Polytypic Compact Printing and Parsing

Patrik Jansson[1] and Johan Jeuring[2]

[1] Computing Science, Chalmers University of Technology, Sweden
`http://www.cs.chalmers.se/~patrikj/`
`patrikj@cs.chalmers.se`
[2] Computer Science, Utrecht University, the Netherlands
`http://www.cs.uu.nl/~johanj/`
`johanj@cs.uu.nl`

Abstract. A generic compact printer and a corresponding parser are constructed. These programs transform values of any regular datatype to and from a bit stream. The algorithms are constructed along with a proof that printing followed by parsing is the identity. Since the binary representation is very compact, the printer can be used for compressing data - possibly supplemented with some standard algorithm for compressing bit streams. The compact printer and the parser are described in the polytypic Haskell extension PolyP.

1 Introduction

Many programs convert data from one format to another; examples are parsers, pretty printers, data compressors, encryptors, functions that communicate with a database, etc. 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. We claim that 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 (part of) a data compression program that uses the structure of the input data.

A lot of 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 CD's. Structure-specific compression methods give much better compression results than conventional compression methods such as the Unix compress utility [3,17]. For example, Unix compress typically requires four bits per byte of Pascal program code, whereas Cameron [6] reports compression results of one bit per byte of Pascal program code. Algorithmic Research B.V. [5] sells compressors for structured data, and reports impressive results. Structured compression is also used in heap compression and binary I/O [16].

The basic idea of the structure-specific compression methods is simple: parse the input file into a structured value (an abstract syntax tree), and construct a

S.D. Swierstra (Ed.): ESOP/ETAPS'99, LNCS 1576, pp. 273–287, 1999.

compact representation of the abstract syntax tree. For example, consider the datatype of binary trees

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

The following (rather artificial) example binary tree

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

can be pretty printed to an (admittedly rather wasteful) text description of *tree* requiring 55 bytes. But since the datatype *Tree a* has two constructors, each constructor can be represented by a single bit. Furthermore, the datatype () has only one constructor so the single element 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. Of course, we are not always this lucky, but the average case is still very compact.

This idea has been around since the beginning of the 1980s, but as far as we are aware, there does not exist a general description of the program, only example instantiations appear in the literature. One of the goals of this paper is to describe the compact printing part, together with its inverse, of the compression program generically. It defines a *polytypic* program (a program that works for large classes of datatypes) for compact printing. Together with a parser generator this program is a generic description of the structured compression program. The implementation (as PolyP code) can be obtained from

`http://www.cs.chalmers.se/~patrikj/poly/`

The compression achieved by our compact printing algorithm, is through a compact representation of the structure of the data using only static information — the type of the data. Traditional (bit stream) compressors using dynamic (statistical) properties of the data are largely orthogonal to our approach and thus the best results are obtained by composing the compact printer with a bit stream compressor.

The fundamental property of the compact printing function *print* is that it has a left inverse[1]: the parsing function *parse*. This is a very common specification pattern: all of the example data conversion problems above are specified as pairs of inverse functions with some additional properties. Another example can be found in Haskell's prelude, which contains functions *show* and its inverse *read* of type:

$$\begin{aligned} \mathit{show} &:: \mathit{Show}\ a \ \Rightarrow\ a \to \mathit{String} \\ \mathit{read} &:: \mathit{Read}\ a \ \Rightarrow\ \mathit{String} \to a \end{aligned}$$

[1] That is, $\mathit{parse} \circ \mathit{print} = \mathit{id}$, but $\mathit{print} \circ \mathit{parse}$ need not be id. In the rest of the paper we will write just inverse, when we really mean left inverse.

Unfortunately, it is very hard to see from their definitions why *read* is the inverse of *show*. In this paper, the driving force behind the construction 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 program considerably compared with our previous attempts.

A second desired property of the compact printing function is that given an element x, the length of *print* x is less than the length of *prettyprint* x, where *prettyprint* is a function that prints a value in a standard fashion, like the *show* function of Haskell. This is in general difficult or impossible to prove, and beyond the scope of this paper. More information can be found in the literature [15].

Summarising, this paper has the following goals:

- construct a polytypic compact printing program together with its inverse;
- show how to construct and calculate with polytypic functions;
- take a first step towards a theory of polytypic data conversion.

This paper is organised as follows. Section 2 briefly introduces polytypic programming. Section 3 defines some basic types and classes, and introduces the compact printing program. Section 4 sketches the construction and correctness proof of the compact printing program. Section 5 concludes. Appendix A describes the laws we need in the proofs.

2 Polytypic programming

The compact printing and parsing functions are polytypic functions. This section briefly introduces polytypic functions in the context of the Haskell extension PolyP [9], and defines some basic polytypic concepts used in the paper. We assume that the reader is familiar with the initial algebra approach to datatypes, and not completely unfamiliar with polytypic programming. For an introduction to polytypic programming, see [1,10].

A polytypic function is a function parametrised on type constructors. Polytypic functions are defined either by induction on the structure of user-defined datatypes, 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 to pattern match against values. Instead, we use two built-in functions, *inn* and *out*, to construct and destruct a value of an arbitrary datatype from and to its top level components. With a recursive datatype $d\ a$ as a fixed point of a pattern functor $\Phi_d\, a$, *inn* and *out* are the fold and unfold isomorphisms showing $d\ a \cong \Phi_d\, a\, (d\ a)$.

$$\begin{aligned} \mathit{inn} :: \mathit{Regular}\ d \ &\Rightarrow\ \ (\mathit{FunctorOf}\ d)\ a\ (d\ a) \to d\ a \\ \mathit{out} :: \mathit{Regular}\ d \ &\Rightarrow\ \ d\ a \to (\mathit{FunctorOf}\ d)\ a\ (d\ a) \end{aligned}$$

The pattern functor is used to capture the (top level) structure of a datatype, for example, a list is either empty or contains one element and a recursive occurrence of a list. Hence: $\mathit{FunctorOf}\ \mathit{List} = \mathit{Empty} + (\mathit{Par} * \mathit{Rec})$. Similarly, the pattern functor of the datatype $\mathit{Tree}\,a$ is $\mathit{Par} + (\mathit{Rec} * \mathit{Rec})$. As a last example, the datatype $\mathit{Rose}\,a$ of rose trees over a:

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

has the pattern functor $\mathit{FunctorOf}\ \mathit{Rose} = \mathit{Par} * (\mathit{List}\ @\ \mathit{Rec})$, where @ denotes 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 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.

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 *psum* defined in figure 1. (The subscripts indicating the type are included for readability and are not part of the definition.) Function *psum* sums the integers in a datatype with integers. We use the naming convention that recursive polytypic functions start with a 'p' (as in polytypic) but non-recursive polytypic definitions start with an 'f' (as in functor). The 'polytypic map', *pmap*, takes a function $f :: a \to b$ and a value $x :: d\ a$, and applies f to all a values in x, giving a value of type $d\ b$. The 'functor map', *fmap*, takes two functions $g :: a \to c$ and $h :: b \to d$ and a value $x :: f\ a\ b$, and applies g to all a values in x, and h to all b values in x, giving a value of type $f\ c\ d$. The definitions of *pmap* and *fmap* can be found in the distribution of PolyP.

$$\begin{array}{l}
\mathit{psum} :: \mathit{Regular}\,d \ \Rightarrow \ d\ \mathit{Int} \to \mathit{Int} \\
\mathit{psum} = \mathit{fsum} \circ \mathit{fmap}\ \mathit{id}\ \mathit{psum} \circ \mathit{out} \\
\\
\textbf{polytypic}\ \mathit{fsum}_f :: \mathit{Bifunctor}\,f \ \Rightarrow \ f\ \mathit{Int}\ \mathit{Int} \to \mathit{Int} \\
\quad = \textbf{case}\,f\ \textbf{of} \\
\qquad \begin{array}{rcl}
g + h & \longrightarrow & \mathit{either}\,\mathit{fsum}_g\,\mathit{fsum}_h \\
g * h & \longrightarrow & \lambda(x, y) \to \mathit{fsum}_g\ x + \mathit{fsum}_h\ y \\
\mathit{Empty} & \longrightarrow & \lambda() \to 0 \\
\mathit{Par} & \longrightarrow & \lambda n \to n \\
\mathit{Rec} & \longrightarrow & \lambda s \to s \\
d\ @\ g & \longrightarrow & \mathit{psum}_d \circ (\mathit{pmap}_d\,\mathit{fsum}_g) \\
\mathit{Const}\ t & \longrightarrow & \lambda x \to 0
\end{array} \\
\\
\begin{array}{lcl}
\mathit{pmap} & :: & \mathit{Regular}\,d \ \Rightarrow \ (a \to b) \to d\ a \to d\ b \\
\mathit{fmap} & :: & \mathit{Bifunctor}\,f \ \Rightarrow \ (a \to c) \to (b \to d) \to f\ a\ b \to f\ c\ d
\end{array}
\end{array}$$

Fig. 1. The definition of *psum*

Note that function *psum* is only defined for *Regular* datatypes $d\ a$. A datatype $d\ a$ is regular (satisfies *Regular* d) if it 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. 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 Basic types and classes

Compact printing. A natural choice for the type of a compact printing function for type a is $a \rightarrow \mathit{Text}$, where *Text* is the type of printed values, for example *String* or [*Bit*]. Since we want to define *print* as a recursive function, this would lead to quadratic behaviour when repeatedly concatenating intermediate results. The standard solution for printing functions is to add an accumulating parameter (to which the output is prepended) thus changing the type to $a \rightarrow \mathit{Text} \rightarrow \mathit{Text}$, or equivalently, to $(a, \mathit{Text}) \rightarrow \mathit{Text}$.

Parsing. Parsing is the inverse of printing, and hence a first approximation of its type is $\mathit{Text} \rightarrow a$. Since we want to apply parsers one after the other, we need both a parsed result and the remaining part of the input string, which can be passed to the next parser. The standard solution for parsing functions is to change the type to $\mathit{Text} \rightarrow (a, \mathit{Text})$.

Side effects as functions. We can make the types for printing and parsing more symmetric by pairing the single *Text* component with a unit type to get the isomorphic type $(a, \mathit{Text}) \rightarrow ((), Text)$ for printing and $((), \mathit{Text}) \rightarrow (a, \mathit{Text})$ for parsing. Both these types are instances of the more general type $\mathit{TextStateArr}\ a\ b$:

$$\textbf{newtype}\ \mathit{TextStateArr}\ a\ b = \mathit{TS}\ ((a, \mathit{Text}) \rightarrow (b, \mathit{Text}))$$

An element of type $\mathit{TextStateArr}\ a\ b$ models a function that takes a value of type a and returns a value of type b, and possibly has a side effect on the state *Text*. Thus a compact printer (for a-values) has type $\mathit{TextStateArr}\ a\ ()$, and a corresponding parser has type $\mathit{TextStateArr}\ ()\ a$.

The first steps. Our goal is to construct two functions and a proof:

- A function *pc* ('polytypic compacting') that takes a compact printing program on the element level a to a compact printing program on the datatype level $d\ a$:

  $$\mathit{pc} :: \mathit{TextStateArr}\ a\ () \rightarrow \mathit{TextStateArr}\ (d\ a)\ ()$$

  For example, the function that compresses the tree in the introductory section is obtained by instantiating the polytypic function *pc* to *Tree* and applying the instance to a (trivial) compact printing program for the type ().

- A function *pu* ('polytypic uncompacting') that takes a parsing program on the element level a to a parsing program on the datatype level $d\ a$:

  $$pu :: \mathit{TextStateArr}\ ()\ a \rightarrow \mathit{TextStateArr}\ ()\ (d\ a)$$

  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 c and u are inverses on the element level a, $pc\ c$ and $pu\ u$ are inverses on the datatype level $d\ a$.

In the following section, instead of using the type $\mathit{TextStateArr}\ a\ b$ in the definitions of *pc* and *pu*, we will use the more abstract type $a \rightsquigarrow b$, where $(\rightsquigarrow)$ is an *arrow* type constructor.

The class *Arrow*. The type $\mathit{TextStateArr}\ a\ b$ encapsulates functions from a to b that manipulate a state of type *Text*. Since a parser could easily use a more complicated type, for example to store statically available information [14], and also the printer could use a more complicated type, we will go one step further in the abstraction by introducing the constructor class *Arrow* [8]:

$$\begin{array}{l}
\mathbf{class}\ \mathit{Arrow}\ (\rightsquigarrow)\ \mathbf{where} \\
\quad \begin{array}{lcl}
\mathit{arr} & :: & (a \rightarrow b) \rightarrow (a \rightsquigarrow b) \\
(\ggg) & :: & (a \rightsquigarrow b) \rightarrow (b \rightsquigarrow c) \rightarrow (a \rightsquigarrow c) \\
(\mathbin{<\!+\!>}) & :: & (a \rightsquigarrow c) \rightarrow (b \rightsquigarrow d) \rightarrow (\mathit{Either}\ a\ b \rightsquigarrow \mathit{Either}\ c\ d) \\
\mathit{first} & :: & (a \rightsquigarrow b) \rightarrow ((a, c) \rightsquigarrow (b, c))
\end{array}
\end{array}$$

The method *arr* of the class *Arrow* embeds functions as arrows and $\mathit{arr}\ \mathit{id}$ together with $(\ggg)$ form the signature of the category with types as objects, and elements of $a \rightsquigarrow b$ as arrows from a to b. This category has a binary (sum) functor (the method $(\mathbin{<\!+\!>})$) and a "half-product" functor (*first*). Below we write $\overrightarrow{f}$ as a shorthand notation for $\mathit{arr}\ f$. In the appendix we formalise the properties we need from *Arrow*s to construct the definitions of functions *pc* and *pu* along with the proof of their correctness.

As an example of programming with arrows, we define *second* — the other half-product — in terms of *first*:

$$\begin{array}{l}
\mathit{second} :: (a \rightsquigarrow b) \rightarrow ((c, a) \rightsquigarrow (c, b)) \\
\mathit{second}\ f = \overrightarrow{\mathit{swap}} \ggg \mathit{first}\ f \ggg \overrightarrow{\mathit{swap}} \\
\\
\mathit{swap} :: (a, b) \rightarrow (b, a) \\
\mathit{swap}\ (a, b) = (b, a)
\end{array}$$

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

$$\begin{array}{rcl}
(\ast\!\!\ast\!\!\ast_i) & :: & (a \rightsquigarrow c) \rightarrow (b \rightsquigarrow d) \rightarrow ((a, b) \rightsquigarrow (c, d)) \\
f \ast\!\!\ast\!\!\ast_1 g & = & \mathit{first}\ f \ggg \mathit{second}\ g \\
f \ast\!\!\ast\!\!\ast_2 g & = & \mathit{second}\ f \ggg \mathit{first}\ g
\end{array}$$

Type constructor *TextStateArr* can be made an instance of *Arrow* as follows:

```
mapFst :: (a → b) → (a, c) → (b, c)
mapFst f (a, c) = (f a, c)

instance Arrow TextStateArr where
  arr f          = TS (mapFst f)
  TS f ⋙ TS g   = TS(g ∘ f)
  TS f <+> TS g  = TS (λ(x, t) → either (λa → mapFst Left (f (a, t)))
                                         (λb → mapFst Right (g (b, t)))
                                         x)
  first (TS f)   = TS (λ((a, c), t) → let (b, t′) = f (a, t)
                                      in ((b, c), t′))
```

Printing 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. Since a constructor can be coded by a single natural number, we can use a class *ArrowNat* to characterise arrows that have operations for printing and parsing constructor numbers:

```
class Arrow (⇝) ⇒ ArrowNat (⇝) where
  printCon :: Nat ⇝ ()
  parseCon :: () ⇝ Nat
  -- Requirement: printCon ⋙ parseCon = id⃗
```

With *Text* = [*Nat*], the instances for *TextStateArr* are straightforward, and the printing algorithm constructed in the following section will 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. We will instead statically determine the number of constructors in the datatype and code every single 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 better results if we use Huffman coding with equal probabilities for the constructors, resulting in a variable number of bits per constructor. Even better results are obtained if we analyse the datatype, and give different probabilities to 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. Since the number of bits used per constructor depends on the type of the value that is compressed, *printCon* and *parseCon* need in general be polytypic functions. Their definitions are omitted, but can be found in the code on the web page for this paper.

In the sequel (⇝) will always stand for an arrow type constructor in the class *ArrowNat* but, as with *Regular*, we often omit the type context for brevity.

4 The construction of the program

We want to construct a function pc that takes a compact printing program on the element level a to a compact printing program on the datatype level $d\ a$, together with a parsing function pu, which takes a compact parsing program on the element level a to a compact parsing program on the datatype level $d\ a$, and a proof that pu is the inverse of pc:

$$\begin{aligned} pc &:: (a \rightsquigarrow ()) \to (d\ a \rightsquigarrow ()) \\ pu &:: (() \rightsquigarrow a) \to (() \rightsquigarrow d\ a) \end{aligned}$$

$$c \ggg u = id \quad \Rightarrow \quad pc\, c \ggg pu\, u = pid\, (c \ggg u) = id \tag{1}$$

In the proofs below we will assume that the arrows c and u satisfy $c \ggg u = id$.

Overview of the construction. The construction can be interpreted either as fusing the printer $pc\ c$ with the parser $pu\ u$ to get an identity arrow id or, equivalently, as splitting the identity arrow into a composition of a printer and a parser. As both the printer and the parser are polytypic functions, and both lift an argument level arrow to a datatype level arrow, we start by presenting a polytypic "identity function" pid that lifts an element level identity arrow to a datatype level identity arrow. Function pid is constructed below together with pc and pu and the proof of equation 1 but the resulting definition is presented already here, in figure 2, to aid the reading. The proof that $pid\ id = id$ is simple and omitted. As we are defining polytypic functions the construction follows the structure of regular datatypes: A regular datatype is a fix-point of a pattern functor, the pattern functor is a sum of products, and the products can involve type parameters, other types, etc.

$$\begin{aligned} &pid \quad :: (a \rightsquigarrow b) \to (d\ a \rightsquigarrow d\ b) \\ &pid\, i = \overrightarrow{out} \ggg fid\, i\, (pid\, i) \ggg \overrightarrow{inn} \\ \\ &\textbf{polytypic}\ fid :: (a \rightsquigarrow c) \to (b \rightsquigarrow d) \to (f\ a\ b \rightsquigarrow f\ c\ d) \\ &\quad = \lambda i\, j \to \textbf{case}\, f\ \textbf{of} \\ &\qquad\qquad g + h \quad \longrightarrow fid\, i\, j \mathbin{<\!\!+\!\!>} fid\, i\, j \\ &\qquad\qquad g * h \quad \longrightarrow (fid\, i\, j) \mathbin{*\!\!*\!\!*_1} (fid\, i\, j) \\ &\qquad\qquad Empty \longrightarrow \overrightarrow{id} \\ &\qquad\qquad Par \quad\ \ \longrightarrow i \\ &\qquad\qquad Rec \quad\ \ \longrightarrow j \\ &\qquad\qquad d\ @\ g \quad \longrightarrow pid\, (fid\, i\, j) \end{aligned}$$

Fig. 2. The definition of *pid* and *fid*.

The arrow $pc\, c$ 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 c. The constructor number is computed by means of function *fcSum*, which also takes care of passing on the recursion to the children. An arrow *printCon* prints the constructor number with the correct number of bits. Finally, function *fcProd* makes sure the information is correctly threaded through the children.

Top level recursion. We want function *pc* to be 'on-line' or lazy: it should output compactly printed data immediately, and given part of the compactly printed data, *pu* should reconstruct part of the input value. Thus functions *pc* and *pu* can also be used to compactly print infinite streams, for example. We have not been able to define function *pc* with a standard recursion operator such as the catamorphism: threading the side effects in the right order turned out to be a problem. Instead of a recursion operator we use explicit recursion on the top level, guided by *fc* and *fu*.

As *pc* decomposes its input value, and compactly prints the constructor and the children by means of a function *fc* (defined below), *pu* must do the opposite: first parse the components using *fu* and then construct the top level value:

$$\begin{aligned} pc\;c &= fc\;c\;(pc\;c) \lll \overrightarrow{out} \\ pu\;u &= fu\;u\;(pu\;u) \ggg \overrightarrow{inn} \end{aligned}$$

Here $f \lll g \overset{def}{=} g \ggg f$ is used to reveal the symmetry of the definitions. Thus we need two new functions, *fc* and *fu*, and we can already guess that we will need a corresponding fusion law:

$$\begin{aligned} fc &:: (a \leadsto ()) \to (b \leadsto ()) \to (f\;a\;b \leadsto ()) \\ fu &:: (() \leadsto a) \to (() \leadsto b) \to (() \leadsto f\;a\;b) \end{aligned}$$

$$fc\;c\;c' \ggg fu\;u\;u' = fid\,(c \ggg u)\,(c' \ggg u') \qquad (2)$$

We will use the following variant of fixed-point fusion [12,13]:[2]

$$\mu f \ggg \mu g = \mu h \quad \Leftarrow \quad f\,c' \ggg g\,u' = h\,(c' \ggg u') \qquad (3)$$

Given (2) we can now prove (1).

$$\begin{aligned} & pc\;c \ggg pu\;u = pid\,(c \ggg u) \\ \Leftarrow\quad & \text{Definitions of } pc,\ pu, \text{ fixed-point theorem (3)} \\ & \overrightarrow{out} \ggg fc\;c\;c' \ggg fu\;u\;u' \ggg \overrightarrow{inn} = h\,(c' \ggg u') \\ \equiv\quad & \text{Equation (2).} \\ & \overrightarrow{out} \ggg fid\,(c \ggg u)\,(c' \ggg u') \ggg \overrightarrow{inn} = h\,(c' \ggg u') \\ \equiv\quad & \bullet \text{ Define } h\,j = \overrightarrow{out} \ggg fid\,(c \ggg u)\,j \ggg \overrightarrow{inn} \\ & True \end{aligned}$$

The resulting definition of function *pid* can be found in figure 2.

[2] Strictly speaking the variables c' and u' on the right hand side of the implication should be $\forall$-quantified over $\{f^i \perp \mid i \in \mathbf{N}\}$ and $\{g^i \perp \mid i \in \mathbf{N}\}$ respectively.

Printing constructors. We want to construct functions *fc* and *fu* such that (2) holds. Furthermore, these functions should do the actual compact printing and parsing of the constructors using $printCon :: Nat \rightsquigarrow ()$ and $parseCon :: () \rightsquigarrow Nat$ from the *ArrowNat* class:

$$\begin{aligned} fc\ c\ c' &= printCon \lll fcSum\ c\ c' \\ fu\ u\ u' &= parseCon \ggg fuSum\ u\ u' \end{aligned}$$

The arrow $fcSum\ c\ c'$ prints a value (using the argument printers c and c' 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 *printCon* prepends the constructor number to the output. As $printCon \ggg parseCon = \overrightarrow{id}$ by assumption, the requirement that function *fu* can be fused with *fc* is now passed on to *fuSum* and *fcSum*:

$$\begin{aligned} fcSum &:: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow Nat) \\ fuSum &:: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (Nat \rightsquigarrow f\ a\ b) \end{aligned}$$

$$fcSum\ c\ c' \ggg fuSum\ u\ u' = fid\ (c \ggg u)\ (c' \ggg u') \tag{4}$$

The arrow *parseCon* reads the constructor number and passes it on to the arrow $fuSum\ u\ u'$ 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 an n-ary sum (of products) on the outermost level. This sum is in PolyP represented by a nested binary sum, which associates to the right. Consequently, we define *fcSum* by induction over the nested sum part of the pattern functor and defer the handling of the product part to *fcProd*:

$$\begin{array}{l} \textbf{polytypic}\ fcSum :: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow Nat) \\ \quad = \lambda c\ c' \to \textbf{case}\ f\ \textbf{of} \\ \qquad \begin{array}{rcl} g + h & \longrightarrow & (fcProd\ c\ c' \mathbin{\diamondplus} fcSum\ c\ c') \ggg \overrightarrow{inn_{Nat}} \\ g & \longrightarrow & fcProd\ c\ c' \ggg \lambda() \to 0 \end{array} \end{array}$$

$$\begin{array}{l} \textbf{polytypic}\ fuSum :: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (Nat \rightsquigarrow f\ a\ b) \\ \quad = \lambda u\ u' \to \textbf{case}\ f\ \textbf{of} \\ \qquad \begin{array}{rcl} g + h & \longrightarrow & (fuProd\ u\ u' \mathbin{\diamondplus} fuSum\ u\ u') \lll \overrightarrow{out_{Nat}} \\ g & \longrightarrow & fuProd\ u\ u' \lll \lambda 0 \to () \end{array} \end{array}$$

The types for *fcProd* and *fuProd* and the corresponding fusion law are unsurprising:

$$\begin{aligned} fcProd &:: (a \rightsquigarrow ()) \to (b \rightsquigarrow ()) \to (f\ a\ b \rightsquigarrow ()) \\ fuProd &:: (() \rightsquigarrow a) \to (() \rightsquigarrow b) \to (() \rightsquigarrow f\ a\ b) \end{aligned}$$

$$\mathit{fcProd}\ c\ c' \ggg \mathit{fuProd}\ u\ u' = \mathit{fid}\ (c \ggg u)\ (c' \ggg u') \tag{5}$$

We prove equation (4) by induction over the nested sum structure of the functor. The induction hypothesis is that (4) holds for the fcSum_h.

The sum case: $g + h$

$$\begin{aligned}
& \mathit{fcSum}_{g+h}\ c\ c' \ggg \mathit{fuSum}_{g+h}\ u\ u' \\
= \quad & \text{Definitions} \\
& (\mathit{fcProd}_g\ c\ c' \mathbin{<\!+\!>} \mathit{fcSum}_h\ c\ c') \ggg \overrightarrow{\mathit{inn}_{Nat}} \ggg \\
& \overrightarrow{\mathit{out}_{Nat}} \ggg (\mathit{fuProd}_g\ u\ u' \mathbin{<\!+\!>} \mathit{fuSum}_h\ u\ u') \\
= \quad & \mathit{out}_{Nat} \circ \mathit{inn}_{Nat} = \mathit{id} \\
& (\mathit{fcProd}_g\ c\ c' \mathbin{<\!+\!>} \mathit{fcSum}_h\ c\ c') \ggg (\mathit{fuProd}_g\ u\ u' \mathbin{<\!+\!>} \mathit{fuSum}_h\ u\ u') \\
= \quad & (\mathbin{<\!+\!>}) \text{ is a bifunctor} \\
& (\mathit{fcProd}_g\ c\ c' \ggg \mathit{fuProd}_g\ u\ u') \mathbin{<\!+\!>} (\mathit{fcSum}_h\ c\ c' \ggg \mathit{fuSum}_h\ u\ u') \\
= \quad & \text{Equation (5) and the induction hypothesis} \\
& \mathit{fid}_g\ (c \ggg u)\ (c' \ggg u') \mathbin{<\!+\!>} \mathit{fid}_h\ (c \ggg u)\ (c' \ggg u') \\
= \quad & \bullet \text{ Define } \mathit{fid}_{g+h} \\
& \mathit{fid}_{g+h}\ (c \ggg u)\ (c' \ggg u')
\end{aligned}$$

The base case: g

$$\begin{aligned}
& \mathit{fcProd}_g\ c\ c' \ggg \lambda() \to 0 \ggg \lambda 0 \to () \ggg \mathit{fuProd}_g\ u\ u' \\
= \quad & \overrightarrow{\lambda() \to 0} \ggg \overrightarrow{\lambda 0 \to ()} = \mathit{id} :: () \leadsto () \\
& \mathit{fcProd}_g\ c\ c' \ggg \mathit{fuProd}_g\ u\ u' \\
= \quad & \text{Equation (5)} \\
& \mathit{fid}_g\ (c \ggg u)\ (c' \ggg u')
\end{aligned}$$

Sequencing the parameters. The last part of the construction of the program is the two functions *fcProd* and *fuProd* defined in figure 3. The earlier functions have calculated and printed the constructors, so what is left is "arrow plumbing". The arrow $\mathit{fcProd}\ c\ c'$ traverses the top level structure of the data and inserts the correct compact printers: c at argument positions and c' at substructure positions. The structure of *fuProd* is very similar but as it is the inverse of *fcProd*, all arrows are composed in the opposite order. The inverse proof is a relatively straightforward induction over the pattern functor structure, but omitted here due to space constraints.

$$
\begin{array}{l}
\textbf{polytypic } \mathit{fcProd} :: (a \leadsto ()) \to (b \leadsto ()) \to (f\ a\ b \leadsto ()) \\
\quad = \lambda c\ c' \to \textbf{case } f \textbf{ of} \\
\qquad \begin{array}{rcl}
g * h & \longrightarrow & (\mathit{fcProd}\ c\ c') \ast\!\!\ast\!\!\ast_2 (\mathit{fcProd}\ c\ c') \ggg \overrightarrow{\lambda((),()) \to ()} \\
\mathit{Empty} & \longrightarrow & \overrightarrow{\mathit{id}} \\
\mathit{Par} & \longrightarrow & c \\
\mathit{Rec} & \longrightarrow & c' \\
d\ @\ g & \longrightarrow & \mathit{pc}\,(\mathit{fcProd}\ c\ c')
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{polytypic } \mathit{fuProd} :: (() \leadsto a) \to (() \leadsto b) \to (() \leadsto f\ a\ b) \\
\quad = \lambda u\ u' \to \textbf{case } f \textbf{ of} \\
\qquad \begin{array}{rcl}
g * h & \longrightarrow & (\mathit{fuProd}\ u\ u') \ast\!\!\ast\!\!\ast_1 (\mathit{fuProd}\ u\ u') \lll \overrightarrow{\lambda() \to ((),())} \\
\mathit{Empty} & \longrightarrow & \overrightarrow{\mathit{id}} \\
\mathit{Par} & \longrightarrow & u \\
\mathit{Rec} & \longrightarrow & u' \\
d\ @\ g & \longrightarrow & \mathit{pu}\,(\mathit{fuProd}\ u\ u')
\end{array}
\end{array}
$$

Fig. 3. The definition of *fcProd* and *fuProd*.

5 Conclusions

Results

- We have constructed a polytypic program for compact printing and parsing of structured data. As far as we are aware, this is the first generic description of a program for compact printing (structured data compression).
- The pair of functions for compact printing and parsing are inverse functions by construction. Since we started applying the inverse function requirement rigorously in the construction of the program, the size and the complexity of the code have been reduced considerably. We think 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 we used monads instead of arrows. Although it is perfectly well possible to construct the compact printing and parsing functions with monads [7], the inverse function construction, and hence the correctness proof, is much simpler with arrows.
- We have shown how to convert data to and from a bit stream. This is an example of a data conversion program, and we hope that the construction in this paper is reusable in solutions for other data conversion problems.

Future work

- The current program produces compact, but not human-readable, output. A pretty printer for structured data has a very similar structure, and we want to investigate how to introduce the right abstractions to obtain a single program for both pretty printing and compact printing of structured data.

- In the future we want to investigate whether or not relations can help to simplify the construction even more, by specifying compact printing as a relation, and letting parsing be its relational converse [2,4].
  We have presented a calculation of a polytypic program. We think that calculating with polytypic functions is still rather cumbersome, and we hope to obtain more theory, in the style of [11], to further simplify calculations with polytypic programs.
- We want to construct polytypic programs for other data conversion problems such as encryption and database communication.

Acknowledgements. Roland Backhouse helped with the fixed point calculation. Joost Halenbeek implemented a polytypic data compression program using monads. The anonymous referees suggested many improvements for contents and presentation.

References

1. R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming — an introduction. To appear in AFP'98, LNCS, Springer-Verlag, 1998.
2. R.C. Backhouse, P.J. de Bruin, P. Hoogendijk, G. Malcolm, T.S. Voermans, and J.C.S.P. van der Woude. Relational catamorphisms. In B. Möller, editor, *Constructing Programs from Specifications*, pages 287–318. North-Holland, 1991.
3. Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression.* Prentice Hall, 1990.
4. R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice-Hall International, 1996.
5. Algorithmic Research B.V. SDR compression products. See `http://www.algoresearch.com/`, 1998.
6. Robert D. Cameron. Source encoding using syntactic information source models. *IEEE Transactions on Information Theory*, 34(4):843–850, 1988.
7. J. Halenbeek. Comparing approaches to generic programming. Master's thesis, Department of Computer Science, Utrecht University, 1998. To appear.
8. John Hughes. Generalising monads. Invited talk at MPC'98, 1998. Slides available from `http://www.md.chalmers.se/Conf/MPC98/talks/JohnHughes/`.
9. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.
10. J. Jeuring and P. Jansson. Polytypic programming. In *AFP'96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.
11. L. Meertens. Calculate polytypically! In *PLILP'96*, volume 1140 of *LNCS*, pages 1–16. Springer Verlag, 1996.
12. Erik Meijer. *Calculating compilers.* PhD thesis, Nijmegen University, 1992.
13. Mathematics of Program Construction Group (Eindhoven Technical University). Fixed-point calculus. *Information Processing Letters*, 53(3):131–136, 1995.
14. Swierstra S.D. and L. Duponcheel. Deterministic, error-correcting combinator parsers. In John Launchbury, Erik Meijer, and Tim Sheard, editors, *Advanced Functional Programming*, LNCS 1129, pages 184–207. Springer-Verlag, 1996.
15. R.G. Stone. On the choice of grammar and parser for the compact analytical encoding of programs. *The Computer Journal*, 29(4):307–314, 1986.

16. M. Wallace and C. Runciman. Heap compression and binary I/O in haskell. In *2nd ACM Haskell Workshop*, 1997.
17. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

A Properties of *Arrow*s

The properties we need from an *Arrow* type constructor for the definitions of the generic printer and parser are most succinctly described using category theoretic terminology. We work in a base category $\mathcal{C}$ of (Haskell-) types and functions and a type constructor $\rightsquigarrow$ is an *Arrow* if we have a category $\mathcal{A}$ with the same types as objects, but with elements of $a \rightsquigarrow b$ as arrows:

$$\begin{aligned}\mathcal{C} &= (H, (\rightarrow), (\circ), id)\\ \mathcal{A} &= (H, (\rightsquigarrow), (\lll), id)\end{aligned}$$

Furthermore $\mathcal{A}$ must have a binary (sum) functor $((<\!+\!>) : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A})$, "half-product" functors $(\mathit{first}_c : \mathcal{A} \rightarrow \mathcal{A})$ and there must be a functor $(\overrightarrow{\cdot} : \mathcal{C} \rightarrow \mathcal{A})$ lifting functions to arrows. A set of laws sufficient for the proof of the correctness of the *print*-*parse*-pair is given in figure 4. We do not require the stronger requirements that $(<\!+\!>)$ should be a true categorical sum or that first_c (combined with second_c) should give a categorical product as this would rule out many useful arrow type constructors. In fact, the proof goes through even with slightly weaker conditions on the arrows than those in figure 4, and thus we may be able to extend the class of possible arrows further.

$\mathcal{A}$ is a category	$id \lll f = f = f \lll id$	$(\lll, id)$
	$(f \lll g) \lll h = f \lll (g \lll h)$	$(\lll,\lll)$
$\overrightarrow{\cdot} : \mathcal{C} \rightarrow \mathcal{A}$	$\overrightarrow{id} = id$	$(\overrightarrow{\cdot}, id)$
$\overrightarrow{a} = a$	$\overrightarrow{f \circ g} = \overrightarrow{f} \lll \overrightarrow{g}$	$(\overrightarrow{\cdot},\lll)$
$(<\!+\!>) : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$	$id <\!+\!> id = id$	$(<\!+\!>, id)$
$a <\!+\!> b = \mathit{Either}\ a\ b$	$(f <\!+\!> g) \lll (f' <\!+\!> g') = (f \lll f') <\!+\!> (g \lll g')$	$(<\!+\!>,\lll)$
$\mathit{first}_c : \mathcal{A} \rightarrow \mathcal{A}$	$\mathit{first}_c\ id = id$	(first_c, id)
$\mathit{first}_c\ a = (a, c)$	$\mathit{first}_c\ (f \lll g) = \mathit{first}_c\ f \lll \mathit{first}_c\ g$	$(\mathit{first}_c,\lll)$
	$\mathit{first}_c\ (f <\!+\!> g) = \mathit{first}_c\ f <\!+\!> \mathit{first}_c\ g$	$(\mathit{first}_c,<\!+\!>)$

Fig. 4. Laws for *Arrow*s.

We denote reverse composition in $\mathcal{A}$ with $(\ggg)$ and we often use the obvious variants of the laws for this operator. When translating the *Arrow* requirements back to a Haskell class we omit *id* as it is equal to $\overrightarrow{id}$. The resulting code is shown in figure 5 where we also introduce some useful abbreviations.

```
class Arrow a where
  arr    ::  (b -> c) -> a b c
  (>>>)  ::  a b c -> a c d -> a b d
  (|||)  ::  a b d -> a c d -> a (Either b c) d
  (<+>)  ::  a b d -> a c e -> a (Either b c) (Either d e)
  first  ::  a b c -> a (b,d) (c,d)
  second ::  a b c -> a (d,b) (d,c)
  -- Defaults:
  f <+> g = (f >>> arr Left) ||| (g >>> arr Right)
  f ||| g = (f <+> g) >>> arr (either id id)

  second f = arr swap >>> first f  >>> arr swap
  first  f = arr swap >>> second f >>> arr swap

-- Utilities
(<<<) :: Arrow a => a c d -> a b c -> a b d
g <<< f = f >>> g

swap :: (a,b) -> (b,a)
swap   ~(x,y) =  (y,x)

data Nat = Z | S Nat

innNat :: Either () Nat -> Nat
innNat  = either (const Z) S

outNat :: Nat   ->  Either () Nat
outNat    (Z)    =  Left   ()
outNat    (S n)  =  Right     n

-- Either and either are predefined in Haskell
data Either a b  =  Left a | Right b

either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)   =  f x
either f g (Right x)  =  g x
```

Fig. 5. The *Arrow* operations as Haskell code.