

1. Introduction

Arrows [11] are a generalization of monads [21]. Just as monads, arrows provide a set of combinators. They make it possible to combine functions in a very general way. In principle, the combinators assume very little about the functions to combine. In fact, these functions may even comprise side-effects. The main application areas of arrows are in the field of interactive programming and data conversion. More specifically, extensive applications have been made in the areas of user interfaces [3], reactive programming [9], and parser combinators [13].

For the general area of data conversion, it may be important to prove invertibility of a specified algorithm. This is, for instance, directly the case in encryption, serialization, marshalling, compression, and parsing but also more indirectly in the area of data base transactions where roll-backs may have to be performed.

The goal of our work is to set up an arrow-based framework for the specification of invertible algorithms. We start with extending the monotypic unidirectional framework of arrows to a monotypic bidirectional framework of bidirectional arrows, *bi-arrows*.

In particular, we represent a pair of conversion functions as a single arrow, such that we can specify both conversion functions by one definition. The advantage of such a single definition is that it reduces the amount of code needed for each conversion pair, because more code can be reused from the arrow library. Basically, one specifies the conversion in one direction (usually the more involved case) and one gets the inverse conversion almost for free. For instance, by specifying a parser one also specifies the pretty printer. The price to pay is that specifying the parser becomes a bit more complicated.

The advantages of programming with arrows and inversion are exploited best in a polytypic or generic framework. Therefore, we extend our monotypic bidirectional framework to the polytypic context. In this context we show how to define several essential combinators and bi-arrow transformers. We give several smaller polytypic examples including invertible (de)serialization. We also discuss how this can be done for the larger example of parsers and pretty-printers.

More specifically, the contributions of this paper are the following.

- We extend the framework of arrows to support *bidirectional arrows*.
- Our approach explicitly uses *embedding-projection arrows*.
- Our approach is suitable for *monotypic and polytypic* conversion functions.
- We show how to define pairs of conversion functions together in one single definition. We show that specifying one direction of conversion also specifies the other direction. We present several monotypic and polytypic *examples* of such definitions.

We use the pure lazy functional language Haskell [17] in our examples. Polytypic examples use Generic Haskell [14], the generic programming extension for Haskell. The code can be downloaded from `http://www.cs.ru.nl/A.vanWeelden/bi-arrows/`. The work can just as easily be expressed in Clean [18] using its built-in generics [1]. We assume general knowledge of arrows and polytypic programming, and we will only briefly recall relevant definitions and techniques.

* Shamelessly stolen from the Lord of the Rings (the book, not the movie).

Haskell'05 September 30, 2005, Tallinn, Estonia.

The next section (section 2) introduces bidirectional arrow combinators. A small *monotypic* invertible program example is given in section 3. This is done by using *embedding-projection arrows*, which are also introduced in that section.

In section 4 the framework is used in a polytypic context and we introduce invertible arrows with state. We present *polytypic traversals* (mappings) on bi-arrows and *state arrows*. These state arrows are used in section 5 to create a somewhat larger example performing (de)serialization of data, based on the structure of a type.

Section 6 introduces monadic programming with bi-arrows. Ways to deal with failure in bi-arrows are introduced and a method to lift monads to bi-arrows is given. An application of bi-arrows, consisting of a parser and a pretty-printer, is created in section 7. The example uses a combination of state, monadic, and embedding-projection arrows.

Finally, section 8 discusses related work and section 9 concludes and mentions prospects for future work.

2. From arrows to bidirectional arrows

This section introduces a bidirectional framework that consists of a set of reversible arrow combinators. These combinators are based on the arrow combinators defined by Hughes [11].

First, we will recall shortly the standard arrow framework (section 2.1). Then we show how these laws have to be adapted for our dyadic bi-arrows framework (section 2.2). Finally, we give specific inversion laws for bi-arrows (section 2.4). In section 3 we show how bidirectional arrows are constructed using a small motivating example.

2.1 Arrows

We briefly recall Hughes's definitions expressed in Haskell as a type constructor class.

```
class Arrow arr where
    arr    :: (a → b) → arr a b              -- pure
    (⋙)    :: arr a b → arr b c → arr a c   -- infixr 1
    first  :: arr a b → arr (a, c) (b, c)
    second :: arr a b → arr (c, a) (c, b)
    (***)  :: arr a c → arr b d →
                           arr (a, c) (b, d)  -- infixr 3
```

As usual, the definition of *** and second can be expressed in terms of first (corresponding to Haskell's default definition of *** and second):

```
f *** g  = first f ⋙ second g
second f = arr swap ⋙ first f ⋙ arr swap

swap = snd `split` fst
split f g = λt → (f t, g t)
```

To allow case distinction Hughes shows that a new combinator is needed. He, therefore, introduces the *choice arrow*:

```
class Arrow arr ⇒ ArrowChoice arr where
    left  :: arr a b → arr (Either a c) (Either b c)
    right :: arr b c → arr (Either d b) (Either d c)
    (+++) :: arr a c → arr b d →
                arr (Either a c) (Either b d)  -- infixr 2
```

As with *** and second, +++ and right can be expressed in terms of left, and Haskell's prelude function either:

```
f +++ g = left f ⋙ right g
right f = arr mirror ⋙ left f ⋙ arr mirror

mirror = Right `either` Left
```

By instantiating the arrow class for → we can use ordinary functions as arrows.

```
instance Arrow (→) where
    arr f   = f
    f ⋙ g = g . f
    first f = f <*> id

instance ArrowChoice (→) where
    left f = f <+> id
```

Here <*> and <+> are the usual product and sum operations for functions:

```
(<*>) :: (a → b) → (c → d) → (a, c) → (b, d)
f <*> g = (f . fst) `split` (g . snd)

(<+>) :: (a → b) → (c → d) →
           Either a c → Either b d
f <+> g = (Left . f) `either` (Right . g)
```

In literature [11, 15, 16], one can find several other combinators and also some derived combinators that make programming with arrows easier, such as:

```
(⋘) :: Arrow arr ⇒
          arr c b → arr b a → arr c a  -- infixl 1
f ⋘ g = g ⋙ f
```

Here, we refrain from giving an exhaustive overview.

2.2 Bidirectional arrows

To support invertibility, we extend the arrows with two new combinators: ↔ (*biarr*/*bipure*) and inv (*inverse*).

The first one, ↔, is similar to the standard arr but instead of a single function it takes *two* functions and lifts them into a bidirectional arrow (bi-arrow) creating a structure that contains them both. The intention is that these functions are each others inverse. The second one, inv, reverses the direction of computation, yielding the inverse of a bi-arrow, which will boil down to swapping the two comprised functions.

```
class Arrow arr ⇒ BiArrow arr where
    (↔) :: (a → b) → (b → a) → arr a b  -- infix 8
    inv  :: arr a b → arr b a
```

We define BiArrow on top of the Arrow class because conceptually bi-arrows form an extension of the arrow class. Moreover, it allows us to use bi-arrows as normal arrows. Since the derived combinators second and right use the arr constructor to build the adapters swapA and mirrorA we have to redefine them using ↔ to make these combinators invertible. Therefore, we introduce:

```
secondA f = swapA ⋙ first f ⋙ swapA
                     where swapA   = swap ↔ swap
rightA f  = mirrorA ⋙ left f ⋙ mirrorA
                     where mirrorA = mirror ↔ mirror
arrA f    = f ↔ const (error "arr has no inverse")
```

where swap and mirror are defined as above.

2.3 Arrow laws for bi-arrows

To reason about programs containing arrow combinators we can use properties that are specific to arrows, the so-called *arrow laws*. The collection of arrow laws is not uniquely defined. The laws we have taken are a subset of the ones postulated by Hughes [11].

We need some adaptation of the laws for our framework. The occurrences of arr f are replaced with the corresponding dyadic operator for bi-arrows: f ↔ g where g is intended to be the inverse of f.

Definition 1 (Composition Laws)

$$
\begin{array}{rcl}
f \ggg (g \ggg h) & = & (f \ggg g) \ggg h \\
f_1 \leftrightarrow g_2 \ggg g_1 \leftrightarrow f_2 & = & (f_1 \ggg g_1) \leftrightarrow (f_2 \ggg g_2) \\
idA \ggg f & = f = & f \ggg idA
\end{array}
$$
where
  $idA = id \leftrightarrow id$

Definition 2 (Pair Laws)

$$
\begin{array}{rcl}
first\ (f \ggg g) & = & first\ f \ggg first\ g \\
first\ (f \leftrightarrow g) & = & (f \star id) \leftrightarrow (g \star id) \\
first\ h \ggg (id \star f) \leftrightarrow (id \star g) & = & (id \star f) \leftrightarrow (id \star g) \ggg first\ h \\
first\ (first\ f) \ggg assocPA & = & assocPA \ggg first\ f
\end{array}
$$
where
$$
\begin{array}{rcl}
assocPA & = & assoc \leftrightarrow cossa \\
assoc\ ((x,y),z) & = & (x,(y,z)) \\
cossa\ (x,(y,z)) & = & ((x,y),z)
\end{array}
$$

In categorial terms, the product type is the dual of the sum type. In general, if a property holds for products, the dual property is valid for sums. The dual is obtained by systematically replacing *split* by *either*, *Left*/*Right* by *fst*/*snd*, *first* by *left*, $\ggg$ by $\lll$, and $f \circ g$ by $g \circ f$. For example, taking the dual of the last product law leads to the following sum law

$$left\ (left\ f) \lll assocSA = assocSA \lll left\ f$$

To obtain the dual *assocSA* of *assocPA* we first express *assoc* and *cossa* in terms of `split`, `fst` and `snd`.

$$
\begin{array}{rcl}
assoc & = & (fst \circ fst)\ \text{'split'}\ ((snd \circ fst)\ \text{'split'}\ snd) \\
cossa & = & (fst\ \text{'split'}\ (fst \circ snd))\ \text{'split'}\ (snd \circ snd)
\end{array}
$$

Now the transformation leads to $assocSA = assocS \leftrightarrow cossaS$, where

$$
\begin{array}{l}
assocS = (Left \circ Left)\ \text{'either'}\ ((Left \circ Right)\ \text{'either'}\ Right) \\
cossaS = (Left\ \ \text{'either'}\ (Right \circ Left))\ \text{'either'}\ (Right \circ Right)
\end{array}
$$

Note that *right* is also the dual of *second*, since *mirror* is the dual of *swap*.

Using the laws above several properties can be proven easily. For example, $first\ idA = idA = second\ idA$ is proven by substituting the definitions for `first` and `second` taken from section 2.1 and applying the appropriate laws for *first* and $\ggg$.

2.4 Inversion Laws

Most importantly, implementations of bi-arrows are proper if they satisfy some additional inversion laws.

Definition 3 (Inversion Laws)

$$
\begin{array}{rcl}
inv\ (inv\ f) & = & f \\
inv\ (f \ggg g) & = & inv\ g \ggg inv\ f \\
inv\ (f \leftrightarrow g) & = & g \leftrightarrow f \\
inv\ (first\ f) & = & first\ (inv\ f) \\
inv\ (left\ f) & = & left\ (inv\ f)
\end{array}
$$

The last two rules are only appropriate for arrows that are pure functions. In a more general case, where arrows can have side-effects (e.g., when monads with internal side effects are lifted to bi-arrows), it is required that, instead of *first* and *left*, *cofirst* and *coleft* respectively are used. These ‘inverse combinators’ are the categorical duals of *first* and *left*. They are needed to revert possible side-effects of *first* and *left*. Throughout the rest of this paper all arrows will be pure. Hence, we will use the rules above since they are sufficient for this paper. Nevertheless, for the rest of the framework no assumptions will be made on the absence of side-effects.

Of course, when introducing a new instance for one of the arrow classes defined above we have to guarantee that all the corresponding laws hold. We say that f is a *bi-arrow* if the composition, pair and inverse laws hold. Let f be an bi-arrow. Then f is *invertible* if

$$inv\ f \ggg f = idA = f \ggg inv\ f$$

The essence of our framework is that invertibility is preserved by our (bi-)arrow combinators. We are working on finishing the details of the formal proof of this property, using the various bi-arrow laws. It will be presented in a separate paper. The emphasis of this paper will be on introducing the framework and on its applications.

3. Monotypic programming with bi-arrows

The idea of using bi-arrows is that after specifying an operation in one direction one gets the inverse of this operation (in the opposite direction).

In this section we first discuss how to create an invertible definition using the bi-arrow definitions (section 3.1). Then, we discuss the inherent differences between functions and bi-arrows (section 3.2). This motivates why we introduce a structure that contains both functions (section 3.3). Finally, we discuss some problems with the use of Paterson notation for bi-arrows (section 3.4).

3.1 A motivating example

How easy or difficult is it to define functions by means of the arrow constructors? In this section we will give an example. Of course, one has to keep in mind that some functions are not easily invertible. Take, for instance, a simple function like $+\!\!+$ (append), which concatenates two lists. It is clear that the inverse cannot be a function with the same type, since in general there are many ways to split a list into two parts.

An example of a function that does have an (obvious) inverse is `reverse`. We take the standard definition as starting point to get an arrow based version. We could have lifted `reverse` to a bi-arrow using `reverse` ↔ `reverse`, but this does not illustrate the concerns of bidirectional programming.

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Case distinction, using arrows, is done by using `left` and `right`, which means that we first have to tag the input with `Left` or `Right`, indicating the empty and non-empty list respectively. Tagging and untagging are done by applying the following bi-arrow, which forms an isomorphic mapping from lists to `Either`s.

```
list2EitherA :: BiArrow arr ⇒
                arr [a] (Either () (a, [a]))
list2EitherA = list_either ↔ either_list
  where
    list_either  []     = Left  ()
    list_either  (x:xs) = Right (x, xs)

    either_list (Left ())       = []
    either_list (Right (x, xs)) = x:xs
```

Now we can give the arrow version of `reverse`: `reverseA`.

```
reverseA :: (ArrowChoice arr, BiArrow arr) ⇒
            arr [a] [a]
reverseA = list2EitherA
         ⋙ right (second reverseA ⋙ appElemA)
         ⋙ inv list2EitherA
```

Here appElemA is an adjusted version of append that takes one element and attaches it to the end of a list. If one specifies invertible arrows it appears to convenient to use ‘symmetrical‘ versions, i.e., arrows that handle the argument and the result symmetrically. This leads to the following definition of appElemA. We will give an example of its usage later in this section.

```
appElemA :: (ArrowChoice arr, BiArrow arr) ⇒
            arr (a, [a]) (a, [a])
appElemA = second list2EitherA ⋙ liftRSA
         ⋙ right (swapXYA ⋙ second appElemA)
         ⋙ inv (second list2EitherA ⋙ liftRSA)
```

The auxiliary arrow liftRSA converts a product–of–sum into a sum–of–product, and swapXYA exchanges the x and y field of a nested pair. The last one is defined in terms of assocPA and swapA introduced in section 2.

```
liftRSA :: BiArrow arr ⇒
           arr (a, Either b c) (Either (a, b) (a, c))
liftRSA = liftr ↔ rtfil
    where
        liftr (x, Left y)  = Left (x, y)
        liftr (x, Right y) = Right (x, y)

        rtfil (Left (x, y))  = (x, Left y)
        rtfil (Right (x, y)) = (x, Right y)

swapXYA :: BiArrow arr ⇒ arr (a, (b, c)) (b, (a, c))
swapXYA = inv assocPA ⋙ first swapA ⋙ assocPA
```

3.2 Functions are not bi-arrows

Although ReverseA is constructed to be invertible, we cannot use the inverse of reverse using the → instance for arrows. This means that the following will not work:

```
(inv reverseA) [1, 2, 3] −− this is a compile time error
```

This is caused by an absence of an instance of BiArrow for →. Since ReverseA itself depends on the BiArrow class, we even cannot write

```
reverseA [1, 2, 3]          −− this is also a compile time error
```

There is no sensible way to define an instance of BiArrow for →. Of course, one *could* define ↔ for functions by dropping the second argument, however, this instance only works in one direction. For the last two examples this would mean that we would not get a compile-time error anymore. Instead we would get the correct result for the latter expression, but evaluation of the first one would result in a run-time error.

3.3 The embedding-projection bi-arrow transformer

We can circumvent this problem by handling inversion explicitly via *embedding-projection* (*EP*) *pairs*. See, for instance, [8]. We generalize the embedding-projections from pairs of functions to be pairs of arrows. This makes EpT an *arrow transformer*, i.e., it enables us to construct bi-arrows on top of existing arrows (particularly functions). Therefore, our type for embedding projections is parameterized with an arrow:

```
data EpT arr a b = Ep {toEp :: arr a b,
                       fromEp :: arr b a}
```

The instances of the (bi-)arrow classes can be defined straightforwardly.

```
instance Arrow arr ⇒ Arrow (EpT arr) where
    arr     = arrA
    f ⋙ g = Ep (toEp f ⋙ toEp g)
                (fromEp g ⋙ fromEp f)
    first f = Ep (first (toEp f)) (first (fromEp f))
    second  = secondA

instance ArrowChoice arr ⇒
        ArrowChoice (EpT arr) where
    left f = Ep (left (toEp f)) (left (fromEp f))
    right  = rightA

instance Arrow arr ⇒ BiArrow (EpT arr) where
    f ↔ g = Ep (arr f) (arr g)
    inv f  = Ep (fromEp f) (toEp f)
```

To ensure the invertibility preserving property of the EpT bi-arrow transformer, one should not use the arr because an arrow constructed with arr has no inverse. We still define the arr function for EpT, in terms of the ↔ and error (using arrA from the previous section) to give a more informative run-time error and to support normal arrow operations.

By adding toEp to the example, we can force the use of the instance for the (EpT →) arrow:

```
toEp reverseA [1, 2, 3]          −− yields [3, 2, 1]
toEp (inv reverseA) [1, 2, 3]    −− yields [3, 2, 1]
```

In the same way, we can show an example of appElemA.

```
toEp appElemA (4, [1, 2, 3])    −− yields (1, [2, 3, 4])
```

3.4 Paterson notation

The example from the previous section clearly shows that, without any support, programming with arrow combinators can be quite complicated.

The notation for arrows as proposed by Paterson [15] can be helpful because it relieves the programmer from defining a lot of small adaptor arrows. For example, the definition of appElemA using this arrow notion becomes:

```
appElemA = proc (e, xs) → case xs of
             [ ]      → returnA ≺ (x, e)
             (x:xs) → do
                        (h, t) ← appElemA ≺ (e, xs)
                        returnA ≺ (x, h:t)
    where returnA = arr id
```

Unfortunately, this syntactic sugar for arrows does not support invertibility. The translation scheme, as described in [15], uses unidirectional adaptors that cannot easily be made bidirectional. The (internal) adaptors are unidirectional, since they are defined using arr instead of ↔. This is similar to the problem we encountered defining bi-arrows as an extension of the original arrow class (the default second also uses arr, hence the introduction of secondA and the like).

4. Polytypic programming with bi-arrows

In the following sections our framework is used in a polytypic context. First, in section 4.1 we present *polytypic traversals* (generalized mappings). We show how to define the right–to–left traversals in terms of the left–to–right using duality. Secondly (section 4.2), we introduce a *state arrow transformer*, i.e., an arrow implementation with which arbitrary arrows can be lifted to an arrow supporting invertible computations on states.

4.1 Polytypic traversals

Polytypic traversals are generalizations of polytypic mappings. They are introduced in Jansson and Jeuring [13]. Polytypic mappings operate on functions, whereas polytypic traversals operate on abstract arrows. Thus, mapping is just a special case of traversal.

However, unlike for mapping, the order of traversal of a data structure now becomes important, due to possible side effects within the arrow.

We specify the traversal operation using the polytypic programming extension of Haskell: Generic Haskell [14]. Every type, except certain predefined/basic types as `Int`, has a generic representation using only sums, products, and units. The Generic Haskell preprocessor can derive[1] the code for a polytypic function, as long as we define the polytypic function for the base instances: `Sum`, `Prod`, and `Unit`.

```
mapl{|a, b|arr|} :: (ArrowChoice arr, BiArrow arr,
                              mapl{|a, b|arr|}) ⇒ arr a b

mapl{|Unit|}     = idA
mapl{|Prod a b|} = inv prodA
                   ⋙ mapl{|a|} *** mapl{|b|}
                   ⋙ prodA
mapl{|Sum a b|}  = inv sumA
                   ⋙ mapl{|a|} +++ mapl{|b|}
                   ⋙ sumA

prodA :: BiArrow arr ⇒ arr (a, b) (Prod a b)
prodA = fst ‘splt‘ snd ↔ exl ‘split‘ exr

sumA :: BiArrow arr ⇒ arr (Either a b) (Sum a b)
sumA = Inl ‘either‘ Inr ↔ Left ‘junc‘ Right
```

Some remarks about `mapl`:

- There is a context restriction on the monotypic type variable `arr`. Generic Haskell expects such type variables to be declared after the polytypic type variables, separated by a `|`.
- Besides the usual context restrictions on `arr` there is also a context restriction over `mapl` itself. This is due to the fact that the `mapl` is polytypic. Usually, these are derived automatically by Generic Haskell[2] and can be omitted.
- The adaptors `prodA` and `sumA` would be superfluous if the definitions of `Prod` and `Sum` would coincide with `(,)` and `Either`. The `splt` and `junc` functions are the `Prod` and `Sum` counterparts of `split` and `either` for tuples and `Either`s, respectively.
- For clarity reasons we have omitted the cases for constructor information (i.e., instances for `Con` and `Label`) as they are not essential for the examples in this paper.

Generic Haskell can derive a specific traversal function for any data type using the schematic representation of that type. In the present paper we will not need derived instances other than for types of kind $\star \to \star$. Unfortunately, Generic Haskell does not yet support the use of generic functions in the context restrictions of type classes and instances. We simulate this by introducing a dummy class, for which define the necessary instances in the obvious way. For types of kind $\star \to \star$ this leads to the class `Gmapl`.

```
class Gmapl t where
    gmapl :: (ArrowChoice arr, BiArrow arr) ⇒
             arr a b → arr (t a) (t b)
```

For instance, we can use polytypic traversal to map the increment function to a tree of integers, using the following data type definition for `Tree`, and instance definition of `Gmapl`

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

instance Gmapl Tree where
    gmapl = mapl{|Tree|}
```

Now we can write, again forcing the use of the (`EpT` →) bi-arrow:

```
toEp (gmapl ((λx → x + 1) ↔ (λx → x - 1)))
                  (Leaf 1 ‘Node‘ Leaf 2 ‘Node‘ Leaf 3)
-- yields Leaf 2 ‘Node‘ Leaf 3 ‘Node‘ Leaf 4
```

The way the $***$ and $+++$ are defined determines the traversal order. Basically, the order is left–to–right because $***$ and $+++$ give preference to `first` end `left` respectively. Analogously, one can define the traversals using right–to–left variants of our basic combinators.

Jansson and Jeuring [13] show that such left–to–right and right–to–left traversals (e.g., `mapl` and `mapr`) form a pair of data conversion functions, which are each others inverse. We want to show here that instead of defining both traversals separately, we can define one of them as the inverse of the other, using bi-arrows. We define the `mapr` (the right–to–left traversal) as the dual of the left–to–right traversal.

```
mapr :: (Gmapl t, ArrowChoice arr, BiArrow arr) ⇒
        arr a b → arr (t a) (t b)
mapr f = inv (gmapl (inv f))

toEp (gmapr ((λx → x + 1) ↔ (λx → x - 1)))
                  (Leaf 1 ‘Node‘ Leaf 2 ‘Node‘ Leaf 3)
-- also yields Leaf 2 ‘Node‘ Leaf 3 ‘Node‘ Leaf 4,
-- because the order does not matter in this example
```

4.2 The state bi-arrow transformer

Like monads, arrows can be used to specify computations with side effects on a state. We will show how to define a state arrow in our bi-arrow framework. This state arrow will be used later in an example to define an invertible pair of conversion functions that: *separate* a functor into its shape and its contents and *combine* the shape and the contents back.

Consider the following *arrow transformer*, which adds a state to a given arrow:

```
newtype StT s arr a b = St {unSt :: arr (a,s) (b,s)}
```

The corresponding instances of `Arrow` and `BiArrow` are defined below. This arrow transformer also occurs in [11]. The instances below can be obtained directly from [11] by replacing the unidirectional adapters (defined by means of *arr*) by bidirectional adapters using ↔.

```
instance BiArrow arr ⇒ Arrow (StT s arr) where
    arr     = arrA
    f ⋙ g = St (unSt f ⋙ unSt g)
    first f = St (swapYZA ⋙
                  first (unSt f)
                  ⋙ swapYZA)
    second  = secondA
```

[1] There is a bug in Generic Haskell 1.42, which makes the preprocessor generate ill-typed code when deriving generic function instances for arrows (or other types of kind $\star \to \star \to \star$). As a work around, our source contains generic function instances for all the types that we use. The Clean version of the source does derive generic function instances correctly. However, the Clean compiler 2.1 gives false uniqueness errors when using arrows with generics. As a work around, we provide a copy of StdGeneric without uniqueness attributes.

[2] There is a bug in Generic Haskell 1.42, which makes it generate an infinite amount of code when omitting these context restrictions on the polytypic function itself. The Clean compiler does not require such context restrictions.

```
instance (ArrowChoice arr, BiArrow arr) ⇒
         ArrowChoice (StT s arr) where
    left f = St (liftLSA ⋙
                 left (unSt f)
                 ⋙ inv liftLSA)
    right  = rightA

instance BiArrow arr ⇒ BiArrow (StT s arr) where
    f ↔ g = St (first (f ↔ g))
    inv f  = St (inv (unSt f))

liftLSA :: (ArrowChoice arr, BiArrow arr) ⇒
           arr (Either a b, c) (Either (a, c) (b, c))
liftLSA = swapA ⋙ liftRSA ⋙ swapA +++ swapA

swapYZA :: BiArrow arr ⇒ arr ((a, b), c) ((a, c), b)
swapYZA = assocPA ⋙ second swapA ⋙ inv assocPA
```

The method ↔ of the state arrow is implemented using `first` and ↔ of the underlying arrow. The composition of state arrows just composes the underlying arrows.

The instance of `StT` for the choice arrow is defined with help of distributivity of the product type over the sum type. As usual, such a property is specified by constructing an appropriate bi-arrow, in this case `liftLSA`, a transformation of `liftRSA` from section 3. Again, only minor modifications of the instance declarations given in [11] were necessary.

4.3 Polytypic shape

We use the state arrow of the previous section to define polytypically an invertible pair of conversion functions that separate a functor into its shape and its contents and combine the shape and the contents back. Expressed as ordinary functions the type signatures of these two functions are:

```
separate :: Functor f ⇒ f a → [a] → (f (), [a])
combine  :: Functor f ⇒ f () → [a] → (f a, [a])
```

Instead of defining these functions as primitives, we will use the invertible state arrow. The data stored in/retrieved from the functor is passed as a state. For list states, we introduce the `getputA` arrow. The `getputA` arrow operates on this state and is used to get an input element from or to add an element to the state.

```
getputA :: BiArrow arr ⇒ StT [a] arr () a
getputA = St (get ↔ put)
    where
        get ((), x:xs) = (x, xs)
        put (x, xs)    = ((), x:xs)
```

Since our shape operations are each others inverse, we only have to specify one of them explicitly. We choose to define the `combine` function by using the polytypic traversals introduced in section 4.1.

```
combine :: (Gmapl t, ArrowChoice arr, BiArrow arr) ⇒
           StT [a] arr (t ()) (t a)
combine = gmapl getputA

separate :: (Gmapl t, ArrowChoice arr, BiArrow arr)⇒
            StT [a] arr (t a) (t ())
separate = inv combine
```

The following example illustrates how we can use `combine` to fill an empty tree structure with integers.

```
(toEp . unSt) combine
   (Leaf () ‘Node‘ Leaf () ‘Node‘ Leaf (), [3, 4, 5])
—— yields Leaf 3 ‘Node‘ Leaf 4 ‘Node‘ Leaf 5
```

```
(toEp . unSt) separate
                 (Leaf 3 ‘Node‘ Leaf 4 ‘Node‘ Leaf 5)
—— yields (Leaf () ‘Node‘ Leaf () ‘Node‘ Leaf (),
—— [3, 4, 5])
```

5. Polytypic (de)serialization

In this section we present an example of encode-decode pair of functions that implement structure-based encoding and decoding of data.

The packing function takes data and converts it into a list of bits (Booleans), whereas the unpacking function recovers data from a list of bits. The bit representation directly represents the structure of data using only *static* information (the type of the data), not *dynamic* information (the value stored in a data structure), like some other compression methods do.

The choice which conversion should be specified is again arbitrary. We pick the decoder, which reads the bits from the input, and produces the original data structure. To obtain such a decoder for any data type, we will give a polytypic specification.

Basic types, like `Char` and `Int`, are encoded with a fixed number of bits. Although we could specify this primitive operation by means of arrow combinators, it appears to be easier to define it as a pure function, and to lift it to an arrow.

```
int2KBitsA :: BiArrow arr ⇒ Int → arr Int [Bool]
int2KBitsA k = int2bits k ↔ bits2int k
    where
        int2bits 0 n = [ ]
        int2bits k n = odd n:int2bits (k-1)
                                          (n ‘div‘ 2)
        bits2int 0 bs          = 0
        bits2int k (True:bs)   = 1+bits2int (k-1) bs*2
        bits2int k (False:bs) = bits2int (k-1) bs*2
```

Now, the decoder for integers can be defined. It expects a list of bits, which has to be taken from the state. This is done by first producing the shape of the list and then by filling this list using the `combine` arrow of the previous section.

```
decodeInt :: (ArrowChoice arr, BiArrow arr) ⇒
             Int → StT [Bool] arr () Int
decodeInt k = createShapeA k ⋙ combine
            ⋙ inv (int2KBitsA k)

createShapeA :: BiArrow arr ⇒ Int → arr () [()]
createShapeA size = create ↔ etaerc
    where
        create () = replicate size ()
        etaerc l | length l == size = ()
```

The encoder for integers is the dual of the decoder for integers:

```
encodeInt :: (ArrowChoice arr, BiArrow arr) ⇒
             Int → StT [Bool] arr Int ()
encodeInt k = inv (decodeInt k)
```

The decoder defined as a polytypic function is:

```
decode{|t|arr|} :: (ArrowChoice arr, BiArrow arr,
                    decode{|t|arr|}) ⇒ StT [Bool] arr () t

decode{|Unit|}      = voidUnitA
decode{|Int|}       = decodeInt 32
decode{|Char|}      = decodeInt 8 ⋙ toEnum ↔ fromEnum
decode{|Bool|}      = getputA
decode{|Prod a b|} = dupVoidA
                     ⋙ decode{|a|} *** decode{|b|}
```

```
                       ⋙ prodA
decode{|Sum a b|}  = getputA ⋙ bool2EitherA
                       ⋙ decode{|a|} +++ decode{|b|}
                       ⋙ sumA
```

voidUnitA is the conversion between () and Unit, dupVoidA duplicates the input (), and bool2eitherA is the isomorphism between the boolean type and the co-product of voids.

```
voidUnitA :: BiArrow arr ⇒ arr () Unit
voidUnitA = (λ() → Unit) ↔ (λUnit → ())

dupVoidA :: BiArrow arr ⇒ arr () ((), ())
dupVoidA = (λ() → ((), ())) ↔ (λ((), ()) → ())

bool2EitherA :: BiArrow arr ⇒
                arr Bool (Either () ())
bool2EitherA = bool2either ↔ either2bool
    where
        bool2either b = if b then Right ()
                             else Left ()

        either2bool (Left ()) = False
        either2bool (Right ()) = True
```

The polytypic decoder is programmed as follows.

- Since Unit can be encoded with zero bits; the case for Unit just returns Unit.
- The case for Booleans just reads one bit.
- The case for integers reads a 32-bit integer with help of the integer decoder defined before.
- The case for characters reads an 8-bit integer and converts into a character.
- The case for pairs first makes two units out of one. Then it applies the decoding componentwise.
- Finally, the case for the sum type first reads one bit to determine whether the left of the right branch should be decoded next.

Using duality we get the encoder for free from the definition of the decoder.

```
encode{|t|arr|} :: (ArrowChoice arr, BiArrow arr,
                decode{|t|arr|}) ⇒ StT [Bool] arr t ()
encode{|t|} = inv decode{|t|}
```

For example, to encode a tree containing the integers 1, 2, and 3 we simply write:

```
(toEp . unSt) encode{|Tree Int|}
          (Leaf 1 `Node` Leaf 2 `Node` Leaf 3, [ ])
```

The output consists of 101 bits: 96 for the integers and 5 bits for the nodes and leaves of the tree structure.

6. Monadic programming with bi-arrows

Up to now, our examples did not have to deal with failure. Of course, the decoding algorithm will not terminate properly if the input data does not correspond to a value, e.g., if some of the bits are missing. For expressing the algorithm this was not essential, but in a real application such an decoding function is not acceptable because it might lead to uncontrolled termination. On the other hand, it is much harder to preserve invertibility if functions are able to fail.

In this section we present appropriate techniques to handle failure without losing invertibility completely. We first introduce bi-arrow definitions for polytypic zipping/unzipping (section 6.1). Then, we define the class ArrowZero (section 6.2) and show how in certain cases it can be used for the zipping example. To obtain a useful implementation of this new class, section 6.3 adds a monadic arrow transformer to our arsenal. As a short example, this monadic bi-arrow is applied to the Maybe monad, which adds support of graceful failure to the polytypic zip function. In section 7 we will extend our collection of arrow classes further with a combinator that, when applied to two arrows, will choose the second one if the first one fails.

6.1 Partial polytypic zipping

First, we introduce a polytypic function that is closely related to the polytypic traversals of section 4.1: polytypic zipping/unzipping. It cannot deal with failure, which we will fix later on.

A binary zipping takes two structures of the same shape and combines them into a single structure. Unzipping does the opposite. In our bidirectional framework, we get unzipping for free if we define zipping as a bi-arrow. This can be done as follows:

```
zip{|a, b, c|arr|} :: (ArrowChoice arr, BiArrow arr,
                    zip{|a, b, c|arr|}) ⇒ arr (a, b) c

zip{|Unit|}     = inv dupUnitA
zip{|Prod a b|}= unprod2A ⋙ zip{|a|} *** zip{|b|} ⋙ prodA
zip{|Sum a b|} = unsum2A ⋙ zip{|a|}+++zip{|b|} ⋙ sumA

dupUnitA :: BiArrow arr ⇒ arr Unit (Unit, Unit)
dupUnitA = (λUnit → (Unit, Unit))
         ↔ (λ(Unit, Unit) → Unit)

unprod2A :: BiArrow arr ⇒
            arr (Prod a b, Prod c d) ((a, c), (b, d))
unprod2A = dorp ↔ prod
    where
        dorp (x1:*:x2, y1:*:y2) = ((x1, y1), (x2, y2))
        prod ((x1, y1), (x2, y2)) = (x1:*:x2, y1:*:y2)

unsum2A :: BiArrow arr ⇒
           arr (Sum a b,Sum c d) (Either (a,c) (b,d))
unsum2A = mus ↔ sum
    where
        mus (Inl l1, Inl l2) = Left (l1, l2)
        mus (Inr r1, Inr r2) = Right (r1, r2)

        sum (Left (l1, l2))  = (Inl l1, Inl l2)
        sum (Right (r1, r2)) = (Inr r1, Inr r2)
```

Just as encode is the inverse of decode, we define unzip as the inverse of zip.

```
unzip{|t|arr|}:: (ArrowChoice arr, BiArrow arr, zip{|t|})
              ⇒ arr c (a, b) → arr (t c) (t a, t b)
unzip{|t|} f = inv (zip{|t|} (inv f))
```

Note that this definition for zip is partial: when two structures do not have the same shape the result of zipping these structures is undefined. Obviously, the inverse of zipping is a total function.

```
toEp (unzip{|Tree|} idA)
                 (Leaf (1, 'a') `Node` Leaf (2, 'b'))
-- yields
-- Leaf 1 `Node Leaf 2, Leaf 'a' `Node` Leaf 'b'
```

Sometimes it is necessary that zipping itself is total, i.e., it should check whether the input structures match and handle it gracefully if not. This is usually done by returning a Maybe value in which Nothing indicates that the structures were not of the same shape/size.

However, in this case the inverse, unzipping, becomes partial: if zipping returns Nothing it is in general impossible to reconstruct the non-matching argument structures.

6.2 Bi-arrows with zero

To deal with operations that can fail we use the ArrowZero class.

```
class Arrow arr ⇒ ArrowZero arr where
  zeroArrow :: arr a b
```

The arrow zeroArrow is the multiplicative zero for composition with pure (bi-)arrows, i.e.,

$$f \ggg \mathit{zeroArrow} = \mathit{zeroArrow} = \mathit{zeroArrow} \ggg f$$

Clearly, this law excludes that zeroArrow has an inverse. However, this does not imply that we completely lose invertibility when zeroArrow is used: in many cases the *left* inverse of a failing operation still exists. More formally, an arrow f if *left-invertible* if *inv* $f \ggg f = idA$

The following derived combinator ||> (left-fanin), which is a bidirectional variant of the ||| (fanin) arrow combinator, appears to be convenient in combination with zeroA.

```
(||>) :: (ArrowChoice arr, BiArrow arr) ⇒ -- infixr 4
         arr a c → arr b c → arr (Either a b) c
f ||> g = f +++ g ⋙ untagRA

untagRA :: BiArrow arr ⇒ arr (Either a a) a
untagRA = id ‘either‘ id ↔ Right
```

From this definition we cannot conclude directly that it is invertible, because id ‘either‘ id is not the inverse of Right and, therefore, the occurrence of ↔ in untagRA is not invertible. We call this combinator *right-biassed* because, in the reverse direction, it always yields Right. Nevertheless, we can show that the ||> combinator preserves left-invertibility. More specifically, it can be shown that the arrow f ||> g is left-invertible if g is left-invertible. Analogously, it follows that left-biassed combinators preserve right-invertibility.

We can use the new combinator ||> with zeroA to extend zip with explicit failure. In fact, the only polytypic instance that changes is the one for Sum, see below. Additionally, we must add the ArrowZero class as a context restriction to the type of zip.

```
zip{|a, b, c|arr|} :: (ArrowZero arr, ArrowChoice arr,
                       BiArrow arr, zip{|a, b, c|arr|}) ⇒
                       arr (a,b) c

zip{|Sum a b|} = unsum2FA
             ⋙ zeroArrow ||> (zip{|a|} +++ zip{|b|})
             ⋙ sumA
unsum2FA = mus ↔ sum
where
  mus (Inl l1, Inl l2) = Right (Left  (l1, l2))
  mus (Inr r1, Inr r2) = Right (Right (r1, r2))
  mus (s1, s2)         = Left (s1, s2)

  sum (Right (Left  (l1, l2))) = (Inl l1, Inl l2)
  sum (Right (Right (r1, r2))) = (Inr r1, Inr r2)
  sum (Left  (s1, s2))         = (s1, s2)
```

Now the adaptor unsum2FA tags the result with an additional sum constructor to indicate whether the constructors matched. In particular, it uses Right in case both constructors were identical, and Left if they were different. In the latter case the zeroArrow branch of ||> is chosen, whereas in the first case the ‘normal’ zip{|a|} +++ zip{|b|} is performed.

6.3 Lifting monads to bi-arrows

To be able to apply zip to concrete data structures we need appropriate instances for our arrow classes, including ArrowZero.

A convenient and flexible way to manage failures, but also to implement other concepts such as non-determinism and states, is obtained by using monads. Monadic arrows are arrows that represent monadic computations.

The goal of this section is twofold: to show how we deal with monadic arrows in the bidirectional arrow framework and to provide the basis for handling failures.

We use the same classes for monads that can be found in Haskell [10]. The basic monad is defined with the *return* and *bind* operations:

```
class Monad m
where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

The plus monad will be used to support failures of monadic arrows, and also to implement *choices*.

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

Usually, the Kleisli arrow transformer is used to represent monadic computations [11, 13], which is defined on a monad m as follows:

```
newtype K m arr a b = K {unK :: arr a (m b)}
```

However, this arrow is not suitable for our purposes, because it is not possible to define an instance of inv on it: it handles the argument and result asymmetrically. As symmetrical version of the Kleisli transformer can be obtained by adjusting the argument type in the definition of K as follows:

```
newtype MoT m arr a b = Mo {unMo :: arr (m a) (m b)}
```

The instances of Arrow, BiArrow and ArrowChoice on MoT require that we are able to traverse the underlying monad. This will be done by using the polytypic mapping Gmapl from section 4.1.

However, this limits the choice for m to data types, because it is impossible to instantiate Gmapl for function types. In the instance definitions we use the auxiliary arrows firstMA and leftMA based on the monadic join and return operations.

```
instance (Gmapl m, Monad m, ArrowChoice arr,
        BiArrow arr) ⇒ Arrow (MoT m arr) where
  arr     = arrA
  f ⋙ g = Mo (unMo f ⋙ unMo g)
  first f = Mo (inv firstMA ⋙
                gmapl (first (unMo f))
                ⋙ firstMA)
  second  = secondA

instance (Monad m, ArrowChoice arr, BiArrow arr,
        Gmapl m) ⇒ ArrowChoice (MoT m arr) where
  left f = Mo (inv leftMA ⋙
               gmapl (left (unMo f))
               ⋙ leftMA)
  right  = rightA

instance (Gmapl m, Monad m, ArrowChoice arr,
        BiArrow arr) ⇒ BiArrow (MoT m arr) where
  f ↔ g = Mo (liftM f ↔ liftM g)
  inv f  = Mo (inv (unMo f))
```

with

```
firstMA :: (Monad m, BiArrow arr) ⇒
           arr (m (m a, b)) (m (a, b))
firstMA = joinP ↔ splitP
    where
        joinP  = (=<<) (λ(mx, y) → mx >>= λx →
                                      return (x, y))
        splitP = (=<<) (λ(x, y) → return
                                      (return x, y))

leftMA :: (Monad m, BiArrow arr) ⇒
          arr (m (Either (m a) b)) (m (Either a b))
leftMA = joinS ↔ splitS
    where
        joinS  = (=<<) ((=<<) (return . Left)
                         `either` (return . Right))
        splitS = (=<<) ((return . Left . return)
                         `either` (return . Right))

liftM :: Monad m ⇒ (a → b) → m a → m b
liftM f m = m >>= λx → return (f x)
```

Here we should mention that invertibility of `firstMA` and `leftMA` depends on the underlying monad. E.g. for the `Maybe` monad it can be shown that both `firstMA` and `leftMA` are invertible; for the list monad this does not hold.

One of the purposes of the monadic arrows is to handle failures. The zero monadic arrow is defined with help of `mzero`.

```
instance (Gmapl m, MonadPlus m, ArrowChoice arr,
        BiArrow arr) ⇒ ArrowZero (MoT m arr) where
    zeroArrow = Mo (const mzero ↔ const mzero)
```

To illustrate the use of the monadic arrow we return to our generic zipping function. For example, combining the information of two trees is successful:

```
(toEp . unM) (zip{|Tree|} idA)
  (Just (Leaf 1 `Node` Leaf 3, Leaf 2 `Node` Leaf 4))
 —yields Just (Leaf (1,2) `Node` Leaf (3,4))
```

And if we try to combine two trees with different shape, it yields the `mzero`:

```
(toEp . unMo) (zip{|Tree|} idA)
                 (Just (Leaf 1 `Node` Leaf 3, Leaf 2))
—yields Nothing
```

7. Parsing and pretty-printing

In this section we show how to define a parser based on our reversible arrow combinators. Again, we will get the inverse, a pretty-printer, for free.

We give an example of a parser for a very simple functional language, specified by the following grammar in *BNF* notation.

```
<Expression>  ::= <Expression> <Expression>
               |  "(" <Expression> ")"
               |  "λ" <Variable> "→" <Expression>
               |  <Variable>
               |  <Constructor>

<Variable>    ::= <String>
<Constructor> ::= <String>
```

The main difference between the decoder of section 5 and a parser is that the decoder does not have to choose between alternatives, since its action for the sum type is solely depends on the next input bit. The parser presented in this section will try alternatives to see, which of them succeeds.

Another difference is that the parser is not completely determined by the type of the term it parses. It is because it needs to parse extra spaces, parentheses etc. Consequently, we cannot expect that the resulting parser is (left *and* right) invertible, because different input sentences, may lead to the same result.

Analogously to encode-decode, we define the parser and derive the corresponding pretty-printer. So, the programmer does not need to write the complete pretty-printer code.

7.1 The plus arrow

Failure of parsers is handled by the `ArrowZero`. What we still need is a combinator that, when applied to two parsers, will choose the second in case the first one fails.

We therefore introduce one further arrow class, comparable to the `MonadPlus` class of monadic parser combinators.

```
class ArrowZero arr ⇒ ArrowPlus arr where
    (<|>) :: arr a b → arr a c → arr a (Either b c)
```

In contrast to the Haskell's arrow plus combinator <+>, our combinator tags its result so we can still see which parser has been chosen.

As said before, if possible the <|> chooses a non-failing computation. This is expressed by the law

$$zeroArrow <|> f = f = f <|> zeroArrow$$

The implementation of `ArrowZero` and `ArrowPlus` for the state arrow is straightforward (`liftLSA` has been defined in section 4.2).

```
instance (ArrowZero arr, BiArrow arr) ⇒
        ArrowZero (StT s arr) where
    zeroArrow = St (first zeroArrow)

instance (ArrowPlus arr, ArrowChoice arr,
        BiArrow arr) ⇒ ArrowPlus (StT s arr) where
    f <|> g = St (unSt f <|> unSt g >>> inv liftLSA)
```

Instantiating `ArrowPlus` for the monadic arrow is much more complex. We defer its definition until the end of this section.

7.2 A concrete parser

As in the previous sections, we will use a combination of the state and monadic arrows to build a concrete example parser. The resulting syntax tree is represented by the data structure.

```
data Expression = App Expression Expression
                | Nested Expression
                | Lambda String Expression
                | Variable String
                | Constructor String
```

Observe that the syntax tree explicitly stores whether an expression was enclosed by brackets. This is done to ensure that, when printing a parsed expression, brackets are displayed correctly.

To abstract from the parsing issues at the lexical level, we assume a separated scanner/lexer and that the parser will work on a list of tokens. This leads to:

```
data Token = Id_T String | Lambda_T | Open_T
           | Close_T | Arrow_T | EOF_T deriving Eq

type Parser arr t  = StT [Token] arr () t
type Printer arr t = StT [Token] arr t ()
```

7.3 Parsing keywords

Before defining a parser for expressions, we introduce two auxiliary parsers to examine the input tokens.

The first one, parseKeyword, tries to read a given token from the input stream. If it succeeds, this token is delivered as result; if not, the parser fails. As with the zip example of section 6.3 we use ||> in combination with zeroArrow to handle failure.

```
parseKeyword token = getputA >>> tagTokenA
                  >>> zeroArrow ||> idA
    where
        tagTokenA = test <-> id `either` id
        test t = if t == token then Right t
                                else Left t
```

The second one examines the input list to see whether the next token is an identifier. Moreover, to distinguish variables (starting with a lower case char) from constructors (starting with a upper case char) this parser is parameterized with a predicate. The parser succeeds in case of an identifier token fulfilling the predicate. Then the identifier itself is returned, otherwise the parser fails.

```
parseIdentifier p = getputA >>> tagIDA p
                 >>> zeroArrow ||> idA
    where
        tagIDA p = tagID p  <-> id `either` Id_T

        tagID p (Id_T name) | p name = Right name
        tagID _ token                = Left token
```

7.4 Parsing expressions

The grammar of our input language is left-recursive, and hence cannot be directly translated into a parser. We introduce an intermediate function for parsing expressions (called *terms*) which are no applications.

```
parseTerm = parseNested
          <|> parseLambda
          <|> parseVariable
          <|> parseConstructor
          >>> toExp <-> fromExp
    where
        toExp = Nested `either` (uncurry Lambda
            `either` (Variable `either` Constructor))

        fromExp (Lambda var exp) =
                                 Right (Left (var, exp))
        fromExp (Variable var)   =
                                Right (Right (Left var))
        fromExp (Constructor c)  =
                                 Right (Right (Right c))
        fromExp (Nested nested)  =       Left nested
```

parseTerm combines parsers for all expression kinds by using the arrow plus combinator. The result, tagged with various Lefts and Rights, is converted by the adapter to_expr ↔ from_expr into the corresponding part of the syntax tree.

For parsing consecutive elements, we use an helper combinator based on *** and the dupVoidA arrow defined in section 5.

```
(<&>) :: BiArrow arr =>                      -- infixl 6
         arr () a -> arr () b -> arr () (a, b)
f <&> g = dupVoidA >>> f *** g

parseLambda = parseKeyword Lambda_T
            <&> parseVariable
            <&> parseKeyword Arrow_T
            <&> parseExpression
            >>> toLambda <-> fromLambda
    where
        toLambda   (((_, v), _), e) = (v, e)
        fromLambda = const Lambda_T `split` fst
                   `split` const Arrow_T `split` snd

parseNested = parseKeyword Open_T
            <&> parseExpression
            <&> parseKeyword Close_T
            >>> toExp <-> fromExp
    where
        toExp ((_, e), _) = e
        fromExp e = ((Open_T, e), Close_T)

parseVariable    = parseIdentifier (isLower . head)
parseConstructor = parseIdentifier (isUpper . head)
```

The parser for applications takes some more doing. It first reads a list of terms and converts this into a tree of binary applications.

We introduce a function parseOneOrMore to parse a list of elements that, when applied to a parser p, tries to parse one or more p-elements.

```
parseOneOrMore p = p <&> parseOneOrMore p <|> p
                 >>> untag <-> tag
    where
        untag (Left  (x, (y, l))) = (x, y:l)
        untag (Right  x)          = (x, [])

        tag (x, y:l) = Left (x, (y, l))
        tag (x, [])  = Right x
```

Note that this parseOneOrMore will try to find the longest list. The parser for expressions can now be expressed easily.

```
parseExpression = parseOneOrMore parseTerm
                >>> uncurry to_apply <-> from_apply []
    where
        to_apply app []     = app
        to_apply app (x:xs) = to_apply (App app x) xs

        from_apply l (App f a) = from_apply (a:l) f
        from_apply l t         = (t, l)
```

Finally, the pretty-printer for expressions is obtained by taking the inverse of the parser.

```
parse :: (ArrowPlus arr, ArrowChoice arr,
         BiArrow arr) => Parser arr Expression
parse = parseExpression <&> parseKeyword EOF_T>>>eofA
    where
        eofA = fst <-> (\x -> (x, EOF_T))

print :: (ArrowPlus arr, ArrowChoice arr,
         BiArrow arr) => Printer arr Expression
print = inv parse
```

7.5 A monadic plus arrow

Before we can really use our parser we have to provide an appropriate implementation of the plus arrow.

More specifically, we need an instance definition of ArrowPlus for the monadic arrow transformer M. Of course, this instance will be based on the mplus of the underlying monad.

```
instance (Gmapl m, MonadPlus m, ArrowChoice arr,
          BiArrow arr) => ArrowPlus (MoT m arr) where
    l <|> r = Mo (dupMA >>>
              (unMo l >>> inlMA) *** (unMo r >>> inrMA)
              >>> inv dupMA)
```

The adapter arrows dupMA, inlMA and inrMA are defined as follows.

```
dupMA :: (MonadPlus m, BiArrow arr) ⇒
         arr (m a) (m a, m a)
dupMA = (λx → (x, x)) ↔ uncurry mplus

inlMA :: (MonadPlus m, BiArrow arr) ⇒
         arr (m a) (m (Either a b))
inlMA = inlM ↔ uninlM
    where
        inlM   = (=<<) (return . Left)
        uninlM = (=<<) (return ‘either‘ const mzero)

inrMA :: (MonadPlus m, BiArrow arr) ⇒
         arr (m a) (m (Either b a))
inrMA = inrM ↔ uninrM
    where
        inrM   = (=<<) (return . Right)
        uninrM = (=<<) (const mzero ‘either‘ return)
```

The adapter dupMA is in general *not* invertible, because the arguments of ↔ are obviously not each others inverse. This means that the instance of <|> is also not invertible, because it defined in terms of dupMA and inv dupMA.

Consequently, when defining an operation using this instance of <|> one does not get invertibility for free, i.e. it is no longer sufficient to prove that all pairs of pure functions lifted with ↔ are each others inverse. To show correctness, global reasoning is required.

In practice, this may imply that the inverse of the operation needs to be fine-tuned in order to produce the expected result. In particular this holds for our parser example. The Nested constructor was added to the syntax tree to be able to reconstruct the brackets that were used to disambiguate expressions.

7.6 Parser/printer examples

Suppose we have the following list of input tokens:

```
tokens = [Open_T, Lambda_T, Id_T ”x”, Arrow_T,
          Id_T ”x”, Close_T, Lambda_T, Id_T ”y”,
          Arrow_T, Id_T ”y”, EOF_T]
```

To parse this and convert it into an expression, we write:

```
(toEp . unMo . unSt) parse (return ((), tokens))
                         :: Maybe (Expression, [Token])
```

And if we want to print the expression:

```
expr = App (Nested (Lambda ”x” (Variable ”x”)))
                              (Lambda ”y” (Variable ”y”))
```

we simply write:

```
(toEp . unMo . unSt) print (return (expr, [ ]))
                         :: Maybe ((), [Token])
```

The Maybe-monad does not reveal that the expression parser is ambiguous.

Suppose we leave out the Nested constructor in the last example expression. Printing this expression will lead to a list of tokens not containing the open and close brackets anymore. Our parser will still be able to parse this list but it will not produce the same expression we have started with: the App will occur inside the first lambda expression. The reason is that our parser only delivers one successful parse.

However, in our framework it is very easy to change the parser in such a way that it delvers all successful parses, namely, by using the list monad instead of the maybe monad. This list monad is a standard implementation of the monad class. So, the only thing we have to change for our example is the type!

```
(toEp . unMo . unSt) parse (return ((), tokens))
                         :: [(Expression, [Token])]
```

Running this expression with the following list of tokens

```
tokens = [Lambda_T, Id_T ”x”, Arrow_T, Id_T ”x”,
          Lambda_T, Id_T ”y”, Arrow_T, Id_T ”y”,
          EOF_T]
```

will now yield two expressions:

```
App (Lambda ”x” (Variable ”x”))
                          (Lambda ”y” (Variable ”y”))
```

and

```
Lambda ”x” (App (Variable ”x”)
                          (Lambda ”y” (Variable ”y”)))
```

8. Related Work

This work is inspired by Jansson and Jeuring [13, 12] who define polytypic functions for parsing and pretty-printing and then prove invertibility. They maintain invertibility using pairs of separate definitions, leading to many proof obligation for the programmer. In contrast, we use one single definition for both conversion directions using invertibility preserving combinators. As a result we only have to prove invertibility for the primitives that are used. Furthermore, our approach is not limited to the example of parsing nor to the use of polytypic functions.

Invertibility is an important practical property used in many algorithms. For instance, it plays an important role in the database world where one has to ensure that any change in a view domain leads to a corresponding change in the underlying data domain.

To ensure this property, Foster et. al. [5] present a domain-specific programming language in which all expressions denote bi-directional transformations on trees. They use two functions, a get function for extracting an abstract view from a concrete one, and a put function that creates an updated concrete view given the original concrete view and the updated abstract view. Using the proper get and put functions, invertibility is guaranteed.

For similar purposes Mu et al. [20] define a programming language in which only injective functions can be defined, thus guaranteeing invertibility. Again put and get functions are defined, but the crux here is to do some bookkeeping when doing a get such that a put can always be made invertible.

A different approach is taken by Robert Glück and Masahiko Kawabe [6, 7]. They try to construct the inverse function from the original one automatically. They use a symmetrical representation for functions such that the inverse function can be constructed by interpreting the original function backwards. Our arrow combinators have a representation with this same property. The main difference with our work is we obtain the inverse function by construction while they try to automatically generate an inverse function from the original one. They use LR-parsing techniques and administrative bookkeeping to invert choices made by conditional branches in the original function.

There is a lot of work about inverting *existing* programs, both functional and imperative, see for example: Dijkstra [4], Chen [2], and Ross [19]. Our approach is more hands-on and focusses on constructing (parts of) programs in an invertible way.

9. Conclusions and Future Work

We feel that we have provided an interesting framework in the area of invertible programming.

We have extended arrows to bidirectional arrows, bi-arrows, that preserve invertibility properties. We have presented several invertible bi-arrow transformers. Bi-arrows were used in a monotypic and in a polytypic context. We introduced ways to deal with state and with monads. A concrete parser/pretty printer example was presented with a discussion of its properties.

For future work we want to provide full formal proof that the framework preserves invertibility properly. Furthermore, we will investigate whether the approach scales up to real world practical examples where invertibility properties are a requirement. Among other things this will require creating a translation scheme similar to Paterson notation in such a way that the required properties are preserved, and that programs are easier to read and write.

Acknowledgments

We would like to thank the anonymous referees of an earlier version of this paper for their helpful comments.

References

[1] A. Alimarine and R. Plasmeijer. A Generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *The 13th International workshop on the Implementation of Functional Languages, IFL'01, Selected Papers*, pages 168–186. Älvsjö, Sweden, Sept. 2002.

[2] W. Chen and J. T. Udding. Program inversion: more than fun! *Sci. Comput. Program.*, 15(1):1–13, 1990.

[3] A. Courtney and C. Elliott. Genuinely Functional User Interfaces. In *Proceedings of the 2001 Haskell Workshop*, September 2001.

[4] E. W. Dijkstra. Program inversion. In *Program Construction*, pages 54–57, 1978.

[5] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Long Beach, California*, 2005. Extended version available as University of Pennsylvania technical report MS-CIS-03-08. Earlier version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[6] R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on lr parsing. 2998:291–306, 2004.

[7] R. Glück and M. Kawabe. Revisiting an automatic program inverter for lisp. volume 40, pages 8–17, New York, 2005. ACM Press.

[8] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, Aug. 2001. The preliminary proceedings appeared as a University of Nottingham technical report.

[9] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Reactive Programming. In J. Jeuring and S. Peyton Jones, editors, *Advanced Functional Programming, 4th International School*, volume 2638 of *LNCS*, Oxford, 2003. Springer.

[10] P. Hudak, J. Peterson, and J. Fasel. A gentle introduction to Haskell 98. http://www.haskell.org/tutorial/, 1999.

[11] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000.

[12] P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In S. D. Swierstra, editor, *Proceedings 8th European Symposium on Programming, ESOP'99, Amsterdam, The Netherlands, 22–28 March 1999*, volume 1576, pages 273–287. Springer-Verlag, Berlin, 1999.

[13] P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.

[14] A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pages 141–152. ACM Press, 2003.

[15] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, Sept. 2001.

[16] R. Paterson. Arrows and Computation. In J. Gibbons and O. de Moor, editors, *The Fun of Programming, A symposium in honour of Professor Richard Bird's 60th birthday*, pages 201–222, Oxford, 2003. Palgrave.

[17] S. Peyton Jones and Hughes J. et al. *Report on the programming language Haskell 98*. University of Yale, 1999. http://www.haskell.org/definition/.

[18] R. Plasmeijer and M. van Eekelen. *Concurrent CLEAN Language Report (version 2.0)*, December 2001. http://www.cs.ru.nl/∼clean/.

[19] B. Ross. Running programs backwards: the logical inversion of imperative computation. *Formal Aspects of Computing Journal*, 9:331–348, 1997.

[20] Z. H. S-C. Mu and M. Takeichi. An algebraic approach to bidirectional updating. In *The Second Asian Symposium on Programming Language and Systems*, volume 3302 of *LNCS*, pages 2–18. Springer, 2004.

[21] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.