Polytypic Data Conversion Programs

Patrik Jansson

Computing Science, Chalmers University of Technology, Sweden
patrikj@cs.chalmers.se, http://www.cs.chalmers.se/~patrikj/

Johan Jeuring

Computer Science, Utrecht University, the Netherlands
johanj@cs.uu.nl, http://www.cs.uu.nl/~johanj/

Abstract

Several generic programs for converting values from regular datatypes to some other format, together with their corresponding inverses, are constructed. Among the formats considered are shape plus contents, compact bit streams and pretty printed strings. The different data conversion programs are constructed using John Hughes' arrow combinators along with a proof that printing (from a regular datatype to another format) followed by parsing (from that format back to the regular datatype) is the identity. The printers and parsers are described in PolyP, a polytypic extension of the functional language Haskell.

1 Introduction

Many programs convert data from one format to another, for example, parsers, pretty printers, data compressors, encryptors and functions that communicate with a database. Some of these programs, such as parsers and pretty printers, critically depend on the structure of the input data. Other programs, such as most data compressors and encryptors, more or less ignore the structure of the data. Using the structure of the input data in a program for a data conversion problem almost always gives a more efficient program with better results. For example, a data compressor that uses the structure of the input data runs faster and compresses better than a conventional data compressor. This paper constructs several polytypic data conversion programs that make use of the structure of the input data. We construct programs for determining the shape of data, traversing, packing and pretty printing data.

1.1 Data conversion programs

1.1.1 Shape.

A value of a container type $d\ a$ can be uniquely represented by its shape (of type $d\ ()$) and a list of its contents (of type $[a]$). As an example, consider the datatype of binary trees with leaves containing values of type a.

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Leaf}\ a \mid \mathit{Bin}\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

The following example binary tree

$$\begin{aligned} \mathit{tree} &:: \mathit{Tree}\ \mathit{Int} \\ \mathit{tree} &= \mathit{Bin}\ (\mathit{Bin}\ (\mathit{Leaf}\ 1)\ (\mathit{Bin}\ (\mathit{Leaf}\ 7)\ (\mathit{Leaf}\ 3)))\ (\mathit{Leaf}\ 8) \end{aligned}$$

can be represented by a pair of its shape

$$\begin{aligned} \mathit{treeShape} &:: \mathit{Tree}\ () \\ \mathit{treeShape} &= \mathit{Bin}\ (\mathit{Bin}\ (\mathit{Leaf}\ ())\ (\mathit{Bin}\ (\mathit{Leaf}\ ())\ (\mathit{Leaf}\ ())))\ (\mathit{Leaf}\ ()) \end{aligned}$$

and its contents $[1, 7, 3, 8]$.

Our first data conversion program is a program *separate* for separating a value into its shape and its contents, together with its inverse: a program *combine* that combines a shape and some contents into a datatype value. These programs are *polytypic* [15] programs: programs that work uniformly for large classes of datatypes. The construction proves that the two functions are each others' inverses. Note that shapes are at the heart of Jay's [14] theory of polytypism, but here we only use *separate* and *combine* as examples of simple data conversion programs.

1.1.2 Traversals.

When working with structured data, one often performs operations on all elements in a structure. A *traversal* is an operation on a structured value that walks through the structure and performs some task at each of the elements stored in the structure. Traversals are simple data conversion programs which leave the shape of the input structure unchanged but which can change, or collect information about, the contents. The classical functional programming combinator $\mathit{map} :: (a \to b) \to [a] \to [b]$ is a very simple traversal over lists. A more general traversal could carry around an environment (for example, a dictionary to spell-check every word in a structured text), update a state (for example, collecting a list of all variables in an abstract syntax tree) or collect multiple results (for example, alternative layouts for pretty printing a tree).

We define general polytypic traversals, *arrow maps*, which can do all of this uniformly for all regular datatypes.

For many applications the traversal order is important, and therefore we define a forward and a backward arrow map and prove that they are inverses. The arrow maps are surprisingly simple to define (they are very similar to the normal polytypic map function) but much of the essential structure of the packing and pretty printing programs is present already at this stage.

1.1.3 Packing.

Many files that are distributed around the world, either over the Internet or on CD-rom, possess structure — examples are databases, HTML files, and JavaScript programs — and it pays to compress these structured files to obtain faster transmission or fewer CDs. Structure-specific compression methods give much better compression results than conventional compression methods such as the Unix compress utility [2,4]. Structured compression is also used in heap compression and binary I/O [23].

The idea of designing structure-specific compression programs has been around since the beginning of the 1980s, but for many years only example instantiations appeared in the literature. This paper describes the compression program generically by combining a polytypic parser with a polytypic packing program. The uncompression program is similarly composed of a polytypic unpacker and a polytypic pretty printer. The first implemented generic description of a packing program was in our earlier work [12].

Our packing algorithm compresses data by compactly representing the structure of the data using only static information — the type of the data. Traditional (bit stream) compressors that use dynamic (statistical) properties of the data are largely orthogonal to our approach and thus better compression results are obtained by composing the packer with a bit stream compressor.

1.1.4 Pretty printing.

Modern programming languages allow the user to define new kinds of data. When testing or debugging a program, the user often wants to see values of these new datatypes. Many languages support the automatic derivation of printing functions for user-defined datatypes. For example, by writing **deriving** *Show* after a Haskell datatype definition, the function *show* for this datatype is obtained for free. Thus in Haskell one can use a built-in polytypic function *show*, but *show* can not be expressed in the language, and one can not define alternative polytypic pretty printing functions.

This paper shows how one can define polytypic versions of the functions *show* and its inverse *read* that work for values of arbitrary regular datatypes. Again, the functions *show* and *read* are each others inverses by construction. Thus we externalize the definitions of these functions — in Haskell they are part of the compiler and can neither be inspected nor changed.

1.2 Constructing data conversion programs

The fundamental property of the four data conversion functions just described is that each of them has a right inverse with respect to forward composition. If we call the conversion function from a structured value *print* and the corresponding function back to the structured value *parse*, we have $print\ ;\ parse = id$. The other composition $parse\ ;\ print$ need not be id for all strings but it is id on the range of *print*: if $s = print\ x$ then $(parse\ ;\ print)\ s = (print\ ;\ parse\ ;\ print)\ x = (id\ ;\ print)\ x = print\ x = s$. The behavior of $parse\ ;\ print$ for other values is not specified. In the rest of the paper we will write just inverse, when we really mean right inverse. This is a very common specification pattern: all data conversion problems are specified as pairs of inverse functions with some additional properties.

In this paper, the driving force behind the definitions of the functions *print* and *parse* is inverse function construction. Thus correctness of *print* and *parse* is guaranteed by construction. Interestingly, when we forced ourselves to only construct pairs of inverse functions, we managed to reduce the size and complexity of the resulting programs considerably compared with our previous attempts.

The conversion programs are expressed using arrows — John Hughes' suggestion for generalizing monads [9]. The arrow combinators can be seen as defining a small (impure) functional language embedded in (pure) Haskell. We use constructor classes to allow for varying interpretations of this embedded language. Thus the conversion programs are implicitly parametrized with respect to the choice of implementation and semantics for this embedded language, and the laws needed to prove the correctness of the conversion programs can be seen as restrictions on the possible implementations.

This paper has the following goals:

- construct a number of polytypic programs for data conversion problems, together with their inverses;
- show how to construct and calculate with polytypic functions.

The implementation of the data conversion programs as PolyP code can be obtained from the polytypic programming WWW page [13].

The rest of this paper is organized as follows. Section 2 briefly introduces polytypic programming. Section 3 constructs polytypic programs for separating a datatype value into its shape and its contents, and for combining shape and contents back to the original value. Section 4 introduces an abstract function concept called arrows, which is used extensively in the following sections. Section 5 defines arrow traversals in two directions and proves that they are inverses. Section 6 sketches the construction and correctness proof of the packing program. Section 7 constructs polytypic programs for showing and reading values of datatypes. Section 8 defines instances of the various arrow classes. Section 9 concludes with an overview of the results, a discussion and some suggestions for future work.

2 Polytypic programming

The data conversion functions constructed in this paper are polytypic functions. This section briefly introduces polytypic functions in the context of the Haskell extension PolyP [11], and defines some basic polytypic concepts used in the paper. We assume that the reader is familiar with the initial algebra approach to datatypes [17], and not completely unfamiliar with polytypic programming. For an introduction to polytypic programming, see [1].

2.1 Notation

We use Haskell [20] notation with a few exceptions for notational convenience. We have already introduced $(\,;)$ for forward composition (that is, $f \;;\; g = g \circ f$) and we sometimes write function names starting with a capital, although Haskell only allows a lower case letter. We use "condensed" operators: for example, the operator `(-+-)` is written $(+)$. As a reminder of the syntax of Haskell we start with a few definitions that will be used in the sequel. The type constructor *Either* constructs a binary sum type, with $l \triangledown r$ as a shorthand notation for case analysis (written *either* l r in Haskell).

$$\textbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

$$\begin{array}{l}
(\triangledown) \quad :: (a \to c) \to (b \to c) \to (\mathit{Either}\ a\ b \to c) \\
l \triangledown r = \lambda x \to \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Left}\ a \quad \longrightarrow l\ a \\
\qquad\qquad\qquad \mathit{Right}\ b \longrightarrow r\ b
\end{array}$$

The call $f + g$ is used to apply either f or g inside *Left* or *Right*.

$$
\begin{aligned}
&(+) \quad :: (a \to c) \to (b \to d) \to (\mathit{Either}\ a\ b \to \mathit{Either}\ c\ d) \\
&f + g = (f\ ;\ \mathit{Left}) \triangledown (g\ ;\ \mathit{Right})
\end{aligned}
$$

The binary product type and its elements are written as pairs (a, b) and the duals of $(\triangledown)$ and $(+)$ are $(\vartriangle)$ and $(*)$.

$$
\begin{aligned}
&(\vartriangle) \quad :: (a \to b) \to (a \to c) \to (a \to (b, c)) \\
&f \vartriangle g = \lambda x \to (f\ x, g\ x)
\end{aligned}
$$

$$
\begin{aligned}
&(*) \quad :: (a \to c) \to (b \to d) \to ((a, b) \to (c, d)) \\
&f * g = (\mathit{fst}\ ;\ f) \vartriangle (\mathit{snd}\ ;\ g)
\end{aligned}
$$

We will often use Haskell's class system [16] to write generic overloaded code. The type of an overloaded function has the form *context* $\Rightarrow$ *type* where *context* lists the class constraints the variables in the type must satisfy. An example is $\mathit{sort} :: \mathit{Ord}\ a \Rightarrow [a] \to [a]$ where a is restricted to be in the class *Ord* of types with a comparison operator.

2.2 Functors and datatypes

A polytypic function is a function parametrized on type constructors. Polytypic functions are defined either by induction on the structure of user-defined type constructors, or defined in terms of other polytypic (and non-polytypic) functions. In the definition of a function that works for an arbitrary (as yet unknown) datatype we cannot use the constructors to build values, nor pattern match against values. Instead, PolyP provides two built-in functions, *in* and *out*, to construct and destruct a value of an arbitrary so-called regular datatype from and to its top level components. As an example, instances of functions *in* and *out* on the datatype *Tree a* are presented in Figure 1.

A datatype $d\ a$ is regular (satisfies $\mathit{Regular}\ d$) if it is not mutually recursive,

$$
\begin{aligned}
&\mathbf{type}\ \mathit{FTree}\ p\ r = \mathit{Either}\ p\ (r, r) \\
\\
&\mathit{in}_{\mathit{Tree}} :: \mathit{FTree}\ a\ (\mathit{Tree}\,a) \to \mathit{Tree}\ a \\
&\mathit{in}_{\mathit{Tree}}\ \ (\mathit{Left}\ x) \qquad\quad = \mathit{Leaf}\ x \\
&\mathit{in}_{\mathit{Tree}}\ \ (\mathit{Right}\ (l, r)) \quad = \mathit{Bin}\ l\ r \\
\\
&\mathit{out}_{\mathit{Tree}} :: \mathit{Tree}\ a \quad \to \mathit{FTree}\ a\ (\mathit{Tree}\ a) \\
&\mathit{out}_{\mathit{Tree}}\ \ (\mathit{Leaf}\ x) \ = \mathit{Left}\ x \\
&\mathit{out}_{\mathit{Tree}}\ \ (\mathit{Bin}\ l\ r) = \mathit{Right}\ (l, r)
\end{aligned}
$$

Figure 1. Instances of *in* and *out* for *Tree*.

contains no function spaces, and if the argument of the type constructor d is the same on the left- and right-hand side of its definition. Every regular datatype $d\ a$ in Haskell is equivalent to the fixed point of a functor. PolyP provides a type constructor $\mathit{FunctorOf}\ d$ (we use Φ_d as a shorthand) for this functor and defines in and out as the fold and unfold isomorphisms showing that $d\ a$ and $\Phi_d\ a\ (d\ a)$ are isomorphic.

$$\begin{aligned} \mathit{in}_d &:: \mathit{Regular}\ d \ \Rightarrow\ \Phi_d\ a\ (d\ a) \to d\ a \\ \mathit{out}_d &:: \mathit{Regular}\ d \ \Rightarrow\ d\ a \to \Phi_d\ a\ (d\ a) \end{aligned}$$

We call Φ_d a pattern functor as it is used to capture the recursion pattern of a datatype, for example, a list is either empty or contains one element and a recursive occurrence of a list. This top level structure is captured by the definition of the pattern functor $\mathit{FList}\ p\ r = \mathit{Either}\ ()\ (p, r)$. We represent pattern functors in a variable free form by introducing a number of functor constructors: with Par for the datatype parameter, Rec for the recursive parameter, Empty for the empty product and $(+)$ and $(*)$ for lifted versions of Either and $(,)$ we can write $\Phi_{\mathit{List}} = \mathit{Empty} + (\mathit{Par} * \mathit{Rec})$. The pattern functor of the datatype $\mathit{Tree}\ a$ is in a similar way represented by $\Phi_{\mathit{Tree}} = \mathit{Par} + (\mathit{Rec} * \mathit{Rec})$. As a last example, the datatype $\mathit{Rose}\ a$ of rose trees over a:

$$\mathbf{data}\ \mathit{Rose}\ a = \mathit{Node}\ a\ (\mathit{List}\ (\mathit{Rose}\ a))$$

has the pattern functor $\Phi_{\mathit{Rose}} = \mathit{Par} * (\mathit{List}\ @\ \mathit{Rec})$, where @ is functor composition. In general, PolyP's pattern functors are generated by the following grammar:

$$f, g, h \ ::= \ g + h \ \mid \ g * h \ \mid \ \mathit{Empty} \ \mid \ \mathit{Par} \ \mid \ \mathit{Rec} \ \mid \ d\ @\ g \ \mid \ \mathit{Const}\ t$$

where d generates regular datatype constructors, and t generates monomorphic types. The pattern functor $\mathit{Const}\ t$ denotes a constant functor with value t. The type context $\mathit{Bifunctor}\ f \ \Rightarrow$ is used to indicate that f is a pattern functor.

Regular datatypes are fixed points of pattern functors: $d\ a \cong \mu(\Phi_d\ a)$. As the functor Φ_d may refer to other (previously defined) regular datatypes in the $d@g$ case, the grammar for regular datatypes is mutually recursive with that for pattern functors. This means that most polytypic definitions are given as two mutually recursive bindings — one for the datatype level and one for the pattern functor level. Similarly, laws for polytypic functions are often proved by mutual induction over the grammars for regular datatypes and pattern functors. This induction is well-founded as a datatype can only refer to a datatype that is defined earlier.

The pattern functor of a Haskell datatype with n constructors is an n-ary sum (of products) on the outermost level. In PolyP this sum is represented by a nested binary sum, which associates to the right. (This representation is used in Section 6.1.) PolyP provides a few built-in polytypic functions to query datatypes and their value constructors for information which is useful when printing and parsing. The polytypic value $constructors_d$ gives a list of representations of the constructors of the datatype $d\ a$ and $noOfCons_d$ gives the number of constructors (the length of the list).

$$
\begin{array}{lll}
constructors_d & :: & [Constructor] \\
noOfCons_d & :: & Int \\
noOfCons_d & = & length\ constructors_d
\end{array}
$$

In these definitions, the subscript d cannot be inferred by the system, but must be supplied through explicit type information. The abstract type *Constructor* has selectors for the name and the precedence level of the constructor.

$$
\begin{array}{l}
name :: Constructor \rightarrow String \\
prec\ \ :: Constructor \rightarrow Prec \\
\mathbf{type}\ \ Prec = Int
\end{array}
$$

In Haskell, all infix operators have a precedence level between 0 and 9 where higher precedence means tighter binding and less need for disambiguating parentheses. It turns out that it makes sense to define a precedence level also for prefix constructors: nullary constructors are at level 10 (they never need to be surrounded by parentheses) and other constructors are at level 9 (this can be seen as the precedence level of the invisible infix application operator).

2.3 The polytypic construct

Using the **polytypic** construct a polytypic function can be defined by induction over the structure of pattern functors. As an example we take the function *flatten* defined in Figure 2. Note that:

- the subscripts indicating the type are included for readability and are not part of the definition (they are automatically inserted during type inference);
- the definitions of *map* and *map2* are given below.

Function *flatten* lists the values of type a in a value of type $d\ a$. As examples, we give the (edited) definitions, generated by PolyP, of functions *flatten* and

$$
\begin{array}{l}
\mathit{flatten}_d :: \mathit{Regular}\ d \ \Rightarrow\ d\ a \to [a] \\
\mathit{flatten}_d = \mathit{out}_d \ ;\ \mathit{map2}_{\Phi_d}\ \mathit{singleton}\ \mathit{flatten}_d \ ;\ \mathrm{FL}_{\Phi_d}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{polytypic}\ \mathrm{FL}_f :: f\ [a]\ [a] \to [a]} \\
\multicolumn{3}{l}{\quad = \textbf{case}\ f\ \textbf{of}} \\
\qquad g + h & \longrightarrow & \mathrm{FL}_g \triangledown \mathrm{FL}_h \\
\qquad g * h & \longrightarrow & \lambda(x, y) \to \mathrm{FL}_g\ x +\!\!+ \mathrm{FL}_h\ y \\
\qquad \mathit{Empty} & \longrightarrow & \lambda() \to [\,] \\
\qquad \mathit{Par} & \longrightarrow & \mathit{id} \\
\qquad \mathit{Rec} & \longrightarrow & \mathit{id} \\
\qquad d\ @\ g & \longrightarrow & \mathit{map}_d\ \mathrm{FL}_g \ ;\ \mathit{flatten}_d \ ;\ \mathit{concat} \\
\qquad \mathit{Const}\ t & \longrightarrow & \mathit{const}\ [\,]
\end{array}
$$

$$
\begin{array}{lll}
\mathit{singleton} :: & a \to & [a] \\
\mathit{singleton} & x \ = & [x]
\end{array}
$$

Figure 2. The definition of *flatten*

FL when instantiated on *Tree*.

$$
\begin{array}{l}
\mathit{flatten}_{\mathit{Tree}} :: \mathit{Tree}\ a \to [a] \\
\mathit{flatten}_{\mathit{Tree}} = \mathit{out}_{\mathit{Tree}} \ ;\ \mathit{map2}_{\mathit{FTree}}\ \mathit{singleton}\ \mathit{flatten}_{\mathit{Tree}} \ ;\ \mathrm{FL}_{\mathit{FTree}}
\end{array}
$$

$$
\begin{array}{lll}
\mathrm{FL}_{\mathit{FTree}} :: \mathit{FTree}\ [a]\ [a] & \to & [a] \\
\mathrm{FL}_{\mathit{FTree}}\quad (\mathit{Left}\ \mathit{xs}) & = & \mathit{xs} \\
\mathrm{FL}_{\mathit{FTree}}\quad (\mathit{Right}\ (\mathit{xs}, \mathit{ys})) & = & \mathit{xs} +\!\!+ \mathit{ys}
\end{array}
$$

2.4 Polytypic map functions

For every regular datatype $d\ a$ we define map_d, that takes a function $p :: a \to b$ and a value $x :: d\ a$, and applies p to all a values in x, giving a value of type $d\ b$. Similarly, for every pattern functor f, we define $\mathit{map2}_f$, that takes two functions $p :: a \to c$ and $r :: b \to d$ and a value $x :: f\ a\ b$, and applies p to all a values in x, and r to all b values in x, giving a value of type $f\ c\ d$. The definitions of *map* and *map2* can be found in Figure 3. As examples, we give the (edited) definitions, generated by PolyP, of functions *map* and *map2* on the datatype $\mathit{Tree}\ a$.

$$
\begin{array}{lll}
\mathit{map}_{\mathit{Tree}} & :: & (a \to b) \to (\mathit{Tree}\ a \to \mathit{Tree}\ b) \\
\mathit{map}_{\mathit{Tree}}\ p & = & \mathit{out}_{\mathit{Tree}} \ ;\ \mathit{map2}_{\mathit{FTree}}\ p\ (\mathit{map}_{\mathit{Tree}}\ p) \ ;\ \mathit{in}_{\mathit{Tree}}
\end{array}
$$

$$
\begin{array}{l}
\mathit{map2}_{\mathit{FTree}} :: (a \to c) \to (b \to d) \to \mathit{FTree}\ a\ b \to \mathit{FTree}\ c\ d \\
\mathit{map2}_{\mathit{FTree}}\ p\ p'\ (\mathit{Left}\ x) \qquad\quad = \mathit{Left}\ (p\ x) \\
\mathit{map2}_{\mathit{FTree}}\ p\ p'\ (\mathit{Right}\ (s, t)) = \mathit{Right}\ (p'\ s, p'\ t)
\end{array}
$$

$$
\begin{array}{lll}
map_d & :: & Regular\ d\ \Rightarrow\ (a \to b) \to (d\ a \to d\ b) \\
map_d\ p & = & out_d\ ;\ map2_{\Phi_d}\ p\ (map_d\ p)\ ;\ in_d
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic}\ map2_f\ ::\ (a \to c) \to (b \to d) \to (f\ a\ b \to f\ c\ d) \\
\quad = \lambda p\ p' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lll}
g + h & \longrightarrow & map2_g\ p\ p' + map2_h\ p\ p' \\
g * h & \longrightarrow & map2_g\ p\ p' * map2_h\ p\ p' \\
Empty & \longrightarrow & id \\
Par & \longrightarrow & p \\
Rec & \longrightarrow & p' \\
d\ @\ g & \longrightarrow & map_d\ (map2_g\ p\ p') \\
Const\ t & \longrightarrow & id
\end{array}
\end{array}
$$

Figure 3. The definition of *map* and *map2*

In the following sections we will construct pairs of inverse functions. The proof of the fact that these pairs of functions are inverses is by calculation: using laws for polytypic functions we calculate that the composition of the pair of functions is the identity. As examples of laws for polytypic functions we present the laws expressing that *map* and *map2* are functors:

$$
\begin{array}{ll}
map_d\ id & = id \\
map_d\ (p\ ;\ u) & = map_d\ p\ ;\ map_d\ u \\
map2_f\ id\ id & = id \\
map2_f\ (p\ ;\ u)\ (p'\ ;\ u') & = map2_f\ p\ p'\ ;\ map2_f\ u\ u'
\end{array}
$$

These laws are easily proved from corresponding laws for $(+)$ and $(*)$ by induction over the structure of regular datatypes.

In the rest of the paper we always assume that $d\ a$ is a regular datatype and that f is a pattern functor, but we omit the contexts ($Regular\ d\ \Rightarrow$ or $Bifunctor\ f\ \Rightarrow$) from the types for brevity.

3 Shape

The shape of a value is its structure without its contents. This section defines functions for separating a datatype value into its shape and its contents, and for combining shape and contents to a datatype value. Furthermore, it proves that the composition of these functions is the identity.

Using the functions *flatten* and *map*, defined in the previous section, it is easy to define a function *separate* that separates a datatype value into its shape and its contents.

$$\begin{array}{lcl} separate & :: & Regular\ d \Rightarrow d\ a \to (d\ (), [a]) \\ separate\ x & = & (shape\ x, flatten\ x) \\ shape & :: & Regular\ d \Rightarrow d\ a \to d\ () \\ shape & = & map\ (const\ ()) \end{array}$$

It is more difficult to define the function *combine*, the inverse of function *separate*. A standard implementation of function *combine* traverses the shape, carrying around the content list, and inserts one element from the list at each of the parameter positions in the shape. Because it is not easy to prove that such a function is the inverse of function *separate*, we redefine function *separate* to make the inverse construction straightforward.

The preceding definition of function *separate* traverses its input datatype value twice: once with *map* (*const* ()), and once with *flatten*. We can fuse these two traversals into a single traversal that carries around an accumulating state parameter. This traversal is carried out by a function similar to *map* which we call an arrow map. The arrow map takes as an argument a function, in this case the function *put*, which at each parameter position prepends the element to the accumulating list, and replaces the element by the empty tuple. To avoid 'pollution' of the types with state information, we introduce a new type constructor *SA* for functions that side-effect on a state.

$$\mathbf{data}\ SA\ s\ a\ b = SA\ ((a, s) \to (b, s))$$

We use the notation $a \leadsto_s b$ for the type $SA\ s\ a\ b$. Using this type and an arrow map called *mapAr*, we obtain the following definition for function *separate*:

$$\begin{array}{lcl} separate & :: & d\ a \leadsto_{[a]} d\ () \\ separate & = & mapAr\ put \\ put & :: & a \leadsto_{[a]} () \\ put & = & SA\ (\lambda(a, as) \to ((), a : as)) \\ mapAr & :: & (a \leadsto_s b) \to (d\ a \leadsto_s d\ b) \end{array}$$

where *mapAr* is defined in Section 5. The *r* in *mapAr* denotes the direction of the traversal: *mapAr* is a right to left traversal. This means that given a tree node with two subtrees, function *mapAr* first traverses the right subtree, and then the left subtree. Direction doesn't matter for normal maps, but for maps that carry around and update a state direction is important. For *separate* we could have used $put' = SA\ (\lambda(a, as) \rightarrow ((), as +\!\!+ [a]))$ and the left to right traversal, *mapAl*, but it turns out that the (somewhat counterintuitive) right to left traversal with *put* is lazier, more efficient and easier to prove correct.

3.2 Function combine

Using the left to right traversing variant of the arrow map, *mapAl*, we can write the inverse of *separate*, called *combine*, as follows.

$$\begin{aligned} &combine :: d\ () \rightsquigarrow_{[a]} d\ a \\ &combine = mapAl\ get \\[4pt] &get \quad :: () \rightsquigarrow_{[a]} a \\ &get \quad = SA\ (\lambda((), a : as) \rightarrow (a, as)) \\[4pt] &mapAl \ :: (a \rightsquigarrow_s b) \rightarrow (d\ \ a \rightsquigarrow_s d\ \ b) \end{aligned}$$

It remains to define the arrow maps, and to prove that *combine* is the inverse of *separate*, that is, *separate* followed by *combine* is the identity. Note that, due to the constructor *SA*, we cannot use normal function composition for values of type $a \rightsquigarrow_s b$. Instead we define a new composition operator ($\ggg$):

$$\begin{aligned} &(\ggg) \ :: \ (a \rightsquigarrow_s b) \rightarrow (b \rightsquigarrow_s c) \rightarrow (a \rightsquigarrow_s c) \\ &SA\ f \ggg SA\ g \qquad\qquad\qquad = SA\ (f\ ;\ g) \end{aligned}$$

It is easy to see that *get* is the inverse of *put*, but we include the proof to introduce the notation we use for calculational equality proofs.

$$\begin{aligned} &\quad put \ggg get \\ &= \quad \{\text{Definitions of } get \text{ and } put\} \\ &\quad SA\ (\lambda(a, as) \rightarrow ((), a : as)) \ggg SA\ (\lambda((), a : as) \rightarrow (a, as)) \\ &= \quad \{\text{Definition of } (\ggg)\} \\ &\quad SA\ ((\lambda(a, as) \rightarrow ((), a : as))\ ;\ (\lambda((), a : as) \rightarrow (a, as))) \\ &= \quad \{\text{Simplification}\} \\ &\quad SA\ id \end{aligned}$$

Here $SA\ id\ ::\ a \rightsquigarrow_{[a]} a$ is the identity on *SA* and the operator (=) is equality on *SA*.

3.3 Function combine is the inverse of separate

The main ingredient of the proof that *combine* is the inverse of *separate* is a law about inverting arrow maps. More specifically, we have that *mapAl* is the inverse of *mapAr* provided the arguments of the maps are inverses:

$$\begin{aligned} &mapAr :: (a \leadsto_s b) \to (d\ \ a \leadsto_s d\ \ b) \\ &mapAl\ :: (b \leadsto_s a) \to (d\ \ b \leadsto_s d\ \ a) \end{aligned}$$

$$(1)\quad p \ggg u = SA\ id \quad \Rightarrow \quad mapAr\ p \ggg mapAl\ u = SA\ id$$

Using this law (which is proved in Section 5) we have:

$$\begin{aligned} & separate \ggg combine \\ = \quad & \quad \{\text{Definitions of } separate \text{ and } combine\} \\ & mapAr\ put \ggg mapAl\ get \\ = \quad & \quad \{\text{Law (1) and } put \ggg get = SA\ id\ \} \\ & SA\ id \end{aligned}$$

This proves the correctness of functions *separate* and *combine*.

4 Arrows and laws

The previous section defines functions that side-effect on a state. Side effecting functions are often modeled as monadic functions, $f\ ::\ a\ \to\ M\ \ b$ for some monad M. However, the inverse proofs in this papers benefit from a more symmetrical abstraction. Therefore, we will use Hughes' abstract class for *arrows* [9], a generalization of monads. Just as with monads, combinator libraries can often be based on an arrow type, but arrows have a wider applicability than monads. We use a hierarchy of arrow classes as embedded domain specific languages for expressing data conversion programs. For additional motivation and background for using arrows, see the papers by Hughes and Paterson [9, 19].

4.1 Basic definitions and laws for arrows

To define the arrow maps and to prove (a generalization of) Law (1), we need a few combinators to construct and combine arrows, together with some laws that relate these combinators. We introduce the arrow combinators together with example implementations for the $SA\ s\ a\ b$ arrow (short form, $a \rightsquigarrow_s b$) but as we will see later, the types and the laws for the combinators form the signature of a general class of arrows. Thus, any program written using these combinators will automatically be parametrized over the instances of this class. In definitions and laws that hold for arbitrary arrows we write $a \rightsquigarrow b$ instead of $a \rightsquigarrow_s b$.

Lifting.

The function that lifts normal functions to functions that also take and return a state value is called arr.

$$\begin{aligned} &arr && :: (a \to b) \to (a \rightsquigarrow_s b) \\ &arr\ f && = SA\ (f \ast id) \end{aligned}$$

We will often write $\overrightarrow{f}$ instead of $arr\ f$. Function arr is a functor from the category of types and functions to the category of types and arrows: it distributes over composition (and trivially preserves the identity).

$$\overrightarrow{f} \ggg \overrightarrow{g} = \overrightarrow{f\ ;\ g}$$

Arrow composition.

Forward composition of arrows (defined already in Section 3.2) satisfies the usual laws: it is associative with $\overrightarrow{id}$ as its unit.

$$\begin{aligned} \overrightarrow{id} \ggg f &= f = f \ggg \overrightarrow{id} \\ (f \ggg g) \ggg h &= f \ggg (g \ggg h) \end{aligned}$$

We denote reverse composition with $(\lll)$, where $f \lll g = g \ggg f$.

Arrows between pairs.

Function $first$ applies an arrow to the first component of a pair, leaving the second component unchanged.

$$\begin{aligned} &first && :: (a \rightsquigarrow_s b) \to ((a, c) \rightsquigarrow_s (b, c)) \\ &first\ (SA\ f) && = SA\ (\lambda((a, c), s) \to \mathbf{let}\ (b, s') = f\ (a, s) \\ & && \qquad\qquad\qquad\qquad\quad\ \mathbf{in}\ \ ((b, c), s')) \end{aligned}$$

Function $first$ is a functor, that is, it preserves (arrow) identities and distributes over (arrow) composition.

$$\begin{aligned} first\ \overrightarrow{f} &= \overrightarrow{f \ast id} \\ first\ (f \ggg g) &= first\ f \ggg first\ g \end{aligned}$$

The corresponding function, *second*, that applies an arrow to the second component of a pair can be defined in terms of *first*:

$$
\begin{aligned}
&\mathit{second} \;\; :: (a \rightsquigarrow b) \to ((c,a) \rightsquigarrow (c,b)) \\
&\mathit{second}\ f = \overrightarrow{\mathit{swap}} \ggg \mathit{first}\ f \ggg \overrightarrow{\mathit{swap}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{swap} :: (a,b) \to (b,a) \\
&\mathit{swap} \quad (a,b) = (b,a)
\end{aligned}
$$

Using *first* and *second* we can define two candidates for product functors, but when the arrows simulate side-effects, neither of these are functors because they fail to preserve composition.

$$
\begin{aligned}
&(\ast\!\!\ast\!\!>), (<\!\!\ast\!\!\ast) :: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to ((a,b) \rightsquigarrow (c,d)) \\
&f \ast\!\!\ast\!\!> g \quad = \mathit{first}\ f \ggg \mathit{second}\ g \\
&f <\!\!\ast\!\!\ast\ g \quad = \mathit{second}\ g \ggg \mathit{first}\ f
\end{aligned}
$$

If one of the two arguments of *first* and *second* is side effect free (doesn't change the state), then *first* commutes with *second*. The canonical form of a side effect free arrow is $\overrightarrow{j}$ for some function j.

$$
\begin{aligned}
&\mathit{first}\ \overrightarrow{j} \ggg \mathit{second}\ g = \mathit{second}\ g \ggg \mathit{first}\ \overrightarrow{j} \\
&\mathit{second}\ \overrightarrow{j} \ggg \mathit{first}\ f = \mathit{first}\ f \ggg \mathit{second}\ \overrightarrow{j}
\end{aligned}
$$

Arrows with a choice.

We can view the arrow combinators as a very small embedded language. With the combinators defined thus far we can embed functions as arrows using $\overrightarrow{\cdot}$, we can plug arrows together using $(\ggg)$ and we can simulate a value environment by using *first*, *second* etc. However, we cannot write conditionals — there is no way to choose between different branches depending on the input.

We lift the operator $(\triangledown) \;::\; (a \to c) \to (b \to c) \to (\mathit{Either}\ a\ b \to c)$ to the arrow level to model a choice between different arrow branches. For state arrows the implementation is straightforward:

$$
\begin{aligned}
&(|||) \qquad\qquad :: (a \rightsquigarrow_s c) \to (b \rightsquigarrow_s c) \to (\mathit{Either}\ a\ b \rightsquigarrow_s c) \\
&\mathit{SA}\ f\ |||\ \mathit{SA}\ g = \mathit{SA}\ (\lambda(x,s) \to ((\lambda a \to f\ (a,s))\ \triangledown\ (\lambda b \to g\ (b,s)))\ x)
\end{aligned}
$$

As a simple exercise in arrow plumbing we define **if**-expressions:

$$
\begin{aligned}
&\mathit{ifA} \qquad :: (a \rightsquigarrow \mathit{Bool}) \to (a \rightsquigarrow b) \to (a \rightsquigarrow b) \to (a \rightsquigarrow b) \\
&\mathit{ifA}\ p\ t\ e = \overrightarrow{\mathit{dup}} \ggg \mathit{first}\ p \ggg \overrightarrow{\mathit{bool2Either}} \ggg (t\ |||\ e) \\
&\quad \textbf{where}\ \mathit{dup}\ a = (a,a) \\
&\qquad\qquad \mathit{bool2Either}\ (b,x) = \textbf{if}\ b\ \textbf{then}\ \mathit{Left}\ x\ \textbf{else}\ \mathit{Right}\ x
\end{aligned}
$$

The lifted variant of operator $(+)$ for arrows is defined by:

$$
\begin{aligned}
&(+\!\!+\!\!+) \quad :: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to (\mathit{Either}\ a\ b \rightsquigarrow \mathit{Either}\ c\ d)\\
&f +\!\!+\!\!+ g = (f \ggg \overrightarrow{\mathit{Left}}) \mid\mid\mid (g \ggg \overrightarrow{\mathit{Right}})
\end{aligned}
$$

Operator $(+\!\!+\!\!+)$ is a bifunctor on arrows — it preserves identities and distributes over composition.

$$
\begin{aligned}
&\overrightarrow{f} +\!\!+\!\!+ \overrightarrow{g} &&= \overrightarrow{f + g}\\
&(f +\!\!+\!\!+ g) \ggg (f' +\!\!+\!\!+ g') &&= (f \ggg f') +\!\!+\!\!+ (g \ggg g')
\end{aligned}
$$

This requirement is stronger (and thus permits fewer instances) than Hughes' [9] requirement.

4.2 A class for arrows

The type $SA\ s\ a\ b$ encapsulates functions from a to b that manipulate a state of type s. However, most of the programs and laws we want to express don't refer to the state. Therefore, we go one step further in the abstraction by introducing the Haskell constructor class *Arrow* [9,19]. An arrow type constructor $(\rightsquigarrow)$ is any two-parameter type constructor that supports the operations of the class *Arrow*. We require a number of laws to hold for the instances of the arrow class and for documentation purposes, we include these laws in the class definition although they can't be directly expressed in Haskell.

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Arrow}\ (\rightsquigarrow)\ \mathbf{where}\\
&\quad \mathit{arr}\ \ :: (a \to b) \to (a \rightsquigarrow b)\\
&\quad (\ggg) :: (a \rightsquigarrow b) \to (b \rightsquigarrow c) \to (a \rightsquigarrow c)\\
&\quad \mathit{first}\ :: (a \rightsquigarrow b) \to ((a, c) \rightsquigarrow (b, c))\\
&\\
&\quad \text{-- } \mathit{Laws}:\\
&\quad \overrightarrow{f} \ggg \overrightarrow{g} &&= \overrightarrow{f\ ;\ g}\\
&\quad \overrightarrow{id} \ggg f &&= f \quad = \quad f \ggg \overrightarrow{id}\\
&\quad (f \ggg g) \ggg h &&= f \ggg (g \ggg h)\\
&\quad \mathit{first}\ \overrightarrow{f} &&= \overrightarrow{f * id}\\
&\quad \mathit{first}\ (f \ggg g) &&= \mathit{first}\ f \ggg \mathit{first}\ g\\
&\quad \mathit{first}\ \overrightarrow{f} \ggg \mathit{second}\ g &&= \mathit{second}\ g \ggg \mathit{first}\ \overrightarrow{f}
\end{aligned}
$$

These laws are a subset of the laws postulated in Hughes' arrow paper [9], but they are sufficient for this paper.

For arrows with a choice operator, $(\mid\mid\mid)$, we define the subclass *ArrowChoice*. We include both the operator $(\mid\mid\mid)$ and $(+\!\!+\!\!+)$, but it is sufficient to define either of them in every instance thanks to the defaults. The default declarations are part of the Haskell class definition and can be seen as laws with immediate implementations.

$$
\begin{array}{l}
\textbf{class } \mathit{Arrow}\ (\leadsto)\ \Rightarrow\ \mathit{ArrowChoice}\ (\leadsto)\ \textbf{where} \\
\quad (+\!\!+\!\!+) :: (a \leadsto c) \to (b \leadsto d) \to (\mathit{Either}\ a\ b \leadsto \mathit{Either}\ c\ d) \\
\quad (|||)\ \ :: (a \leadsto c) \to (b \leadsto c) \to (\mathit{Either}\ a\ b \leadsto c) \\
\\
\quad \texttt{--}\ \mathit{Defaults}: \\
\quad f +\!\!+\!\!+ g = (f \ggg \overrightarrow{\mathit{Left}})\ |||\ (g \ggg \overrightarrow{\mathit{Right}}) \\
\quad f\ |||\ g\ \ = (f +\!\!+\!\!+ g) \ggg \overrightarrow{\mathit{id} \triangledown \mathit{id}} \\
\\
\quad \texttt{--}\ \mathit{Laws}: \\
\quad \overrightarrow{f} +\!\!+\!\!+ \overrightarrow{g} \qquad\qquad\qquad\ = \overrightarrow{f + g} \\
\quad (f +\!\!+\!\!+ g) \ggg (f' +\!\!+\!\!+ g') = (f \ggg f') +\!\!+\!\!+ (g \ggg g') \\
\quad (f\ |||\ g) \ggg h \qquad\qquad\ \ = (f \ggg h)\ |||\ (g \ggg h)
\end{array}
$$

This definition of the class *ArrowChoice* differs from the definition in Hughes' paper [9] in the following way. Hughes uses $\mathit{left} :: (a \leadsto b) \to (\mathit{Either}\ a\ c \leadsto \mathit{Either}\ b\ c)$ as the class member and defines $(|||)$ and $(+\!\!+\!\!+)$ in terms of *left*. Hughes' laws for *left* are very similar to the laws for *first*. For the proofs we need slightly stronger laws for $(|||)$ and $(+\!\!+\!\!+)$ than can be derived from the laws for *left* and therefore we choose to define this variant of the *ArrowChoice* class including the stronger requirement.

The type constructor $\mathit{SA}\ s$ is made an instance of *Arrow* and *ArrowChoice* by taking the definitions of *arr*, $(\ggg)$, *first*, $(|||)$ and $(+\!\!+\!\!+)$ from Section 4.1. Normal functions are trivially *Arrow*s and they support choice:

$$
\begin{array}{l}
\textbf{instance } \mathit{Arrow}\ (\to)\ \textbf{where} \\
\quad \mathit{arr}\ f\ \ \ = f \\
\quad f \ggg g = f\ ;\ g \\
\quad \mathit{first}\ f\ = f \ast \mathit{id} \\
\\
\textbf{instance } \mathit{ArrowChoice}\ (\to)\ \textbf{where} \\
\quad f +\!\!+\!\!+ g = f + g \\
\quad f\ |||\ g\ \ = f \triangledown g
\end{array}
$$

With the definitions from these instances, three of the laws from the *Arrow* and the *ArrowChoice* classes can be rewritten to a form which more clearly indicates that $\overrightarrow{\cdot}$ lifts composition, *first* and choice from normal functions to arrows:

$$
\begin{array}{l}
\overrightarrow{f} \ggg \overrightarrow{g} = \overrightarrow{f \ggg g} \\
\mathit{first}\ \overrightarrow{f}\ \ \ = \overrightarrow{\mathit{first}\ f} \\
\overrightarrow{f} +\!\!+\!\!+ \overrightarrow{g} = \overrightarrow{f +\!\!+\!\!+ g}
\end{array}
$$

Many side effecting computations can be captured by the *Arrow* signature, including all functions returning monadic results: we can define a *Kleisli* arrow for every Haskell *Monad* [22]:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Kleisli}\ m\ a\ b = \mathit{Kleisli}\ (a \to m\ b) \\[1ex]
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Arrow}\ (\mathit{Kleisli}\ m)\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{arr}\ f & = & \mathit{Kleisli}\ (\lambda a \to \mathit{return}\ (f\ a)) \\
\mathit{Kleisli}\ f \ggg \mathit{Kleisli}\ g & = & \mathit{Kleisli}\ (\lambda a \to f\ a \gg\!= g) \\
\mathit{first}\ (\mathit{Kleisli}\ f) & = & \mathit{Kleisli}\ (\lambda (a,c) \to f\ a \gg\!= \lambda b \to \mathit{return}\ (b,c))
\end{array} \\[1ex]
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{ArrowChoice}\ (\mathit{Kleisli}\ m)\ \textbf{where} \\
\quad \mathit{Kleisli}\ f \mathbin{|||} \mathit{Kleisli}\ g = \mathit{Kleisli}\ (f \triangledown g)
\end{array}
$$

Examples of type constructors which are monads are *Maybe*, [] (the list type constructor) and *Bag* where a value of type *Bag a* is an unordered collection of values of type a.

4.3 An inverse law for arrow products

The two product operators ($\ast\!\!\!\!\gg$) and ($\ll\!\!\!\ast$) are related by a general inverse law, which will turn out to be useful in the following sections. The law only requires the side-effects to be inverses. If $f \ggg f' = \overrightarrow{i}$, then the arrow f' un-does the side-effects of the arrow f, leaving just a side-effect free computation $\overrightarrow{i}$. If i is chosen to be id, then we regain the earlier inverse concept. The more general inverse concept will be used in the rest of the paper. The generalized inverse law for the product operators is:

$$
\begin{array}{l}
(\ll\!\ast) :: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to ((a,b) \rightsquigarrow (c,d)) \\
(\ast\!\gg) :: (c \rightsquigarrow a) \to (d \rightsquigarrow b) \to ((c,d) \rightsquigarrow (a,b))
\end{array}
$$

$$
(2) \quad f \ggg f' = \overrightarrow{i} \;\Rightarrow\; g \ggg g' = \overrightarrow{j} \;\Rightarrow\; (f \ll\!\ast g) \ggg (f' \ast\!\gg g') = \overrightarrow{i \ast j}
$$

By symmetry, the law is also true if ($\ll\!\ast$) and ($\ast\!\gg$) are swapped.

Perhaps a word on notation is appropriate here. We present the types of the product operators together with the inverse law, to stress that we are not dealing with just a pair of inverse functions, but rather with a triple containing two functions and a proof that they are inverses. We take a curried view of functions with two arguments, that is, they have type $a \to b \to c$ rather than $(a,b) \to c$. Similarly, we prefer to write a proof term with two premises as $P \Rightarrow Q \Rightarrow R$, instead of the more traditional $P \wedge Q \;\Rightarrow\; R$. Thus we stress that the components of the triple share the same structure: they take two arrows (two proofs) and return an arrow (a proof).

Proof: To prove (2), we assume $f \ggg f' = \overrightarrow{i}$ and $g \ggg g' = \overrightarrow{j}$ and calculate as follows:

$$
\begin{array}{ll}
 & (f \mathbin{<\!\!*} g) \ggg (f' \mathbin{*\!\!>} g') \\
= & \quad \{\text{Definitions of } <\!\!* \text{ and } *\!\!> \} \\
 & \mathit{second}\ g \ggg \mathit{first}\ f \ggg \mathit{first}\ f' \ggg \mathit{second}\ g' \\
= & \quad \{\mathit{first} \text{ is a functor}\} \\
 & \mathit{second}\ g \ggg \mathit{first}\ (f \ggg f') \ggg \mathit{second}\ g' \\
= & \quad \{\text{Assumption 1}\} \\
 & \mathit{second}\ g \ggg \mathit{first}\ \overrightarrow{i} \ggg \mathit{second}\ g' \\
= & \quad \{\overrightarrow{i} \text{ is side-effect free}\} \\
 & \mathit{first}\ \overrightarrow{i} \ggg \mathit{second}\ g \ggg \mathit{second}\ g' \\
= & \quad \{\mathit{second} \text{ is a functor}\} \\
 & \mathit{first}\ \overrightarrow{i} \ggg \mathit{second}\ (g \ggg g') \\
= & \quad \{\text{Assumption 2}\} \\
 & \mathit{first}\ \overrightarrow{i} \ggg \mathit{second}\ \overrightarrow{j} \\
= & \quad \{\mathit{first}, \mathit{second} \text{ and } (\ggg) \text{ preserve } \overrightarrow{\cdot}\} \\
 & \overrightarrow{(i \mathbin{-\!\!*\!\!-} id)\ ;\ (id \mathbin{-\!\!*\!\!-} j)} \\
= & \quad \{(-\!\!*\!\!-) \text{ is a bifunctor}\} \\
 & \overrightarrow{i \mathbin{-\!\!*\!\!-} j}
\end{array}
$$

□

4.4 Fixed point induction and arrows

Section 5 proves an inverse law (4) for arrow maps. A similar law for normal maps can be proved with the fusion law for catamorphisms but, unfortunately, the fusion law does not generalize to arrows. In the proof of the law for arrow maps we will use instead the following variant of fixed point induction:

Definition 1 *A relation P is* inclusive *if and only if for all chains of tuples* $(a_1^i, ..., a_n^i)$

$$(\forall\ i.\ P\ (a_1^i, ..., a_n^i)) \Rightarrow P\ (\textstyle\bigsqcup_i a_1^i, ..., \bigsqcup_i a_n^i)$$

Theorem 2 *Fixed point induction: [21, def. 6.26]*

For every inclusive *relation* P, *and for all functions* $i_1 \ldots i_n$:

$$(P\ (\bot, \ldots, \bot)\ \wedge\ \forall\ f_1 \ldots f_n.\ P\ (f_1, \ldots, f_n) \Rightarrow P\ (i_1\ f_1, \ldots, i_n\ f_n)) \\ \Rightarrow P\ (\mathit{fix}\ i_1, \ldots, \mathit{fix}\ i_n)$$

We can instantiate Theorem 2 to a form that is more suitable for our purposes: let $n = 3$ and let the (inclusive) relation $P(x, y, z) = x \ggg y = \overrightarrow{z}$. Treating the base case separately, the instance takes the following form:

$$(3)\quad \begin{array}{l} (\forall p', u', i'.\ p' \ggg u' = \overrightarrow{i'} \ \Rightarrow\ f\ p' \ggg g\ u' = \overrightarrow{h\ i'}) \\ \Rightarrow\ \ \mathit{fix}\ f \ggg \mathit{fix}\ g = \overrightarrow{\mathit{fix}\ h} \end{array}$$

For the base case $(\bot \ggg \bot = \overrightarrow{\bot})$ to hold universally we require $\overrightarrow{\cdot}$ and $(\ggg)$ to be strict. This requirement is trivially satisfied in a strict setting (in $CPO_\bot$ where all functions are strict) but for the proof it is sufficient to require strictness for the embedded language (the arrow class member functions) and not for the host language (Haskell). In fact, experiments with the implementation indicate that this requirement can be further weakened, although the full details remain to be investigated.

5 Traversals as arrow maps

In Section 3, *separate* and *combine* were defined using the arrow maps *mapAr* and *mapAl*. The arrow maps can be seen as simple data conversion programs, which change the contents but leave the shape of the data unchanged. Using the arrow combinators from Section 4 we can now define the arrow maps, and prove a generalization of (1): if u is the inverse of p, then a left traversal with u is the inverse of a right traversal with p.

$$\begin{array}{l} \mathit{mapAr} :: \mathit{ArrowChoice}\ (\rightsquigarrow)\ \Rightarrow\ (a \rightsquigarrow b) \to (d\ \ a \rightsquigarrow d\ \ b) \\ \mathit{mapAl}\ :: \mathit{ArrowChoice}\ (\rightsquigarrow)\ \Rightarrow\ (a \rightsquigarrow b) \to (d\ \ a \rightsquigarrow d\ \ b) \end{array}$$

$$(4)\quad p \ggg u = \overrightarrow{i} \quad \Rightarrow \quad \mathit{mapAr}_d\ p \ggg \mathit{mapAl}_d\ u = \overrightarrow{\mathit{map}_d\ i}$$

By symmetry the law holds also if mapAr_d and mapAl_d are swapped. The definitions of the arrow maps are obtained by a straightforward generalization of *map* to arrows.

$$\begin{array}{l} \mathit{mapAr}_d\ p\ = \overrightarrow{\mathit{out}_d} \ggg \mathrm{T}\scriptstyle{\mathrm{R}}_{\Phi_d}\ p\ (\mathit{mapAr}_d\ p) \ggg \overrightarrow{\mathit{in}_d} \\ \mathit{mapAl}_d\ u\ = \overrightarrow{\mathit{out}_d} \ggg \mathrm{T}\scriptstyle{\mathrm{L}}_{\Phi_d}\ u\ (\mathit{mapAl}_d\ u) \ggg \overrightarrow{\mathit{in}_d} \end{array}$$

$$
\begin{array}{l}
\textbf{polytypic}\ \textsc{Tr}_f :: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to (f\ a\ b \rightsquigarrow f\ c\ d) \\
\quad = \lambda p\ p' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad \begin{array}{lll}
g + h & \longrightarrow & \textsc{Tr}_g\ p\ p' +\!\!\!+\!\!\!+ \textsc{Tr}_h\ p\ p' \\
g * h & \longrightarrow & \textsc{Tr}_g\ p\ p' \mathrel{<\!\!*\!\!*} \textsc{Tr}_h\ p\ p' \\
\mathit{Empty} & \longrightarrow & \overrightarrow{\mathit{id}} \\
\mathit{Par} & \longrightarrow & p \\
\mathit{Rec} & \longrightarrow & p' \\
d\ @\ g & \longrightarrow & \mathit{mapAr}_d\ (\textsc{Tr}_g\ p\ p') \\
\mathit{Const}\ t & \longrightarrow & \overrightarrow{\mathit{id}}
\end{array}
\end{array}
$$

Figure 4. The definition of the right to left traversal $\textsc{Tr}$.

Functions $\textsc{Tr}$ and $\textsc{Tl}$ are the corresponding generalizations of *map2*. All functions used in the definition of *map2* are lifted to the arrow level. For all cases except the product functor case there is only one choice for a reasonable lifting, but when we lift the operator $(-\!\!*\!\!-)$ we have two possible choices: $(\mathrel{<\!\!*\!\!*})$ and $(\mathrel{*\!\!*\!\!>})$. This is the only difference between the two traversal functions: the right to left traversal, $\textsc{Tr}$, uses $(\mathrel{<\!\!*\!\!*})$ and the left to right traversal, $\textsc{Tl}$, uses $(\mathrel{*\!\!*\!\!>})$. Function $\textsc{Tr}$ is defined in Figure 4 but as function $\textsc{Tl}$ is almost identical its definition is omitted. Note that $\textsc{Tr}$ and $\textsc{Tl}$ are in general not functors, because $(\mathrel{<\!\!*\!\!*})$ and $(\mathrel{*\!\!*\!\!>})$ are not functors. Functions $\textsc{Tr}$ and $\textsc{Tl}$ satisfy the following inverse law:

$$
\begin{array}{l}
\textsc{Tr}_f :: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to (f\ a\ b \rightsquigarrow f\ c\ d) \\
\textsc{Tl}_f :: (c \rightsquigarrow a) \to (d \rightsquigarrow b) \to (f\ c\ d \rightsquigarrow f\ a\ b)
\end{array}
$$

$$(5)\quad p \ggg u = \overrightarrow{i} \;\Rightarrow\; p' \ggg u' = \overrightarrow{i'} \;\Rightarrow\; \textsc{Tr}_f\ p\ p' \ggg \textsc{Tl}_f\ u\ u' = \overrightarrow{\mathit{map2}_f\ i\ i'}$$

Note the close correspondence between this law and the inverse law for the product operators (2).

5.1 The arrow maps are inverses

The proof of Equation (4) can be interpreted either as fusing *mapAr* p with *mapAl* u to get a pure arrow $\overrightarrow{\mathit{map}_d\ i}$ or, equivalently, as splitting the function $\mathit{map}_d\ i$ into a composition of two arrow maps. The proof is by induction over the structure of a regular datatype $d\ a$. As the grammars for datatypes and pattern functors are mutually recursive we get two induction hypotheses. The datatype level hypothesis is, that Equation (4) holds for datatypes defined earlier, and the pattern functor level hypothesis is, that Equation (5) holds for the sub-functors. We rewrite the definitions of *mapAr*, *mapAl* and *map* to expose the top level fixed points:

$$
\begin{aligned}
\mathit{mapAr}\ p &= \mathit{fix}\ (\lambda p' \to \overrightarrow{\mathit{out}} \ggg \mathrm{T}\textsc{r}\ p\ p' \ggg \overrightarrow{\mathit{in}}) \\
\mathit{mapAl}\ u &= \mathit{fix}\ (\lambda u' \to \overrightarrow{\mathit{out}} \ggg \mathrm{T}\textsc{l}\ u\ u' \ggg \overrightarrow{\mathit{in}}) \\
\mathit{map}\ i &= \mathit{fix}\ (\lambda i' \to \mathit{out}\ ;\ \mathit{map2}\ i\ i'\ ;\ \mathit{in})
\end{aligned}
$$

We assume $p \ggg u = \overrightarrow{i}$ and calculate as follows:

$$
\begin{array}{ll}
& \mathit{mapAr}\ p \ggg \mathit{mapAl}\ u = \overrightarrow{\mathit{map}\ i} \\
\Leftarrow & \quad \{\text{Definitions of } \mathit{mapAr},\ \mathit{mapAl}, \text{ fixed point law (3)}\} \\
& p' \ggg u' = \overrightarrow{i'} \Rightarrow \\
& \overrightarrow{\mathit{out}} \ggg \mathrm{T}\textsc{r}\ p\ p' \ggg \overrightarrow{\mathit{in}} \ggg \overrightarrow{\mathit{out}} \ggg \mathrm{T}\textsc{l}\ u\ u' \ggg \overrightarrow{\mathit{in}} = \\
& \overrightarrow{\mathit{out}\ ;\ \mathit{map2}\ i\ i'\ ;\ \mathit{in}} \\
\equiv & \quad \{\overrightarrow{f} \ggg \overrightarrow{g} = \overrightarrow{f;g},\ \mathit{in}\ ;\ \mathit{out} = \mathit{id},\ f \ggg \overrightarrow{\mathit{id}} = f\} \\
& p' \ggg u' = \overrightarrow{i'} \Rightarrow \\
& \overrightarrow{\mathit{out}} \ggg \mathrm{T}\textsc{r}\ p\ p' \ggg \mathrm{T}\textsc{l}\ u\ u' \ggg \overrightarrow{\mathit{in}} = \overrightarrow{\mathit{out}} \ggg \overrightarrow{\mathit{map2}\ i\ i'} \ggg \overrightarrow{\mathit{in}} \\
\Leftarrow & \quad \{\text{Law (5) and the assumption: } p \ggg u = \overrightarrow{i}\} \\
& \mathit{True}
\end{array}
$$

We prove Law (5) by induction over the structure of the pattern functor f. Because there are seven constructors for functors, we have to verify seven cases. Although this is laborious, we want to show at least one complete proof of a statement about polytypic functions.

The sum case, $g + h$:

$$
\begin{array}{ll}
& \mathrm{T}\textsc{r}_{g+h}\ p\ p' \ggg \mathrm{T}\textsc{l}_{g+h}\ u\ u' \\
= & \quad \{\text{Definitions}\} \\
& (\mathrm{T}\textsc{r}_g\ p\ p' +\!\!\!+\!\!\!+ \mathrm{T}\textsc{r}_h\ p\ p') \ggg (\mathrm{T}\textsc{l}_g\ u\ u' +\!\!\!+\!\!\!+ \mathrm{T}\textsc{l}_h\ u\ u') \\
= & \quad \{(+\!\!\!+\!\!\!+) \text{ is a bifunctor}\} \\
& (\mathrm{T}\textsc{r}_g\ p\ p' \ggg \mathrm{T}\textsc{l}_g\ u\ u') +\!\!\!+\!\!\!+ (\mathrm{T}\textsc{r}_h\ p\ p' \ggg \mathrm{T}\textsc{l}_h\ u\ u') \\
= & \quad \{\text{Induction hypothesis (5) (twice)}\} \\
& \overrightarrow{\mathit{map2}_g\ i\ i'} +\!\!\!+\!\!\!+ \overrightarrow{\mathit{map2}_h\ i\ i'} \\
= & \quad \{\overrightarrow{f} +\!\!\!+\!\!\!+ \overrightarrow{g} = \overrightarrow{f + g}, \text{ definition of } \mathit{map2}_{g+h}\} \\
& \overrightarrow{\mathit{map2}_{g+h}\ i\ i'}
\end{array}
$$

The product case, $g * h$:

$$
\begin{array}{ll}
& \mathrm{T}\textsc{r}_{g*h}\ p\ p' \ggg \mathrm{T}\textsc{l}_{g*h}\ u\ u' \\
= & \quad \{\text{Definitions}\}
\end{array}
$$

$$(\text{T}\textsc{r}_g\ p\ p' \mathbin{\lessdot\!\!\ast} \text{T}\textsc{r}_h\ p\ p') \ggg (\text{T}\textsc{l}_g\ u\ u' \mathbin{\ast\!\!\gtrdot} \text{T}\textsc{l}_h\ u\ u')$$
$$= \quad \{\text{Inverse law for products (2), induction hypothesis (5) (twice)}\}$$
$$\overrightarrow{map2_g\ i\ i' \ast map2_h\ i\ i'}$$
$$= \quad \{\text{definition of } map2_{g*h}\}$$
$$\overrightarrow{map2_{g*h}\ i\ i'}$$

The empty case, ***Empty*****:**

$$\text{T}\textsc{r}_{Empty}\ p\ p' \ggg \text{T}\textsc{l}_{Empty}\ u\ u'$$
$$= \quad \{\text{Definitions}\}$$
$$\overrightarrow{id} \ggg \overrightarrow{id}$$
$$= \quad \{\overrightarrow{id} \text{ is the unit of } \ggg\}$$
$$\overrightarrow{id}$$
$$= \quad \{\text{Definition of } map2_{Empty}\}$$
$$\overrightarrow{map2_{Empty}\ i\ i'}$$

The constant case *Const* t is proved in exactly the same way as the empty case — the calculation is omitted.

The parameter case, ***Par*****:**

$$\text{T}\textsc{r}_{Par}\ p\ p' \ggg \text{T}\textsc{l}_{Par}\ u\ u'$$
$$= \quad \{\text{Definitions}\}$$
$$p \ggg u$$
$$= \quad \{\text{Assumption}\}$$
$$\overrightarrow{i}$$
$$= \quad \{\text{Definition of } map2_{Par}\}$$
$$\overrightarrow{map2_{Par}\ i\ i'}$$

The recursive case, *Rec*, is proved in exactly the same way as the parameter case — the calculation is omitted.

The composition case, $d\ @\ g$**:**

$$\text{T}\textsc{r}_{d@g}\ p\ p' \ggg \text{T}\textsc{l}_{d@g}\ u\ u'$$
$$= \quad \{\text{Definitions}\}$$
$$mapAr_d\ (\text{T}\textsc{r}_g\ p\ p') \ggg mapAl_d\ (\text{T}\textsc{l}_g\ u\ u')$$
$$= \quad \left\{\begin{array}{l}\text{The top level induction hypothesis (4) is}\\ f \ggg g = \overrightarrow{h} \quad \Rightarrow \quad mapAr\ f \ggg mapAl\ g = \overrightarrow{map\ h}\\ \text{where we take } f = \text{T}\textsc{r}_g\ p\ p',\ g = \text{T}\textsc{l}_g\ u\ u' \text{ and } h = map2\ i\ i'\\ \text{and induction hypothesis (5) is precisely } f \ggg g = \overrightarrow{h}.\end{array}\right.$$

$$\overrightarrow{map_d\ (map2_g\ i\ i')}$$
$$= \quad \{\text{Definition of } map2_{d@g}\}$$
$$\overrightarrow{map2_{d@g}\ i\ i'}$$

This completes the proof.

In the conclusions we discuss (how to simplify) proving statements about polytypic functions.

6 Packing

This section sketches the construction and correctness proof of a polytypic packing program. The basic idea of the packing program is simple: given a datatype value (an abstract syntax tree), construct a compact (bit stream) representation of the abstract syntax tree. For example, the following rather artificial binary tree, called *treeShape* in the introductory section,

$$\begin{aligned}&treeShape :: Tree\ ()\\&treeShape = Bin\ (Bin\ (Leaf\ ())\ (Bin\ (Leaf\ ())\ (Leaf\ ())))\ (Leaf\ ())\end{aligned}$$

can be pretty-printed to a text representation of *treeShape* requiring 55 bytes. However, because the datatype *Tree a* has only two constructors, each constructor can be represented by a single bit. Furthermore, the datatype () has only one constructor, so the single element (also written ()) can be represented by 0 bits. Thus we get the following representations:

```
Bin (Bin (Leaf ()) (Bin (Leaf ()) (Leaf ()))) (Leaf ())
 1    1    0         1    0         0            0
```

The compact representation consists of 7 bits, so only 1 byte is needed to store this tree. In fact, the pretty-printed text of a value of type *Tree* () is asymptotically 64 times bigger than the compact representation.[1] Of course, this is an unusually simple datatype, but the average case is still very compact.

Given a datatype value, the polytypic packing function prepends the compact representation of the value to a state, on which it side effects. Let *Text* be the

[1] A value of type *Tree* () with n leaves has $n-1$ internal nodes. A leaf is printed as the seven character string `"Leaf ()"` and a node as `"Bin ("`, left subtree, `") ("`, right subtree, `")"` — a total of nine characters per node. Thus the pretty printed string representation of a tree contains exactly $7n + 9(n-1) = 16n - 9$ bytes while the compact representation with one bit per constructor contains $2n-1$ bits. The ratio is then $8(16n-9)/(2n-1) \approx 8(16n-8)/(2n-1) = 64$.

type of packed values, for example *String* or [*Bit*]. Then the packing function can be implemented using the state arrow type constructor *SA Text*, but we will keep the arrow type abstract and only require that it supports packing of constructors.

To pack a value of type $d\ a$ we need a function that can pack values of type a. We could use *separate* and *combine* to reduce the packing problem to packing the structure and the contents separately, but instead we parametrize on the element level (un)packing function. Note that with Hinze style polytypism [7], this parametrization comes for free.

Our goal is to construct two functions and a proof:

- A function *pack* ('polytypic packing') that takes an element level packer to a datatype level packer.

  $$pack\ ::\ (a \rightsquigarrow ()) \rightarrow (d\ a \rightsquigarrow ())$$

  For example, the function that packs the tree $treeShape :: Tree\ ()$ is obtained by instantiating the polytypic function *pack* on *Tree* and applying the instance to a (trivial) packing program for the type ().
- A function *unpack* ('polytypic unpacking') that takes an unpacker on the element level a to an unpacker on the datatype level $d\ a$:

  $$unpack\ ::\ (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow d\ a)$$

  For the *Tree* () example the element level parsing program is a function that parses nothing, and returns (), the value of type ().
- A proof that if p and u are inverses on the element level a, then $pack\ p$ and $unpack\ u$ are inverses on the datatype level $d\ a$.

Representing constructors.

To construct the printer and the parser we need a little more structure than provided by the *Arrow* class – we need a way of handling constructors. Because a constructor can be coded by a single natural number, we only need operations for printing and parsing constructor numbers. With $Text = [Int]$ and the arrow *SA Text* the operations *put* and *get* from Section 3 would work, and the printing algorithm constructed in the following section would in its simplest form just output a list of numbers given an argument tree of any type. A better solution is to code these numbers as bits and here we have some choices on how to proceed. We could decide on a fixed maximal size for numbers and store them using their binary representation but, as most datatypes have few constructors, this would waste space. Instead we determine statically the total number of constructors in the datatype and pass this number as the first argument to *packCon* and *unpackCon*.

$$
\begin{aligned}
&\textbf{class}\ \mathit{ArrowChoice}\ (\leadsto)\ \Rightarrow\ \mathit{ArrowPack}\ (\leadsto)\ \textbf{where}\\
&\quad \mathit{packCon}\quad\ \, :: \mathit{Int} \to (\mathit{Int} \leadsto ())\\
&\quad \mathit{unpackCon} :: \mathit{Int} \to (() \leadsto \mathit{Int})\\
&\quad \texttt{--}\ \mathit{Laws}:\\
&\quad \mathit{packCon}\ n \ggg \mathit{unpackCon}\ n = \overrightarrow{id_{\mathit{Int}}}
\end{aligned}
$$

Functions *packCon* and *unpackCon* can then code every single constructor number in only as many bits as needed. For an n-constructor datatype we use just $\lceil \log_2 n \rceil$ bits to code a constructor. An interesting effect of this coding is that the constructor of any single constructor datatype will be coded using 0 bits!

We obtain slightly better results by using $\lfloor \log_2 n \rfloor$ bits for some of the constructors. Even better results are obtained if we analyze the datatype, and use Huffman coding with different probabilities for the different constructors. However, our goal is not to squeeze the last bit out of our data, but rather to show how to construct the polytypic program. The definitions of *packCon* and *unpackCon* are straightforward and omitted (they can be found in the code on the web page [13] for this paper.)

In the rest of this section $(\leadsto)$ will always stand for an arrow type constructor in the class *ArrowPack* but, as with *Regular*, we often omit the type context for brevity.

6.1 The construction of the packing function

We construct a printing function *pack*, which promotes an element level packer to a datatype level packer, together with a parsing function *unpack*, which similarly promotes an unpacker to the datatype level. If the element level arguments are inverses, then we want *unpack* to be the inverse of *pack*:

$$
\begin{aligned}
&\mathit{pack}\quad\ \, :: (a \leadsto ()) \to (d\ \ a \leadsto ())\\
&\mathit{unpack} :: (() \leadsto a) \to (() \leadsto d\ \ a)
\end{aligned}
$$

$$(6)\quad p \ggg u = \overrightarrow{i}\ \Rightarrow\ \mathit{pack}\ p \ggg \mathit{unpack}\ u = \overrightarrow{\mathit{map}\ i}$$

In the following proofs we will assume that the argument packer p and the unpacker u satisfy $p \ggg u = \overrightarrow{i}$.

Overview of the construction.

Again, the construction can be interpreted as fusing the 'printer' *pack* with the 'parser' *unpack* to get a pure arrow $\overrightarrow{map\ i}$. As we are defining polytypic functions the construction follows the structure of regular datatypes: a regular datatype is a fixed point of a pattern functor, the pattern functor is a sum of products of type terms, and the terms can involve type parameters, other types, etc.

The arrow *pack* p prints a compact representation of a value of type $d\ a$. It does this by recursing over the value, printing each constructor by computing its constructor number, and each element by using the argument printer p. The constructor number is computed by means of function $\textsc{Ps}$ ('Pack Sum'), which also takes care of passing on the recursion to the children. Function *packCon* prints the constructor number with the correct number of bits. Finally, function $\textsc{Pp}$ ('Pack Product') makes sure the information is correctly threaded through the children.

Top level recursion.

We define functions *pack* and *unpack* by explicit recursion on the top level, guided by $\textsc{Pt}$ ('Pack Top-level') and $\textsc{Ut}$ ('Unpack Top-level'). As *pack* decomposes its input value, and compactly prints the constructor and the children by means of a function $\textsc{Pt}$ (defined later), *unpack* must do the opposite: first parse the components using $\textsc{Ut}$ and then construct the top level value:

$$
\begin{aligned}
pack_d\ p &= \textsc{Pt}\ noOfCons_d\ p\ (pack_d\ p) \lll \overrightarrow{out} \\
unpack_d\ u &= \textsc{Ut}\ noOfCons_d\ u\ (unpack_d\ u) \ggg \overrightarrow{in}
\end{aligned}
$$

Here $(\lll)$ is used to reveal the symmetry of the definitions. Thus we need two new functions, $\textsc{Pt}$ and $\textsc{Ut}$, and we can already guess that we will need a corresponding fusion law:

$$
\begin{aligned}
&\textsc{Pt} :: Int \to (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto ()) \\
&\textsc{Ut} :: Int \to (() \leadsto a) \to (() \leadsto b) \to (() \leadsto f\ a\ b)
\end{aligned}
$$

$$
\begin{aligned}
(7)\quad & p \ggg u = \overrightarrow{i} \ \Rightarrow\ p' \ggg u' = \overrightarrow{i'} \ \Rightarrow \\
& \textsc{Pt}\ n\ p\ p' \ggg \textsc{Ut}\ n\ u\ u' = \overrightarrow{map2\ i\ i'}
\end{aligned}
$$

Equation (6) follows from (7) and the fixed point law (3).

Packing constructors.

We want to construct functions $\textsc{Pt}$ and $\textsc{Ut}$ such that (7) holds. Furthermore, these functions should do the actual packing and unpacking of the constructors using $packCon :: Int \to (Int \leadsto ())$ and $unpackCon :: Int \to (() \leadsto Int)$ from the *ArrowPack* class:

$$\begin{aligned} \textsc{Pt}\ n\ p\ p' &= packCon\ n \lll \textsc{Ps}\ p\ p' \\ \textsc{Ut}\ n\ u\ u' &= unpackCon\ n \ggg \textsc{Us}\ u\ u' \end{aligned}$$

The arrow $\textsc{Ps}\ p\ p'$ packs a value (using the argument packers p and p' for the parameters and the recursive structures, respectively) and returns the number of the top level constructor, by determining the position of the constructor in the pattern functor (a sum of products). The arrow *packCon* prepends the constructor number to the output. As $packCon\ n \ggg unpackCon\ n = \overrightarrow{id}$ by assumption, the requirement that function $\textsc{Pt}$ can be fused with $\textsc{Ut}$ is now passed on to $\textsc{Ps}$ and $\textsc{Us}$ ('Unpack Sum'):

$$\begin{aligned} \textsc{Ps} &:: (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto Int) \\ \textsc{Us} &:: (() \leadsto a) \to (() \leadsto b) \to (Int \leadsto f\ a\ b) \end{aligned}$$

$$(8)\quad p \ggg u = \overrightarrow{i} \;\Rightarrow\; p' \ggg u' = \overrightarrow{i'} \;\Rightarrow\; \textsc{Ps}\ p\ p' \ggg \textsc{Us}\ u\ u' = \overrightarrow{map2\ i\ i'}$$

The arrow *unpackCon* reads the constructor number and passes it on to the arrow $\textsc{Us}\ u\ u'$, which selects the desired constructor and uses its argument parsers u and u' to fill in the parameter and recursive component slots in the functor value.

Calculating constructor numbers.

The pattern functor of a Haskell datatype with n constructors is a nested sum (of products) on the outermost level. We could use a single bit in each of the nested sums to express the choice between left and right, but that would result in a unary encoding of the constructor numbers and that would be expensive for datatypes with many constructors. Instead we first calculate the constructor number, and then code that number as a bit string (using *packCon*). We define $\textsc{Ps}$ by induction over the nested sum part of the pattern functor and defer the handling of the product part to $\textsc{Pp}$ ('Pack Product'). (The definitions of in_{Int} and out_{Int} are in Figure 5.)

$$
\begin{aligned}
&\mathit{in}_{\mathit{Int}} :: \mathit{Either}\ ()\ \mathit{Int} \to \mathit{Int}\\
&\mathit{in}_{\mathit{Int}} = (\mathit{const}\ \mathit{0}) \triangledown (1+)\\[4pt]
&\mathit{out}_{\mathit{Int}} :: \mathit{Int} \qquad\ \ \to \mathit{Either}\ ()\ \mathit{Int}\\
&\mathit{out}_{\mathit{Int}}\ \ 0 \qquad\quad\ \ = \mathit{Left}\ \ \ ()\\
&\mathit{out}_{\mathit{Int}}\ \ (n+1) = \mathit{Right}\quad n
\end{aligned}
$$

Figure 5. The definitions of $\mathit{in}_{\mathit{Int}}$ and $\mathit{out}_{\mathit{Int}}$ as Haskell code.

$$
\begin{aligned}
&\textbf{polytypic}\ \textsc{Ps}_f :: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow \mathit{Int})\\
&\quad = \lambda p\ p' \to \textbf{case}\ f\ \textbf{of}\\
&\qquad\qquad\qquad g+h \longrightarrow \overrightarrow{\mathit{in}_{\mathit{Int}}} \lll (\textsc{Pp}\ p\ p' +\!\!\!+\!\!\!+ \textsc{Ps}\ p\ p')\\
&\qquad\qquad\qquad g \qquad\ \longrightarrow \overrightarrow{\lambda() \to 0} \lll \textsc{Pp}\ p\ p'\\[4pt]
&\textbf{polytypic}\ \textsc{Us}_f :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (\mathit{Int} \rightsquigarrow f\ a\ b)\\
&\quad = \lambda u\ u' \to \textbf{case}\ f\ \textbf{of}\\
&\qquad\qquad\qquad g+h \longrightarrow \overrightarrow{\mathit{out}_{\mathit{Int}}} \ggg (\textsc{Up}\ u\ u' +\!\!\!+\!\!\!+ \textsc{Us}\ u\ u')\\
&\qquad\qquad\qquad g \qquad\ \longrightarrow \overrightarrow{\lambda 0 \to ()} \ggg \textsc{Up}\ u\ u'
\end{aligned}
$$

The types for $\textsc{Pp}$ and $\textsc{Up}$ ('Unpack Product') and the corresponding fusion law are unsurprising:

$$
\begin{aligned}
&\textsc{Pp} :: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ())\\
&\textsc{Up} :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b)
\end{aligned}
$$

(9) $p \ggg u = \overrightarrow{i} \ \Rightarrow\ p' \ggg u' = \overrightarrow{i'} \ \Rightarrow\ \textsc{Pp}\ p\ p' \ggg \textsc{Up}\ u\ u' = \overrightarrow{\mathit{map2}\ i\ i'}$

The proof of Equation (8) using (9) is by induction over the nested sum structure of the functor. The proof is very similar to the corresponding proof for $\textsc{Tr}$ and therefore omitted.

Sequencing the parameters.

The last part of the construction of the program consists of the two functions $\textsc{Pp}$ and $\textsc{Up}$ defined in Figure 6. The earlier functions have calculated and printed the constructors, so what is left is "arrow plumbing". The arrow $\textsc{Pp}\ p\ p'$ traverses the top level structure of the data and inserts the correct compact printers: p at argument positions and p' at substructure positions. The only difference between $\textsc{Up}$ and $\textsc{Pp}$ is, as with *mapAr* and *mapAl* earlier, the traversal direction in the product case — visible in the use of ($\lll\!\!*$) and ($*\!\!\ggg$) respectively. The inverse proof is very similar to that for $\textsc{Tr}$ and $\textsc{Tl}$, and is omitted. The case for *Const t* is not included as no reasonable definition can be given that is polymorphic in t. Specific cases for *Const Int*, *Const Bool*,

$$
\begin{array}{l}
\textbf{polytypic}\ \textsc{Pp}_f :: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\
\quad = \lambda p\ p' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lcl}
g * h & \longrightarrow & \overrightarrow{\lambda((),()) \to ()} \lll (\textsc{Pp}_g\ p\ p' \mathbin{<\!\!*} \textsc{Pp}_h\ p\ p') \\
\mathit{Empty} & \longrightarrow & \overrightarrow{\mathit{id}} \\
\mathit{Par} & \longrightarrow & p \\
\mathit{Rec} & \longrightarrow & p' \\
d \mathbin{@} g & \longrightarrow & \mathit{pack}_d\ (\textsc{Pp}_g\ p\ p')
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic}\ \textsc{Up}_f :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \\
\quad = \lambda u\ u' \to \textbf{case}\ f\ \textbf{of} \\
\qquad\qquad\qquad \begin{array}{lcl}
g * h & \longrightarrow & \overrightarrow{\lambda() \to ((),())} \ggg (\textsc{Up}_g\ u\ u' \mathbin{*\!\!>} \textsc{Up}_h\ u\ u') \\
\mathit{Empty} & \longrightarrow & \overrightarrow{\mathit{id}} \\
\mathit{Par} & \longrightarrow & u \\
\mathit{Rec} & \longrightarrow & u' \\
d \mathbin{@} g & \longrightarrow & \mathit{unpack}_d\ (\textsc{Up}_g\ u\ u')
\end{array}
\end{array}
$$

Figure 6. The definition of Pp ('Pack Product') and Up ('Unpack Product').

and for other monomorphic types can easily be added.

7 Pretty printing

Modern programming languages allow the user to define new kinds of data. When testing or debugging a program, the user often wants to see values of these new datatypes. Many languages support the automatic derivation of printing functions for user-defined datatypes. For example, in Haskell one can write **deriving** (*Show*, *Read*) after a datatype definition, and obtain the function *show* (which prints values of the datatype) and *read* (which reads them back) for free. Thus a Haskell programmer can use (instances of) a few predefined polytypic functions, but she has no influence over their definitions nor any means of defining her own polytypic functions.

This section shows how one can *define* polytypic versions of the functions *show* and *read*. The polytypic functions *show* and *read* are each others inverses by construction. The polytypic parsing function can be used together with Haskell's functions for parsing (including proper treatment of operator fixities) but for technical reasons (involving left recursion), parsing of infix constructors is not supported.

This subsection introduces a class *ArrowReadShow* that provides the arrow operations that are used in pretty printing and parsing. The new operations are divided into four classes: *ArrowZero*, *ArrowPlus*, *ArrowSymbol* and *ArrowPrec*. The two first classes are used for error handling and are present already in Hughes' arrow paper [9], but the last two classes are new. The operations of *ArrowSymbol* are used to print and parse symbols, and the operations of *ArrowPrec* handle operator precedences.

Arrows that can fail

Up to now the data conversion programs did not have to handle failure. The unpacking algorithm would of course benefit from error handling to allow for bad input data, but no error handling is essential for expressing the algorithm. But to parse a text representation of data values we really need to choose between different parsers (for different constructors) and hence some parsers must be able to fail. Therefore we define the class *ArrowZero* for arrows that can fail:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Arrow}\ (\rightsquigarrow)\ \Rightarrow \mathit{ArrowZero}\ (\rightsquigarrow)\ \textbf{where}\\
\quad \mathit{zeroA} :: a \rightsquigarrow b\\[4pt]
\quad \text{--}\ \mathit{Laws}:\\
\quad \overrightarrow{f} \ggg \mathit{zeroA} = \mathit{zeroA} = \mathit{zeroA} \ggg \overrightarrow{f}
\end{array}
$$

The arrow *zeroA* is the multiplicative zero for composition with (at least) pure arrows and, as we will see later, the additive zero of a plus operator for arrows.

Error handling

The operator $(\Leftrightarrow)$ in the class *ArrowPlus* builds a parser that uses a second arrow if the first one fails. The operator $(\lhd\!|\!\rhd)$ is a kind of dual to the choice operator $(|||)\ ::\ (a \rightsquigarrow c) \to (b \rightsquigarrow c) \to (\mathit{Either}\ a\ b \rightsquigarrow c)$ from *ArrowChoice*. The choice operator makes a choice depending on the input, while the operator $(\lhd\!|\!\rhd)$ makes a choice depending on some hidden state and delivers the result in the corresponding summand in the output.

$$
\begin{array}{l}
\textbf{class } \mathit{ArrowZero}\ (\leadsto)\ \Rightarrow \mathit{ArrowPlus}\ (\leadsto)\ \textbf{where} \\
\quad (\lhd\!\rhd) :: (a \leadsto b) \to (a \leadsto c) \to (a \leadsto \mathit{Either}\ b\ c) \\
\quad (\Leftrightarrow) :: (a \leadsto b) \to (a \leadsto b) \to (a \leadsto b) \\[1ex]
\quad \texttt{--}\ \mathit{Defaults}: \\
\quad f \lhd\!\rhd g = (f \ggg \overrightarrow{\mathit{Left}}) \Leftrightarrow (g \ggg \overrightarrow{\mathit{Right}}) \\
\quad f \Leftrightarrow g\ = (f \lhd\!\rhd g) \ggg \overrightarrow{id \triangledown id} \\[1ex]
\quad \texttt{--}\ \mathit{Laws}: \\
\quad \mathit{zeroA} \Leftrightarrow f \qquad\ \ = f \quad = \quad f \Leftrightarrow \mathit{zeroA} \\
\quad f \lhd\!\rhd \mathit{zeroA} \qquad = f \ggg \overrightarrow{\mathit{Left}} \\
\quad \mathit{zeroA} \lhd\!\rhd f \qquad = f \ggg \overrightarrow{\mathit{Right}} \\
\quad f \ggg (g \lhd\!\rhd h) = (f \ggg g) \lhd\!\rhd (f \ggg h)
\end{array}
$$

The default definitions show that only one of $(\lhd\!\rhd)$ or $(\Leftrightarrow)$ need be defined — the relation between the *ArrowPlus* operators is the same as that between the *ArrowChoice* operators. The arrow *zeroA* is the zero of the plus operator $(\Leftrightarrow)$.

Reading and writing symbols

Almost all arrow classes thus far have been very general and useful for a wide variety of applications, but for pretty printing and parsing we need a few more specific tools. To print and parse symbols (constructor names, parentheses and spaces) we use the class *ArrowSymbol*:

$$
\begin{array}{l}
\textbf{class } \mathit{Arrow}\ (\leadsto)\ \Rightarrow \mathit{ArrowSymbol}\ (\leadsto)\ \textbf{where} \\
\quad \mathit{readSym}\ \ :: \mathit{Symbol} \to (a \leadsto a) \\
\quad \mathit{showSym} :: \mathit{Symbol} \to (a \leadsto a) \\[1ex]
\quad \texttt{--}\ \mathit{Laws}: \\
\quad \mathit{showSym}\ s \ggg \mathit{readSym}\ s\ \ = \overrightarrow{id} \\
\quad \mathit{showSym}\ s \ggg \mathit{readSym}\ s' = \mathit{zeroA} \quad \Leftarrow \quad s \neq s' \\[1ex]
\textbf{type } \mathit{Symbol} = \mathit{String}
\end{array}
$$

The two laws capture the minimal requirements needed to prove that *show* and *read* are inverses: reading a symbol is the inverse of writing the same symbol but trying to read another symbol back will fail. As examples we give one arrow for printing and one for parsing parenthesized expressions:

$$
\begin{array}{l}
\mathit{parenthesize}, \mathit{deparenthesize} :: \mathit{ArrowSymbol}\ (\leadsto) \Rightarrow (a \leadsto b) \to (a \leadsto b) \\
\mathit{parenthesize}\ f \quad\ = \mathit{showSym}\ \texttt{"("} \lll f \lll \mathit{showSym}\ \texttt{")"} \\
\mathit{deparenthesize}\ f = \mathit{readSym}\ \texttt{"("} \ggg f \ggg \mathit{readSym}\ \texttt{")"}
\end{array}
$$

Precedence levels

Finally, we define the class *ArrowPrec* to handle precedence levels and parentheses. Our formulation is inspired by the functions *showsPrec* and *readsPrec* in the Haskell classes *Show* and *Read*.

$$
\begin{array}{lcl}
\mathit{showsPrec} :: \mathit{Show}\ a \Rightarrow \mathit{Prec} \to a \to \mathit{String} & \to & \mathit{String} \\
\mathit{readsPrec} :: \mathit{Read}\ b \Rightarrow \mathit{Prec} \to \mathit{String} & \to & [(b, \mathit{String})]
\end{array}
$$

The first argument passed to *showsPrec* and *readsPrec* is the precedence level of the surrounding expression. It is used to determine whether or not the element of type a should be surrounded by parentheses.

Function *showParen* (*readParen*) is used to conditionally enclose its printer (parser) argument with parentheses. The printer *showParen* $p\ f$, encloses f with parentheses if and only if p is lower than the precedence of the environment. The printer (parser) *setPrec* $p\ f$ sets the precedence level to p in the environment of f.

$$
\begin{array}{l}
\mathbf{class}\ \mathit{ArrowSymbol}\ (\leadsto)\ \Rightarrow \mathit{ArrowPrec}\ (\leadsto)\ \mathbf{where} \\
\quad \mathit{setPrec} \quad\ \ :: \mathit{Prec} \to (a \leadsto b) \to (a \leadsto b) \\
\quad \mathit{readParen}\ :: \mathit{Prec} \to (a \leadsto b) \to (a \leadsto b) \\
\quad \mathit{showParen} :: \mathit{Prec} \to (a \leadsto b) \to (a \leadsto b) \\
\\
\quad \text{-- } \mathit{Laws}: \\
\quad x \ggg y = \overrightarrow{z} \Rightarrow \mathit{setPrec}\ p\ x \ggg \mathit{setPrec}\ p\ y = \overrightarrow{z} \\
\quad x \ggg y = \overrightarrow{z} \Rightarrow \mathit{showParen}\ p\ (\mathit{showSym}\ n \lll x) \ggg \\
\qquad\qquad\qquad\quad \mathit{readParen}\ p\ (\mathit{readSym}\ n \ggg y) = \overrightarrow{z} \\
\quad n \neq n' \qquad\ \ \Rightarrow \mathit{showParen}\ p\ (\mathit{showSym}\ n \lll x) \ggg \\
\qquad\qquad\qquad\quad \mathit{readParen}\ p'\ (\mathit{readSym}\ n' \ggg y) = \mathit{zeroA}
\end{array}
$$

Show and read

The polytypic functions *show* and *read* use operations from *ArrowChoice* and from all of the four classes just defined, and to capture this succinctly in the types, we define the class synonym *ArrowShowRead*:

$$
\begin{array}{l}
\mathbf{class}\ (\mathit{ArrowChoice}\ (\leadsto), \mathit{ArrowPlus}\ (\leadsto), \mathit{ArrowPrec}\ (\leadsto)) \\
\quad \Rightarrow \mathit{ArrowShowRead}\ (\leadsto)
\end{array}
$$

For the rest of this section, all occurrences of $(\leadsto)$ will denote an arrow in the class *ArrowShowRead*.

In this section we define a polytypic show function *show* and a polytypic read function *read* and we prove that *read* is the inverse of *show*:

$$\begin{aligned} show &:: (a \rightsquigarrow ()) \rightarrow (d\ a \rightsquigarrow ()) \\ read &:: (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow d\ a) \end{aligned}$$

(10) $s \ggg r = \overrightarrow{i} \;\Rightarrow\; show\ s \ggg read\ r = \overrightarrow{map\ i}$

Note that we embed the precedence level handling in the arrow type, so that each time s or r is used, the correct precedence level is determined by the local environment. Compared to the Haskell functions *showsPrec* and *readsPrec* for a specific datatype we make three changes:

- we generalize the types by using arrows,
- we parameterize by the element level show (read) operation,
- and we define polytypic versions that work for all types $d\ a$.

The definition is divided into four levels, following the structure of datatype definitions: the top level (*show* and *read*) is a recursive definition, the second level (Ss and Rs) breaks down the sum structure of the pattern functor, the third level (Sp and Rp) analyzes the product structure and finally the fourth level (Sr and Rr) deals with parameters and uses of other datatypes.

The top level calculates the list of constructors of the datatype and passes them down to the next level. The second level shows (reads) the constructor name and handles parentheses (depending on the precedence levels of the expressions). The third level inserts spaces between the arguments of the constructors and marks the arguments as being subexpressions (potentially needing parentheses). Finally the bottom level just applies the appropriate show (read) functions passed down as parameters or calls *show* (*read*) for occurrences of other datatypes.

Top level recursion

The built-in polytypic value $constructors_d :: [Constructor]$ (introduced in Section 2.2) is the list of constructors of the datatype $d\ a$. In the following proofs we use two properties of the constructor list: the list has at least one element (there are no 0-constructor datatypes in Haskell) and all the constructor names are distinct.

Function *show* uses *out* to expose the top level structure of the datatype value and handles the recursion by passing itself as an argument to Ss ('Show Sum', defined later). Similarly, *read* calls RS ('Read Sum') and converts the result to a datatype value using *in*.

$$\begin{aligned} show_d\ s &= \mathrm{Ss}_{\Phi_d}\ constructors_d\ s\ (show_d\ s) \lll \overrightarrow{out_d} \\ read_d\ r &= \mathrm{Rs}_{\Phi_d}\ constructors_d\ r\ (read_d\ r) \ggg \overrightarrow{in_d} \end{aligned}$$

The two helper functions Ss and Rs have their own inverse law:

$$\begin{aligned} &\mathrm{Ss}_f :: [Constructor] \to (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\ &\mathrm{Rs}_f :: [Constructor] \to (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \end{aligned}$$

$$(11)\ \begin{aligned} &s \ggg r = \overrightarrow{i} \Rightarrow s' \ggg r' = \overrightarrow{i'} \Rightarrow \\ &\mathrm{Ss}_f\ cs\ s\ s' \ggg \mathrm{Rs}_f\ cs\ r\ r' = \overrightarrow{map2_f\ i\ i'} \end{aligned}$$

Equation (10) now follows from Equation (11) by fixed point induction (3).

Printing constructors

On the top level, every pattern functor is a right associative sum, and this is mirrored in the definitions of Ss and Rs as well as in the corresponding part of the proof. We use $name :: Constructor \to String$ to access the constructor names and we check the precedence level with $prec :: Constructor \to Prec$ to determine when parentheses are needed.

$$\begin{aligned} &\textbf{polytypic}\ \mathrm{Ss}_f :: [Constructor] \to (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\ &\quad = \lambda(c : cs)\ s\ s' \to \textbf{case}\ f\ \textbf{of} \\ &\qquad\qquad g + h \longrightarrow \mathrm{Ss}_g\ [c]\ s\ s' \mid\mid\mid \mathrm{Ss}_h\ cs\ s\ s' \\ &\qquad\qquad g \longrightarrow showParen\ (prec\ c) \\ &\qquad\qquad\qquad (showSym\ (name\ c) \lll \mathrm{Sp}_g\ s\ s') \end{aligned}$$

$$\begin{aligned} &\textbf{polytypic}\ \mathrm{Rs}_f :: [Constructor] \to (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \\ &\quad = \lambda(c : cs)\ r\ r' \to \textbf{case}\ f\ \textbf{of} \\ &\qquad\qquad g + h \longrightarrow \mathrm{Rs}_g\ [c]\ r\ r' <\!\!|\!\!> \mathrm{Rs}_h\ cs\ r\ r' \\ &\qquad\qquad g \longrightarrow readParen\ (prec\ c) \\ &\qquad\qquad\qquad (readSym\ (name\ c) \ggg \mathrm{Rp}_g\ r\ r') \end{aligned}$$

Functions Sp (for 'Show Product') and Rp (for 'Read Product') have the following properties:

$$\begin{aligned} &\mathrm{Sp}_f :: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\ &\mathrm{Rp}_f :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \end{aligned}$$

$$(12)\ s \ggg r = \overrightarrow{i} \Rightarrow s' \ggg r' = \overrightarrow{i'} \Rightarrow \mathrm{Sp}_f\ s\ s' \ggg \mathrm{Rp}_f\ r\ r' = \overrightarrow{map2_f\ i\ i'}$$

We prove (11) by induction over the nested sum part of the pattern functor. We strengthen the induction hypothesis to include also the following law. For all p, x and y, and for all $c' \notin cs'$:

$$\begin{array}{ll}
(13)\ \mathrm{Rs}_f\ cs'\ r\ r' \lll \mathit{showParen}\ p\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) & = \mathit{zeroA} \\
(14)\ \mathrm{Ss}_f\ cs'\ s\ s' \ggg \mathit{readParen}\ p\ (\mathit{showSym}\ (\mathit{name}\ c') \ggg y) & = \mathit{zeroA}
\end{array}$$

We assume $s \ggg r = \overrightarrow{i}$ and $s' \ggg r' = \overrightarrow{i'}$ and calculate as follows for Equation (11):

The sum case, $g + h$:
We prove the three equations separately, starting with (11):

$$\begin{array}{ll}
& \mathrm{Ss}_{g+h}\ (c : cs)\ s\ s' \ggg \mathrm{Rs}_{g+h}\ (c : cs)\ r\ r' \\
= & \quad \{\text{Definitions}\} \\
& (\mathrm{Ss}_g\ [c]\ s\ s' \mathbin{|||} \mathrm{Ss}_h\ cs\ s\ s') \ggg (\mathrm{Rs}_g\ [c]\ r\ r' \mathbin{<\!\!|\!\!>} \mathrm{Rs}_h\ cs\ r\ r') \\
= & \quad \{\text{Distribution laws for } (|||) \text{ and } (<\!\!|\!\!>)\} \\
& ((\mathrm{Ss}_g\ [c]\ s\ s' \ggg \mathrm{Rs}_g\ [c]\ r\ r') \mathbin{<\!\!|\!\!>} (\mathrm{Ss}_g\ [c]\ s\ s' \ggg \mathrm{Rs}_h\ cs\ r\ r')) \mathbin{|||} \\
& ((\mathrm{Ss}_h\ cs\ s\ s' \ggg \mathrm{Rs}_g\ [c]\ r\ r') \mathbin{<\!\!|\!\!>} (\mathrm{Ss}_h\ cs\ s\ s' \ggg \mathrm{Rs}_h\ cs\ r\ r'))
\end{array}$$

The first term is identical to the term in the default-case below. Use induction hypothesis (13) and (14) for the second and third terms, and induction hypothesis (11) for the fourth term.

$$\begin{array}{ll}
& (\overrightarrow{\mathit{map2}_g\ i\ i'} \mathbin{<\!\!|\!\!>} \mathit{zeroA}) \mathbin{|||} (\mathit{zeroA} \mathbin{<\!\!|\!\!>} \overrightarrow{\mathit{map2}_h\ i\ i'}) \\
= & \quad \{\text{Laws for } \mathit{zeroA} \text{ and } (<\!\!|\!\!>)\} \\
& (\overrightarrow{\mathit{map2}_g\ i\ i'} \ggg \overrightarrow{\mathit{Left}}) \mathbin{|||} (\overrightarrow{\mathit{map2}_h\ i\ i'} \ggg \overrightarrow{\mathit{Right}}) \\
= & \quad \{\text{Relation between } (|||) \text{ and } (+\!\!+\!\!+)\} \\
& \overrightarrow{\mathit{map2}_g\ i\ i'} \mathbin{+\!\!+\!\!+} \overrightarrow{\mathit{map2}_h\ i\ i'} \\
= & \quad \{(+\!\!+\!\!+) \text{ preserves } \overrightarrow{\cdot}\} \\
& \overrightarrow{\mathit{map2}_g\ i\ i' + \mathit{map2}_h\ i\ i'} \\
= & \quad \{\text{Definition of } \mathit{map2}_{g+h}\} \\
& \overrightarrow{\mathit{map2}_{g+h}\ i\ i'}
\end{array}$$

Now we turn to (13):

$$\begin{array}{ll}
& \mathit{showParen}\ p\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) \ggg \mathrm{Rs}_{g+h}\ cs'\ r\ r' \\
= & \quad \{\text{Definition of } \mathrm{Rs}_{g+h}, \text{ let } (c : cs) = cs'\} \\
& \mathit{showParen}\ p\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) \ggg \\
& (\mathrm{Rs}_g\ [c]\ r\ r' \mathbin{<\!\!|\!\!>} \mathrm{Rs}_h\ cs\ r\ r') \\
= & \quad \{\text{Distribution law for } (<\!\!|\!\!>)\}
\end{array}$$

$$
\begin{aligned}
& (\mathit{showParen}\ p\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) \ggg \mathrm{Rs}_g\ [c]\ r\ r') \mathbin{<\!\!|\!\!>} \\
& (\mathit{showParen}\ p\ (\mathit{showSym}\ (\mathit{name}\ c') \lll x) \ggg \mathrm{Rs}_h\ \mathit{cs}\ r\ r') \\
= & \qquad \{\text{The second law of } \mathit{showParen} \text{ and the induction hypothesis}\} \\
& \mathit{zeroA} \mathbin{<\!\!|\!\!>} \mathit{zeroA} \\
= & \qquad \{\text{Laws for } (\mathbin{<\!\!|\!\!>}) \text{ and } \mathit{zeroA}\} \\
& \mathit{zeroA}
\end{aligned}
$$

The proof of (14) is very similar and omitted.

The default case, g:
As the constructor list has the same number of elements as the number of sub-functors in the sum structure of the functor, there will be only one element left in the constructor list in the base case. Thus we can match on $[c]$ instead of $(c : cs)$.

$$
\begin{aligned}
& \mathrm{Ss}_g\ [c]\ s\ s' \ggg \mathrm{Rs}_g\ [c]\ r\ r' \\
= & \qquad \{\text{Definitions}\} \\
& \mathit{showParen}\ (\mathit{prec}\ c)\ (\mathit{showSym}\ (\mathit{name}\ c) \lll \mathrm{Sp}_g\ s\ s') \ggg \\
& \mathit{readParen}\ (\mathit{prec}\ c)\ (\mathit{readSym}\ (\mathit{name}\ c) \ggg \mathrm{Rp}_g\ r\ r') \\
= & \qquad \left\{\begin{array}{l}\text{With } p = \mathit{prec}\ c,\ n = \mathit{name}\ c,\ x = \mathrm{Sp}_g\ s\ s' \text{ and } y = \\ \mathrm{Rp}_g\ r\ r' \text{ we can apply the first law for } \mathit{showParen} \text{ as} \\ x \ggg y = \overrightarrow{\mathit{map2}_g\ i\ i'} \text{ is exactly the ind. hyp. (12).}\end{array}\right. \\
& \overrightarrow{\mathit{map2}_g\ i\ i'}
\end{aligned}
$$

Both (13) and (14) follow immediately from the second law of *showParen*.

Printing constructor arguments

The function Sp (Rp) inserts (reads) a space before each argument of a constructor, and sets the precedence level of each argument to $\mathit{high} = \mathit{10}$ (to force parentheses except for atomic subexpressions).

$$
\begin{aligned}
& \textbf{polytypic}\ \mathrm{Sp}_f :: (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto ()) \\
& \quad = \lambda s\ s' \to \textbf{case}\ f\ \textbf{of} \\
& \qquad\qquad g * h \quad \longrightarrow \overrightarrow{\lambda((),()) \to ()} \lll (\mathrm{Sp}_g\ s\ s' \mathbin{<\!\!*\!\!*} \mathrm{Sp}_h\ s\ s') \\
& \qquad\qquad \mathit{Empty} \longrightarrow \overrightarrow{\lambda() \to ()} \\
& \qquad\qquad g \qquad\ \ \longrightarrow \mathit{showSym}\ \texttt{" "} \lll \mathit{setPrec}\ \mathit{high}\ (\mathrm{Sr}_g\ s\ s')
\end{aligned}
$$

$$
\begin{aligned}
& \textbf{polytypic}\ \mathrm{Rp}_f :: (() \leadsto a) \to (() \leadsto b) \to (() \leadsto f\ a\ b) \\
& \quad = \lambda r\ r' \to \textbf{case}\ f\ \textbf{of} \\
& \qquad\qquad g * h \quad \longrightarrow \overrightarrow{\lambda() \to ((),())} \ggg (\mathrm{Rp}_g\ r\ r' \mathbin{*\!\!*\!\!>} \mathrm{Rp}_h\ r\ r') \\
& \qquad\qquad \mathit{Empty} \longrightarrow \overrightarrow{\lambda() \to ()} \\
& \qquad\qquad g \qquad\ \ \longrightarrow \mathit{readSym}\ \texttt{" "} \ggg \mathit{setPrec}\ \mathit{high}\ (\mathrm{Rr}_g\ r\ r')
\end{aligned}
$$

Here functions S_R (for 'Show Rest') and R_R (for 'Read Rest') have the following properties:

$$
\begin{aligned}
\text{SR}_f &:: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f \;\, a \;\, b \rightsquigarrow ()) \\
\text{RR}_f &:: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f \;\, a \;\, b)
\end{aligned}
$$

(15) $s \ggg r = \overrightarrow{i} \Rightarrow s' \ggg r' = \overrightarrow{i'} \Rightarrow \text{SR}_f \;\, s \;\, s' \ggg \text{RR}_f \;\, r \;\, r' = \overrightarrow{map2_f \;\, i \;\, i'}$

We prove (12) by induction over the product structure of the functor f:

The product case, $g * h$**:**

$$
\begin{aligned}
& \text{SP}_{g*h} \;\, s \;\, s' \ggg \text{RP}_{g*h} \;\, r \;\, r' \\
= \quad & \{\text{Definitions}\} \\
& (\text{SP}_g \;\, s \;\, s' \lll\!* \; \text{SP}_h \;\, s \;\, s') \ggg \overrightarrow{\lambda((),()) \to ()} \ggg \\
& \overrightarrow{\lambda() \to ((),())} \ggg (\text{RP}_g \;\, r \;\, r' *\!\!\ggg \text{RP}_h \;\, r \;\, r') \\
= \quad & \{(\lambda((),()) \to ()) \; ; (\lambda() \to ((),())) = id_{((),())}\} \\
& (\text{SP}_g \;\, s \;\, s' \lll\!* \; \text{SP}_h \;\, s \;\, s') \ggg (\text{RP}_g \;\, r \;\, r' *\!\!\ggg \text{RP}_h \;\, r \;\, r') \\
= \quad & \{\text{Inverse law for } (\lll\!*), \text{ induction hypothesis (twice)}\} \\
& \overrightarrow{map2_g \;\, i \;\, i' \twoheadleftarrow\!\!* \; map2_h \;\, i \;\, i'} \\
= \quad & \{\text{Definition of } map2_{g*h}\} \\
& \overrightarrow{map2_{g*h} \;\, i \;\, i'}
\end{aligned}
$$

The empty case, *Empty***:**
Trivial.

The base case, g**:**

$$
\begin{aligned}
& \text{SP}_g \;\, s \;\, s' \ggg \text{RP}_g \;\, r \;\, r' \\
= \quad & \{\text{Definitions}\} \\
& setPrec \;\, high \;\, (\text{SR}_g \;\, s \;\, s') \ggg showSym \;\, \texttt{" "} \ggg \\
& readSym \;\, \texttt{" "} \ggg setPrec \;\, high \;\, (\text{RR}_g \;\, r \;\, r') \\
= \quad & \{\text{Law for } showSym \text{ and } readSym\} \\
& setPrec \;\, high \;\, (\text{SR}_g \;\, s \;\, s') \ggg setPrec \;\, high \;\, (\text{RR}_g \;\, r \;\, r') \\
= \quad & \{\text{Law for } setPrec \text{ and Equation (15)}\} \\
& \overrightarrow{map2_g \;\, i \;\, i'}
\end{aligned}
$$

Printing the rest

At the bottom level all that is left is to apply the correct printer (parser): *Par* and *Rec* select from the parameters and $d@g$ calls the top level *show* (*read*) recursively.

$$
\begin{array}{l}
\textbf{polytypic } \mathrm{SR}_f :: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\
\quad = \lambda s\ s' \to \textbf{case } f \textbf{ of} \\
\qquad\qquad\qquad \begin{array}{lll}
Par & \longrightarrow & s \\
Rec & \longrightarrow & s' \\
d\ @\ g & \longrightarrow & show_d\ (\mathrm{SR}_g\ s\ s')
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic } \mathrm{RR}_f :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \\
\quad = \lambda r\ r' \to \textbf{case } f \textbf{ of} \\
\qquad\qquad\qquad \begin{array}{lll}
Par & \longrightarrow & r \\
Rec & \longrightarrow & r' \\
d\ @\ g & \longrightarrow & read_d\ (\mathrm{RR}_g\ r\ r')
\end{array}
\end{array}
$$

As for packing, the case for *Const t* is not included, but specific cases for monomorphic types can easily be added.

The only remaining proof obligation is (15) and the proof is once again by induction on the structure of the functor — the *Par* and *Rec* cases follow immediately from the assumptions, and the $d\ @\ g$ case from the top level induction hypothesis (10) and the local induction hypothesis (15). This completes the proof that *read* is the inverse of *show*.

8 Generating arrow instances

Most of the code presented in this paper is generic in two ways. We use polytypism to parametrize our definitions by a regular datatype, and we use Haskell's constructor classes to parametrize by the choice of concrete arrow implementation. Using PolyP, we obtain specific instances of the polytypic functions automatically, but we do have to write instances for the arrow classes. This section describes a few general arrow constructors and shows how to combine them to obtain an example instance for *ArrowShowRead* that satisfies the necessary laws.

8.1 Kleisli arrows

We have already presented three arrow instances: the trivial function arrow $a \to b$, the state arrow $a \rightsquigarrow_s b$ for any state type s and the Kleisli arrows *Kleisli m a b* for every monad m. The Kleisli arrows were defined in Section 4 together with instances for *Arrow* and *ArrowChoice*. If the underlying monad has a zero and a plus operation (is an instance of the Haskell class *MonadPlus*), then we can define instances for *ArrowZero* and *ArrowPlus* as well:

$$\textbf{data}\ \mathit{Kleisli}\ m\ a\ b = \mathit{Kleisli}\ (a \to m\ b)$$

$$\begin{array}{l}\textbf{instance}\ \mathit{MonadPlus}\ m \Rightarrow \mathit{ArrowZero}\ (\mathit{Kleisli}\ m)\ \textbf{where}\\ \quad \mathit{zeroA} = \mathit{Kleisli}\ (\mathit{const}\ \mathit{mzero})\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathit{MonadPlus}\ m \Rightarrow \mathit{ArrowPlus}\ (\mathit{Kleisli}\ m)\ \textbf{where}\\ \quad \mathit{Kleisli}\ f \mathbin{<\!+\!>} \mathit{Kleisli}\ g = \mathit{Kleisli}\ (\lambda x \to \mathit{mplus}\ (f\ x)\ (g\ x))\end{array}$$

The *Maybe* monad and the *Bag* monad are in *MonadPlus*, but the the list monad is not (with the standard Haskell definition) as the order of the alternative parse results depends on how the parser is expressed. To allow for multiple parse results, we therefore use bags instead of lists.

8.2 State arrow transformers

The state arrow can be generalized to a state arrow *transformer* that adds state passing to any other arrow. We follow Hughes' [9] terminology and call arrow transformers *functors* for short.

$$\textbf{data}\ \mathit{StateFunctor}\ s\ (\leadsto)\ a\ b = \mathit{SF}\ ((a, s) \leadsto (b, s))$$

With this definition the simple state arrow $SA\ s$ is equivalent to adding state passing to the trivial arrow: $\mathit{StateFunctor}\ s\ (\to)$. The state functor instances for *Arrow*, *ArrowChoice*, *ArrowZero* and *ArrowPlus* are in Figure 7. The laws included in the definition of the different arrow classes are satisfied for these instances.

8.3 Monad arrow transformers

All the arrow type constructors defined so far were defined also in Hughes' arrow paper [9], but the following construction is new. The monad arrow constructor *MonadFunctor* wraps a monad around the arrow type.

$$\textbf{data}\ \mathit{MonadFunctor}\ m\ (\leadsto)\ a\ b = \mathit{MF}\ (m\ (a \leadsto b))$$

For every monad m and every arrow $(\leadsto)$ this construction gives us a new arrow $\mathit{MonadFunctor}\ m\ (\leadsto)$, but for this new arrow to support choice, failure and error handling we need to restrict the monad to, essentially, a reader monad. The reader arrow transformer *ReaderFunctor* is a special case of the monad

$$
\begin{aligned}
&\textbf{instance } \mathit{Arrow}\ (\rightsquigarrow) \Rightarrow \mathit{Arrow}\ (\mathit{StateFunctor}\ s\ (\rightsquigarrow))\ \textbf{where}\\
&\quad \overrightarrow{f} \qquad\qquad\quad = \mathit{SF}\ (\overrightarrow{\mathit{first}\ f})\\
&\quad \mathit{SF}\ f \ggg \mathit{SF}\ g = \mathit{SF}\ (f \ggg g)\\
&\quad \mathit{first}\ (\mathit{SF}\ f) \quad = \mathit{SF}\ (\overrightarrow{\mathit{swap23}} \ggg \mathit{first}\ f \ggg \overrightarrow{\mathit{swap23}})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{swap23} :: ((a, b), s) \to ((a, s), b)\\
&\mathit{swap23} \quad ((a, b), s) \ = \ ((a, s), b)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{ArrowChoice}\ (\rightsquigarrow) \Rightarrow \mathit{ArrowChoice}\ (\mathit{StateFunctor}\ s\ (\rightsquigarrow))\ \textbf{where}\\
&\quad \mathit{SF}\ f \mid\mid\mid \mathit{SF}\ g = \mathit{SF}\ (\overrightarrow{\mathit{eitherout}} \ggg (f \mid\mid\mid g))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eitherout} \qquad\quad :: (\mathit{Either}\ a\ a', s) \to \mathit{Either}\ (a, s)\ (a', s)\\
&\mathit{eitherout}\ (x, s) = (\mathit{pairs} + \mathit{pairs})\ x\ \textbf{where}\ \mathit{pairs}\ a = (a, s)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{ArrowZero}\ (\rightsquigarrow) \Rightarrow \mathit{ArrowZero}\ (\mathit{StateFunctor}\ s\ (\rightsquigarrow))\ \textbf{where}\\
&\quad \mathit{zeroA} = \mathit{SF}\ \mathit{zeroA}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } \mathit{ArrowPlus}\ (\rightsquigarrow) \Rightarrow \mathit{ArrowPlus}\ (\mathit{StateFunctor}\ s\ (\rightsquigarrow))\ \textbf{where}\\
&\quad \mathit{SF}\ f \oplus \mathit{SF}\ g = \mathit{SF}\ (f \oplus g)
\end{aligned}
$$

Figure 7. Instance declarations for the state arrow transformer.

arrow transformer:

$$\textbf{type } \mathit{ReaderFunctor}\ r = \mathit{MonadFunctor}\ ((\to)\ r)$$

The transformer *ReaderFunctor* r adds an environment of type r to any arrow. This can also be simulated with *StateFunctor* but when no update is needed, *ReaderFunctor* is more efficient and also simplifies the proofs. We use the shorthand notation $a \overset{r}{\rightsquigarrow} b$ for *ReaderFunctor* $r\ (\rightsquigarrow)\ a\ b$. (Note the difference between the notation $a \rightsquigarrow_s b$ for the state arrow type and $a \overset{r}{\rightsquigarrow} b$ for the reader arrow type.) The instances for *MonadFunctor* and *ReaderFunctor* are in Figure 8.

Two useful operations on *ReaderFunctor* arrows are *getEnv* and *comapEnv*:

$$
\begin{aligned}
&\mathit{getEnv}\ :: \ \mathit{Arrow}\ (\rightsquigarrow) \Rightarrow a \overset{r}{\rightsquigarrow} r\\
&\mathit{getEnv} = \mathit{MF}\ (\lambda \mathit{env} \to \overrightarrow{\lambda a \to \mathit{env}})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{comapEnv} :: (s \to r) \to \qquad (a \overset{r}{\rightsquigarrow} b) \qquad \to \qquad (a \overset{s}{\rightsquigarrow} b)\\
&\mathit{comapEnv} \qquad f \qquad\quad (\mathit{MF}\ \mathit{envreader}) \ = \ \mathit{MF}\ (f\ ;\ \mathit{envreader})
\end{aligned}
$$

The arrow *getEnv* ignores its input and returns the value of the environment. The arrow *comapEnv* $f\ q$ adapts q to a different environment by transforming the environment value by f.

$$
\begin{array}{l}
\textbf{instance } (\mathit{Monad}\ m, \mathit{Arrow}\ (\leadsto)) \Rightarrow \mathit{Arrow}\ (\mathit{MonadFunctor}\ m\ (\leadsto))\ \textbf{where} \\
\quad \overrightarrow{f} \qquad\qquad\qquad = \mathit{MF}\ (\mathit{liftM0}\ \overrightarrow{f}) \\
\quad \mathit{MF}\ f \ggg \mathit{MF}\ g = \mathit{MF}\ (\mathit{liftM2}\ (\ggg)\ f\ g) \\
\quad \mathit{first}\ (\mathit{MF}\ f) \quad\ \ = \mathit{MF}\ (\mathit{liftM}\ \mathit{first}\ f)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{ArrowChoice}\ (\leadsto) \Rightarrow \mathit{ArrowChoice}\ (\mathit{ReaderFunctor}\ r\ (\leadsto))\ \textbf{where} \\
\quad \mathit{MF}\ f \mathbin{|||} \mathit{MF}\ g = \mathit{MF}\ (\mathit{liftM2}\ (|||)\ f\ g)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{ArrowZero}\ (\leadsto) \Rightarrow \mathit{ArrowZero}\ (\mathit{ReaderFunctor}\ r\ (\leadsto))\ \textbf{where} \\
\quad \mathit{zeroA} = \mathit{MF}\ (\mathit{liftM0}\ \mathit{zeroA})
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{ArrowPlus}\ (\leadsto) \Rightarrow \mathit{ArrowPlus}\ (\mathit{ReaderFunctor}\ r\ (\leadsto))\ \textbf{where} \\
\quad \mathit{MF}\ f \mathbin{<\!|\!>} \mathit{MF}\ g = \mathit{MF}\ (\mathit{liftM2}\ (<\!|\!>)\ f\ g) \\
\quad \mathit{MF}\ f \mathbin{<\!\!+\!\!>} \mathit{MF}\ g = \mathit{MF}\ (\mathit{liftM2}\ (<\!\!+\!\!>)\ f\ g)
\end{array}
$$

$$
\begin{array}{ll}
\mathit{liftM0} & :: \mathit{Monad}\ m \Rightarrow a \to m\ a \\
\mathit{liftM0}\ f & = \mathit{return}\ f \\
\mathit{liftM} & :: \mathit{Monad}\ m \Rightarrow (a \to b) \to (m\ a \to m\ b) \\
\mathit{liftM}\ f\ m & = m \ggg\!= \lambda x \to \mathit{return}\ (f\ x) \\
\mathit{liftM2} & :: \mathit{Monad}\ m \Rightarrow (a \to b \to c) \to (m\ a \to m\ b \to m\ c) \\
\mathit{liftM2}\ f\ m\ n & = m \ggg\!= \lambda x \to n \ggg\!= \lambda y \to \mathit{return}\ (f\ x\ y)
\end{array}
$$

Figure 8. Instance declarations for *MonadFunctor* and *ReaderFunctor*.

8.4 An instance for ArrowShowRead

We can combine the three general arrow constructors to obtain an arrow RS that can be made an instance of *ArrowShowRead*:

$$
\begin{array}{l}
\textbf{type } \mathit{RS} = \mathit{ReaderFunctor}\ \mathit{Prec}\ (\mathit{StateFunctor}\ \mathit{Tokens}\ (\mathit{Kleisli}\ \mathit{Bag})) \\
\textbf{type } \mathit{Tokens} = [\mathit{String}]
\end{array}
$$

The transformer *ReaderFunctor Prec* adds an environment containing an integer to handle the precedence level, the transformer *StateFunctor Tokens* adds a state containing a token list and the inner arrow *Kleisli Bag* handles the bag of alternative parses. By unfolding the definitions of the arrow constructors we get

$$\mathit{RS}\ a\ b \cong \mathit{Prec} \to (a, \mathit{Tokens}) \to \mathit{Bag}\ (b, \mathit{Tokens})\ .$$

This can be compared with the types for *showsPrec* and *readsPrec* from the Haskell prelude.

$$
\begin{array}{lll}
\mathit{showsPrec} :: \mathit{Show}\ a \Rightarrow \mathit{Prec} \to a \to \mathit{String} & \to & \mathit{String} \\
\mathit{readsPrec} :: \mathit{Read}\ b \Rightarrow \mathit{Prec} \to \mathit{String} & \to & [(b, \mathit{String})]
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{ArrowSymbol}\ \mathit{RS}\ \textbf{where} \\
\quad \mathit{showSym}\ s = \mathit{MF}\ (\mathit{return}\ (\mathit{SF}\ (\mathit{second}\ \overrightarrow{(s\,:)}))) \\
\quad \mathit{readSym}\ s\ \ = \mathit{MF}\ (\mathit{return}\ (\mathit{SF}\ (\mathit{second}\ (\mathit{Kleisli}\ (\mathit{readToken}\ s)))))
\end{array}
$$

$$
\begin{array}{ll}
\mathit{readToken}\ t\ (s : ss) \mid s == t & = \mathit{return}\ ss \\
\mathit{readToken}\ t\ \_ & = \mathit{mzero}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{ArrowPrec}\ \mathit{RS}\ \textbf{where} \\
\quad \mathit{setPrec}\ p \qquad\ \ = \mathit{comapEnv}\ (\mathit{const}\ p) \\
\quad \mathit{showParen}\ p\ f = \mathit{ifHighPrec}\ p\ (\mathit{parenthesize}\ f)\ f \\
\quad \mathit{readParen}\ p\ f\ \ = \mathit{ifHighPrec}\ p\ (\mathit{deparenthesize}\ f)\ f
\end{array}
$$

$$
\begin{array}{l}
\mathit{ifHighPrec} :: \mathit{ArrowChoice}\ (\leadsto) \Rightarrow \mathit{Prec} \to (a \overset{\mathit{Prec}}{\leadsto} b) \to (a \overset{\mathit{Prec}}{\leadsto} b) \to (a \overset{\mathit{Prec}}{\leadsto} b) \\
\mathit{ifHighPrec}\ p = \mathit{ifA}\ (\mathit{getEnv} \ggg \overrightarrow{(> p)})
\end{array}
$$

Figure 9. Instances for *ArrowSymbol* and *ArrowPrec*.

These types use *String* where *RS* uses *Tokens*, and lists instead of *Bag*s but are otherwise very similar to $RS\ a\ ()$ and $RS\ ()\ b$, respectively.

The arrow *RS* is by construction an arrow with choice, zero and plus. Hence, all we need to make *RS* an instance of *ArrowShowRead* are the instances for *ArrowSymbol* and *ArrowPrec* in Figure 9. Function *readSym* is the standard item parser and *showSym* is even simpler (both ignore the precedence). The functions *setPrec*, *showParen* and *readParen* use the precedence level environment to determine when to read or write parentheses (using *deparenthesize* and *parenthesize*). The proofs that the instances satisfy the laws of the classes are long but relatively simple.

9 Results and conclusions

Overview of the results

We have defined the following pairs of data conversion programs and related them with inverse laws:

- Shape plus content: (Section 3)

$$
\begin{array}{l}
\mathit{separate} :: d\ a \leadsto_{[a]} d\ () \\
\mathit{combine} :: d\ () \leadsto_{[a]} d\ a \\
\mathit{separate} \ggg \mathit{combine} = \overrightarrow{\mathit{id}}
\end{array}
$$

- Arrow maps: (Section 5)

$$\begin{array}{l} \mathit{mapAr} :: \mathit{ArrowChoice}\ (\leadsto)\ \Rightarrow\ (a \leadsto b) \to (d\ \ a \leadsto d\ \ b) \\ \mathit{mapAl} :: \mathit{ArrowChoice}\ (\leadsto)\ \Rightarrow\ (a \leadsto b) \to (d\ \ a \leadsto d\ \ b) \\ p \ggg u = \overrightarrow{i}\ \ \Rightarrow\ \ \mathit{mapAr}\ p \ggg \mathit{mapAl}\ u = \overrightarrow{\mathit{map}\ i} \end{array}$$

- Packing: (Section 6)

$$\begin{array}{l} \mathit{pack}\ \ \ \ :: \mathit{ArrowPack}\ (\leadsto)\ \Rightarrow\ (a \leadsto ()) \to (d\ \ a \leadsto ()) \\ \mathit{unpack} :: \mathit{ArrowPack}\ (\leadsto)\ \Rightarrow\ (() \leadsto a) \to (() \leadsto d\ \ a) \\ p \ggg u = \overrightarrow{i}\ \ \Rightarrow\ \ \mathit{pack}\ p \ggg \mathit{unpack}\ u = \overrightarrow{\mathit{map}\ i} \end{array}$$

- Pretty printing: (Section 7)

$$\begin{array}{l} \mathit{show} :: \mathit{ArrowShowRead}\ (\leadsto)\ \Rightarrow\ (a \leadsto ()) \to (d\ \ a \leadsto ()) \\ \mathit{read}\ \ :: \mathit{ArrowShowRead}\ (\leadsto)\ \Rightarrow\ (() \leadsto a) \to (() \leadsto d\ \ a) \\ s \ggg r = \overrightarrow{i}\ \ \Rightarrow\ \ \mathit{show}\ s \ggg \mathit{read}\ r = \overrightarrow{\mathit{map}\ i} \end{array}$$

We can combine the last two applications to obtain compression and decompression. The composition of the polytypic read function *read* with the packing function *pack* gives a structured compression algorithm *compress* that takes a plain text representation of a datatype value to a bit stream. The corresponding decompression algorithm *decompress* is a composition of the unpacking function *unpack* and the polytypic show function *show*. Function *decompress* is the inverse of *compress* for all strings that represent a value. This fact follows from the inverse laws for pretty printing and packing.

Conclusions

We have constructed polytypic programs for several data conversion problems. As far as we are aware, these are the first implemented generic descriptions of programs for data conversion problems. Recent work by Hinze [6] also contains a polytypic show function and a simple packing function, but his language still lacks an implementation.

For each of the data conversion problems considered in this paper we construct a pair of functions. These pairs of functions are inverse functions by construction. Since we started applying the inverse function requirement rigorously in the construction of the programs, the size and the complexity of the code have been reduced considerably. Compare for example Björk's [3] and Huisman's [10] definitions, with the polytypic read and show functions defined in this paper. We firmly believe that such a rigorous approach is the only way to obtain elegant solutions to involved polytypic problems. Another concept that simplified the construction and form of the program is arrows. In our first attempts to polytypic programs for packing and unpacking we used monads instead of arrows. Although it is possible to construct the (un)packing functions with monads (see Halenbeek [5]), the inverse function construction, and hence the correctness proof, is simpler with arrows. Loosely speaking, arrows are more easily inverted as input and output are handled symmetrically.

We have shown how to construct programs for several data conversion problems. We expect that our programs and proofs will be very useful in the construction of programs for other data conversion problems.

Although all our data conversion programs are linear, both time and space efficiency of our programs leave much to be desired. We expect that sufficiently sophisticated forms of partial evaluation will improve the performance of our programs considerably. We want to experiment with partial evaluation of polytypic functions in the future.

We have presented a few calculations of polytypic programs. We think that calculating with polytypic functions is still rather cumbersome, and we hope to obtain more theory, in the style of Meertens [18] and Hinze [8], to simplify calculations with polytypic programs. If we take Hinze's approach to polytypic programming [7], then we only have 4 constructors for pattern functors instead of 7, and this should reduce the length of the proofs. In collaboration with Hinze, we are currently working on an implementation of Generic Haskell as a successor to PolyP.

Acknowledgments This paper has been improved by constructive comments by Ralf Hinze on polytypism and general presentation, by Ross Paterson on arrows and their laws and by Roland Backhouse on the fixed point calculations. Joost Halenbeek implemented a polytypic data compression program using monads. The anonymous referees suggested many improvements regarding contents and presentation.

References

[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer-Verlag, 1999.

[2] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice Hall, 1990.

[3] Staffan Björk. Parsers, pretty printers and PolyP. Master's thesis, University of Göteborg, 1997. Examensarbeten 1997:31. Available from the Polytypic programming www page [13].

[4] Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.

[5] J. Halenbeek. Comparing approaches to polytypic programming. Master's thesis, Department of Computer Science, Utrecht University, 1998. INF/SRC-99-02.

[6] Ralf Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the Third Haskell Workshop*, Technical report of Utrecht University, UU-CS-1999-28, 1999.

[7] Ralf Hinze. A new approach to generic functional programming. In *POPL'00*, pages 119–132. ACM Press, 2000.

[8] Ralf Hinze. Polytypic values possess polykinded types. In *Mathematics of Program Construction*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.

[9] John Hughes. Generalising monads to arrows. *Science of Computer Programming, special issue on Mathematics of Program Construction*, 37(1–3):67–112, 2000.

[10] Marieke Huisman. The calculation of a polytypic parser. Master's thesis, Department of Computer Science, Utrecht University, 1996. INF/SRC-96-19.

[11] P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.

[12] P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In Doaitse Swierstra, editor, *ESOP'99*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.

[13] Patrik Jansson. The WWW home page for polytypic programming. Available from `http://www.cs.chalmers.se/~patrikj/poly/`, 2000.

[14] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[15] J. Jeuring. Polytypic pattern matching. In *FPCA'95*, pages 238–248. ACM Press, 1995.

[16] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, 1995.

[17] D.J. Lehmann and M.B. Smyth. Algebraic specification of data types: A synthetic approach. *Math. Systems Theory*, 14:97–139, 1981.

[18] L. Meertens. Calculate polytypically! In *PLILP'96*, volume 1140 of *LNCS*, pages 1–16. Springer-Verlag, 1996.

[19] Ross Paterson. Embedding a class of domain-specific languages in a functional language. Available from `http://www.soi.city.ac.uk/~ross/papers/`, 2000.

[20] Simon Peyton Jones [editor], John Hughes [editor], Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Simon Fraser, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98 — A non-strict, purely functional language. Available from `http://www.haskell.org/definition/`, February 1999.

[21] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Wm. C. Brown Publishers, 1988.

[22] P. Wadler. Comprehending monads. In *Proceedings 1990 ACM Conference on Lisp and Functional Programming*, pages 61–78, 1990.

[23] M. Wallace and C. Runciman. Heap compression and binary I/O in Haskell. In *2nd ACM Haskell Workshop*, 1997.