Polytypic Data Conversion Programs
P. Jansson and J. Jeuring

Presentation by Cas Plattel

October 23, 2012
Table of Contents

1 Introduction

2 Definitions

3 Polytypic programming

4 Results

5 Conclusion
What is this paper about?

Data conversion programs

Programs that:

1. Determine the shape of data
2. Traverse data
3. Package data
4. Pretty print

They have in common that each of them has an inverse

Conversion programs are expressed using John Hughes’ arrows. The laws needed to prove the correctness of conversion can be seen as restrictions on the possible implementations.
Introduction

What is this paper about?

Goals

- Construct a number of polytypic programs for data conversion problems, with their inverses
- Show how to construct and calculate with polytypic functions

Connections with the course

- Provides programs that work on generic datatypes
- Served as a base for Generic Haskell
Table of Contents

1 Introduction

2 Definitions

3 Polytypic programming

4 Results

5 Conclusion
Polytypic function

A polytypic function is a function parametrized on type constructors. Can either be defined by induction on the structure of user-defined datatypes or by other functions.

Forward composition

\[ f ; g = g \cdot f \]
In this paper most code is created in PolyP

PolyP extends Haskell with a construct for writing polytypic functions.

- Looks a lot like generic libraries
- Data conversion functions are polytypic functions in PolyP
- Has functions \textit{in} and \textit{out} which work like the \textit{from} and \textit{to} (as seen in the course)
PolyP pattern functors grammar:

\[ f, g, h ::= g + h \mid g \ast h \mid \text{Empty} \mid \text{Par} \mid \text{Rec} \mid d \circ g \mid \text{Const } t \]

For instance we can write a Rose in PolyP grammar in this way:

\[
\textbf{data } \text{Rose } a = \text{Node } a \ (\text{List } (\text{Rose } a ) ) \\
\Phi_{\text{Rose}} = \text{Par} \ast (\text{List } @ \text{Rec})
\]
Definitions
Map functions in PolyP

Map functions

\[ \text{map}_d :: \text{Regular } d \Rightarrow (a \rightarrow b) \rightarrow (d \ a \rightarrow d \ b) \]
\[ \text{map}_d p = \text{out}_d ; \text{map}_2 d p (\text{map}_d p) ; \text{in}_d \]
\[ \text{map}_2 f :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (f \ a \ b \rightarrow f \ c \ d) \]

\textbf{polytypic} \quad \text{map}_2 :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (f \ a \ b \rightarrow f \ c \ d)

\[ = \lambda p \ p' \rightarrow \textbf{case } f \ \textbf{of} \]
\[ g \ + \ h \rightarrow \text{map}_2 g p p' + \text{map}_2 h p p' \]
\[ g \ast h \rightarrow \text{map}_2 g p p' \ast \text{map}_2 h p p' \]
\[ \text{Empty} \rightarrow \text{id} \]
\[ \text{Par} \rightarrow p \]
\[ \text{Rec} \rightarrow p' \]
\[ d \ @ g \rightarrow \text{map}_d (\text{map}_2 g p p') \]
\[ \text{Const } t \rightarrow \text{id} \]
Table of Contents

1. Introduction

2. Definitions

3. Polytypic programming
   - Shape and contents
   - Additional definitions: Arrows
   - Traversal
   - Packaging
   - Show and read

4. Results

5. Conclusion
Shape and contents

Functions are shown for

- Separating a datatype value into its shape and its contents
- Combining shape and contents to a datatype value

```
separate :: Regular d ⇒ d a → (d(), [a])
separate x = (shape x, flatten x)
shape :: Regular d ⇒ d a → d()
shape = map (const ())
```
Writing *combine* would be slightly harder, we'll have to traverse the shape and insert elements from a list.

**Problem**

How do we prove *combine* is the inverse of *separate*?

**Solution:** arrows!
Fuse *flatten* and *shape* together into an arrow map that keeps track of state and does one single traversal. To avoid altering all types with state information a new type is created, constructor \( SA \), for functions that side-effect on a state.

```haskell
data SA s a b = SA ((a, s) \rightarrow (b, s))
```

Notation for this in the paper is \( a \leadsto_s b \).
Now we can use this type and an arrow map (called \( \text{mapAr} \)) to redefine \( \text{separate} \).

\[
\begin{align*}
\text{separate} & : \ d \ a \ \simrightarrow_{[a]} \ d() \\
\text{separate} & = \text{mapAr} \ \text{put} \\
\text{put} & : \ a \ \simrightarrow_{[a]} \ () \\
\text{put} & = \text{SA} \ (\lambda \ (a, \ as) \rightarrow ((), \ a : \ as)) \\
\text{mapAr} & : \ (a \ \simrightarrow_{s} \ b) \rightarrow (d \ a \ \simrightarrow_{s} \ d \ b)
\end{align*}
\]

The \( r \) in \( \text{mapAr} \) denotes the direction of the traversal, for instance, a given tree would be traveled right first then the left part.
Using the left to right traversing variant of the arrow map, \( \text{mapAl} \), we can write \( \text{combine} \).

\[
\begin{align*}
\text{combine} & \colon d(\) \rightsquigarrow[a] d\ a \\
\text{combine} & = \text{mapAl get} \\
\text{get} & \colon () \rightsquigarrow[a] a \\
\text{get} & = \text{SA} (\lambda ((), a : as) \to (a, as)) \\
\text{mapAl} & \colon (a \rightsquigarrow_s b) \to (d\ a \rightsquigarrow_s d\ b)
\end{align*}
\]

Now with both \( \text{separate} \) and \( \text{combine} \) as arrow maps we can show that they’re each others inverses.
Because we can’t use function composition on our SA we create a new composition operator

\[(\gg) :: (a \rightsquigarrow_s b) \rightarrow (b \rightsquigarrow_s c) \rightarrow (a \rightsquigarrow_s c)\]

\[\text{SA } f \gg \text{SA } g = \text{SA } (f ; g)\]

Now we can proof separate and combine are inverses by proving put and get are inverses (recall definitions of separate and combine)

\[\text{put } \gg \text{get} = \{\text{Definitions of get and put}\}\]

\[\text{SA } (\lambda (a, as) \rightarrow ((), a : as)) \gg \text{SA } (\lambda ((), a : as) \rightarrow (a, as))\]

\[= \{\text{Definition of (\gg) }\} \]

\[\text{SA } (\lambda (a, as) \rightarrow ((), a : as)) ; \text{SA } (\lambda ((), a : as) \rightarrow (a, as))\]

\[= \{\text{Simplification}\}\]

\[\text{SA } id\]
Using arrows to generalize monads gives a wider applicability.

We create a function that lifts function to the level where they can take and return states.

\[
\text{arr :: (a } \rightarrow \text{ b) } \rightarrow \text{ (a } \leadsto_{s} \text{ b)}
\]
\[
\text{arr f = SA (f } \leftarrow_{s} \text{ id)}
\]

A different way to write this is \( \overrightarrow{f} \) instead of \( \text{arr f} \).
For pairs a function \textit{first} is defined which applies an arrow to the first component and leaves the second unchanged.

\begin{align*}
\text{first} &:: \ (a \rightsquigarrow_s b) \rightarrow ((a, c) \rightsquigarrow_s (b, c)) \\
\text{first} \ (SA \ f) &= \ SA \ (\lambda ((a, c), s) \rightarrow \text{let} \ (b, s') = f \ (a, s) \ \text{in} \ ((b, c), s'))
\end{align*}

A function \textit{second} does the same but then for the second component.

\begin{align*}
\text{second} &:: \ (a \rightsquigarrow_s b) \rightarrow ((c, a) \rightsquigarrow_s (c, b)) \\
\text{second} \ f &= \ 	ext{\underline{swap}} \ \gg \ \text{\underline{swap}} \ \gg \ \text{first} \ f \ \gg \ 	ext{\underline{swap}}
\end{align*}
Type SA \( s \ a \ b \) encapsulates function from \( a \) to \( b \) that manipulate state \( s \)

But what about functions that don’t refer to the state?

```haskell
class Arrow (↝) where
  arr :: (a → b) → (a ↝ b)
  (↝) :: (a ↝ b) → (b ↝ c) → (a ↝ c)
  first :: (a ↝ b) → ((a, c) ↝ (b, c))
```

Arrows imply a number of laws (for sake of time left out here).
Subclass \textit{ArrowChoice} for choice operators.

\begin{verbatim}
class Arrow (↝) ⇒ ArrowChoice (↝) where
  (+++) :: (a ↝ c) → (b ↝ d) → (Either a b ↝ Either c d)
  (|||) :: (a ↝ c) → (b ↝ d) → (Either a b ↝ c)
\end{verbatim}

Combining the classes \textit{Arrow} and \textit{ArrowChoice} we can create instances for \textit{SA}s and with those create easier proof for our conversion programs.
How to traverse a datatype? We’ve seen \textit{mapAr} and \textit{mapAl} before.

\[
\text{mapAr} :: \text{ArrowChoice} (\rightsquigarrow) \Rightarrow (a \rightsquigarrow b) \rightarrow (d \ a \rightsquigarrow d \ b)
\]

\[
\text{mapAl} :: \text{ArrowChoice} (\rightsquigarrow) \Rightarrow (a \rightsquigarrow b) \rightarrow (d \ a \rightsquigarrow d \ b)
\]

Their definitions of their arrow maps:

\[
\text{mapAr}_d \ p = \overrightarrow{\text{out}}_d \ 	ext{TR}_d \ p \ (\text{mapAr}_d \ p) \ 	ext{TR}_d \ \overrightarrow{\text{in}}_d
\]

\[
\text{mapAl}_d \ u = \overrightarrow{\text{out}}_d \ 	ext{TL}_d \ u \ (\text{mapAl}_d \ u) \ 	ext{TL}_d \ \overrightarrow{\text{in}}_d
\]

TR and TL are the traversal functions.
Traversing functions

Their definitions:

**polytypic** \( TR_f :: (a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (f \ a \ b \rightarrow f \ c \ d) \)

\[
= \lambda p \ p' \rightarrow \text{case } f \ of \\
g + h \rightarrow TR_g \ p \ p' + + + TR_h \ p \ p' \\
g \ast h \rightarrow TR_g \ p \ p' \triangleleft TR_h \ p \ p' \\
\text{Empty} \rightarrow \text{id} \\
\text{Par} \rightarrow p \\
\text{Rec} \rightarrow p' \\
d \odot g \rightarrow \text{mapAr}_d(\ TR_g \ p \ p') \\
\text{Const } t \rightarrow \text{id}
\]

TL is quite similar but uses \( \ast \Rightarrow \) in the product case to traverse from left to right and some recursive calls are changed.
Packaging

The next program the paper discusses is a packaging program, let's first define this:

**Packaging program**

Given a datatype value (an abstract syntax tree), construct a compact (bit stream) representation of the abstract syntax tree.

An example is the binary tree `treeShape`:

```haskell
treeShape :: Tree ()
```

```haskell
treeShape = Bin( Bin( Leaf() ) ( Bin( Leaf() ) ( Leaf() ))) ( Leaf() )
```

Which can be converted to the following bit stream:

```
Bin (Bin (Leaf ())) (Bin (Leaf ())) (Leaf ())) (Leaf ())
1 1 0 1 0 0 0
```
Creating a packaging program we need to do three things

- Define function `pack`, which takes an element level packer to a datatype level packer

\[
\text{pack} :: (a \rightsquigarrow ()) \rightarrow (d \ a \rightsquigarrow ())
\]

- Define function `unpack`, which takes an unpacker on the element level and takes it to an unpacker on the datatype level

\[
\text{unpack} :: (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow d \ a)
\]

- Proof that if \( p \) and \( u \) are inverses on element \( a \), then `pack \ p` and `unpack \ p` are inverses on the datatype \( d \ a \)
Functions \textit{pack} and \textit{unpack} are defined by recursion on top level by $PT$ and $UT$, packing and unpacking functions on top level.

\[
\text{pack}_d \ p = PT \ \text{noOfCons}_d \ p \ (\text{pack}_d \ p) \ll out
\]
\[
\text{unpack}_d \ u = UT \ \text{noOfCons}_d \ u \ (\text{unpack}_d \ u) \gg in
\]

The arrow \textit{pack} $p$ uses printer $p$ for printing arguments. The function \textit{noOfCons} is just an integer which is the number of constructors possible.
Packaging
Actual packing and unpacking

\[ P_T :: \text{Int} \rightarrow (a \rightsquigarrow ()) \rightarrow (b \rightsquigarrow ()) \rightarrow (f \ a \ b \rightsquigarrow ()) \]
\[ P_T \ n \ p \ p' = \text{packCon} \ n \lll P_S \ p \ p' \]

\( P_S \) packs a value with \( p \) for parameters and \( p' \) for recursive structures. Arrow \( \text{packCon} \) converts constructors to numbers.

\[ U_T :: \text{Int} \rightarrow (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow b) \rightarrow (() \rightsquigarrow f \ a \ b) \]
\[ U_T \ n \ u \ u' = \text{unpackCon} \ n \rrr U_S \ u \ u' \]

Arrow \( \text{unpackCon} \) takes a constructor number and converts it to a constructor. \( U_S \) uses parsers \( u \) to fill parameters and \( u' \) for the recursive slots.
The last part is defining $P_S$ and $U_S$, very similar to $mapAr$ and $mapAl$ since they traverse the data structure.

- Difference between $P_S$ and $U_S$ is (like $mapAr$ and $mapAl$) the traversal direction
- Inverse proof is very similar
- Differ in the way sums are handled
How to define polytypic versions of \textit{show} and \textit{read}?

- New class \texttt{ArrowReadShow} with four classes of operations:
  1. \texttt{ArrowZero} (error handling)
  2. \texttt{ArrowPlus} (error handling)
  3. \texttt{ArrowSymbol} (print/parse symbols)
  4. \texttt{ArrowPrec} (operator precedences)
Data conversion with possible failure. ArrowZero is defined for pure failure

```haskell
class Arrow (↝) ⇒ ArrowZero (↝) where
    zeroA :: a ↝ b
```

ArrowPlus has operators which take alternatives if one argument fails.

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

(<|>) makes choice depending on hidden state or depending on summand in output.
(<+>) uses second arrow argument if first fails.
Show and read
Reading and writing symbols

Class for printing and parsing symbols:

```
class Arrow (⇝) ⇒ ArrowSymbol (⇝) where
  readSym :: Symbol → (a ⇝ a)
  showSym :: Symbol → (a ⇝ a)

type Symbol = String
```

Class that handles precedence (priority) levels.

```
class ArrowSymbol (⇝) ⇒ ArrowPrec (⇝) where
  setPrec :: Prec → (a ⇝ b) → (a ⇝ b)
  readPrec :: Prec → (a ⇝ b) → (a ⇝ b)
  showPrec :: Prec → (a ⇝ b) → (a ⇝ b)
```
The polytypic functions *show* and *read* defined in the paper use the operators from the previous declared classes.

```haskell
class (ArrowChoice (↝), ArrowPlus (↝), ArrowPrec (↝)) ⇒ ArrowShowRead (↝)
```

We create the conversion program by using four levels:

1. Top level recursion: *show* and *read*
2. Breaking down sum structure: *Ss* and *Rs*
3. Breaking down product structure: *Sp* and *Rp*
4. Dealing with parameters and other datatypes: *Sr* and *Rr*
Show and read

The four levels will in short do:

1. Expose top level and handle recursion
2. Handle parentheses
3. Insert spaces between arguments of constructors
4. Apply show (or read) functions on parameters
Show and read
The code

\[
\text{show}_d \ s = \text{Ss}_d \ \text{constructors}_d \ s \ (\text{show}_d \ s) \ \ll \ \text{out}_d \\
\text{read}_d \ r = \text{Rs}_d \ \text{constructors}_d \ r \ (\text{read}_d \ r) \ \gg \ \text{in}_d
\]

\[
\text{Ss}_f :: [\text{Constructor}] \rightarrow (a \rightsquigarrow ()) \rightarrow (b \rightsquigarrow ()) \rightarrow (f \ a \ b \rightsquigarrow ()) \\
\text{Rs}_f :: [\text{Constructor}] \rightarrow (() \rightsquigarrow a) \rightarrow (() \rightsquigarrow b) \rightarrow (() \rightsquigarrow f \ a \ b)
\]

Where \([\text{Constructor}]\) is the list of constructors of datatype \(d \ a\).

\text{polytypic} \ \text{Ss}_f :: [\text{Constructor}] \rightarrow (a \rightsquigarrow ()) \rightarrow (b \rightsquigarrow ()) \rightarrow (f \ a \ b \rightsquigarrow ()) \\
= \lambda(c : cs) \ s \ s' \rightarrow \text{case } f \ \text{of} \\
g + h \rightarrow \text{Ss}_f \ [c] \ s \ s' \ ||| \ \text{Ss}_h \ cs \ s \ s' \\
g \rightarrow \text{showParen}(\text{prec } c)(\text{showSym}(\text{name } c) \ \ll \ \text{Sp}_g \ s \ s')
Show and read
Products

Product case for show:

**polytypic** \( Sp_f :: (a \leadsto ()) \rightarrow (b \leadsto ()) \rightarrow (f \ a \ b \leadsto ()) \)

\[ = \lambda s \ s' \rightarrow \text{case } f \text{ of} \]
\[ g \ast h \rightarrow \lambda() \rightarrow ((()),()) \ll (Sp_g \ s \ s' \leftarrow* \ Sp_h \ s \ s') \]
\[ \text{Empty} \rightarrow \lambda() \rightarrow () \]
\[ g \rightarrow \text{showSym " " } \ll \text{setPrec high } (Sr_g \ s \ s') \]

Rest case for show:

**polytypic** \( Sr_f :: (a \leadsto ()) \rightarrow (b \leadsto ()) \rightarrow (f \ a \ b \leadsto ()) \)

\[ = \lambda s \ s' \rightarrow \text{case } f \text{ of} \]
\[ Par \rightarrow s \]
\[ Rec \rightarrow s' \]
\[ d @ g \rightarrow \text{show}_d (Sr_g \ s \ s') \]
Table of Contents

1 Introduction

2 Definitions

3 Polytypic programming

4 Results

5 Conclusion
We’ve seen four data conversion programs being defined:

**Shape and contents**

- `separate :: d a ⇝ a d ()`
- `combine :: d () ⇝ a d a`

\[ \text{separate} \gg \text{combine} = \text{id} \]

**Arrow maps (traversal)**

- `mapAr :: ArrowChoice (⇝) ⇒ (a ⇝ b) → (d a ⇝ d b)`
- `mapAl :: ArrowChoice (⇝) ⇒ (a ⇝ b) → (d a ⇝ d b)`

\[ p \gg u = \overrightarrow{i} ⇒ \text{mapAr } p \gg\gg \text{mapAl } u = map \overrightarrow{i} \]
Results

Packing

\[ \text{pack} :: \text{ArrowPack} (\rightsquigarrow) \Rightarrow (a \rightsquigarrow \texttt{()}) \rightarrow (d \ a \rightsquigarrow \texttt{()}) \]

\[ \text{unpack} :: \text{ArrowPack} (\rightsquigarrow) \Rightarrow (\texttt{()} \rightsquigarrow a) \rightarrow (\texttt{()} \rightsquigarrow d \ a) \]

\[ p \gg u = \map i \Rightarrow \text{pack} \ p \gg \text{unpack} \ u = \map i \]

Pretty printing

\[ \text{show} :: \text{ArrowShowRead} (\rightsquigarrow) \Rightarrow (a \rightsquigarrow \texttt{()}) \rightarrow (d \ a \rightsquigarrow \texttt{()}) \]

\[ \text{read} :: \text{ArrowShowRead} (\rightsquigarrow) \Rightarrow (\texttt{()} \rightsquigarrow a) \rightarrow (\texttt{()} \rightsquigarrow d \ a) \]

\[ s \gg r = \map i \Rightarrow \text{show} \ s \gg \text{read} \ r = \map i \]

These last two can be used together to create compression and decompression.
Since inverse functions are implemented inside the construction of the programs, the size and complexity of the code are reduced compared to other work (example: polytypic read and show by Björk and Huisman).

Time and space efficiency not optimal, might be improved in future

Using arrows creating the programs simplified the construction and form, made it easier to create inverses and proof their correctness.

The programs shown in the paper serve as a base for other programs that solve their data conversion problems.

For instance, Generic Haskell is based upon this
Conclusion
Paper research conclusion

- Has a nice introduction
- Long read because of the new definitions (arrows, etc)
- Interesting concepts