A SIMPLE APPROACH TO GENERIC FUNCTIONAL PROGRAMMING

RALF HINZE

Institut für Informatik III, Universität Bonn, Römerstraße 164
53117 Bonn, Germany
E-mail: ralf@informatik.uni-bonn.de

February, 1999

A brief guide to the lecture

Why?—Lists and $size$

Observation: There are many interesting data structures for which we may want to define a $size$ or $length$ function.

$$\textbf{data}\ List\ a \quad = \quad Nil \mid Cons\ a\ (List\ a)$$

Here is a simple list.

$$2\ 7\ 19\ 65 \quad \Longrightarrow \quad Cons\ 2\ (Cons\ 7\ (Cons\ 19\ (Cons\ 65\ Nil)))$$

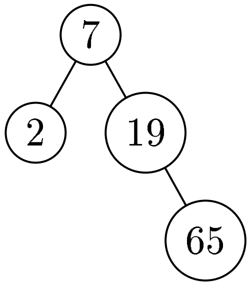
The function $size$ is defined inductively on the structure of $List$.

$$\begin{array}{lcl} size & :: & List\ a \rightarrow Int \\ size\ Nil & = & 0 \\ size\ (Cons\ a\ as) & = & 1 + size\ as \end{array}$$

Why?—Binary trees and $size$

$$\textbf{data}\ Bintree\ a \quad = \quad Empty \mid Node\ (Bintree\ a)\ a\ (Bintree\ a)$$

Here is a binary tree and its representation.



$$\Longrightarrow \quad \begin{array}{l} Node\ (Node\ Empty\ 2\ Empty) \\ \qquad 7 \\ \qquad (Node\ Empty \\ \qquad\qquad 19\ (Node\ Empty\ 65\ Empty)) \end{array}$$

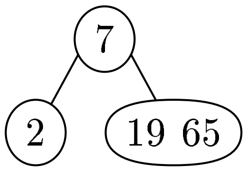
The function $size$ is defined inductively on the structure of $Bintree$.

$$\begin{array}{lcl} size & :: & Bintree\ a \to Int \\ size\ Empty & = & 0 \\ size\ (Node\ \ell\ a\ r) & = & size\ \ell + 1 + size\ r \end{array}$$

Why?—2-3 trees and *size*

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree23}\ a & = & \mathit{Empty} \\
 & | & \mathit{Node2}\ (\mathit{Tree23}\ a)\ a\ (\mathit{Tree23}\ a) \\
 & | & \mathit{Node3}\ (\mathit{Tree23}\ a)\ a\ (\mathit{Tree23}\ a)\ a\ (\mathit{Tree23}\ a)
\end{array}
$$

Here is a 2-3 tree and its representation.



$$
\Longrightarrow \quad \begin{array}{l}
\mathit{Node2}\ (\mathit{Node2}\ \mathit{Empty}\ 2\ \mathit{Empty}) \\
\qquad 7 \\
\qquad (\mathit{Node3}\ \mathit{Empty}\ 19\ \mathit{Empty}\ 65\ \mathit{Empty})
\end{array}
$$

The function *size* is defined inductively on the structure of *Tree23*.

$$
\begin{array}{lcl}
\mathit{size} & :: & \mathit{Tree23}\ a \rightarrow \mathit{Int} \\
\mathit{size}\ \mathit{Empty} & = & 0 \\
\mathit{size}\ (\mathit{Node2}\ t_1\ a_1\ t_2) & = & \mathit{size}\ t_1 + 1 + \mathit{size}\ t_2 \\
\mathit{size}\ (\mathit{Node3}\ t_1\ a_1\ t_2\ a_2\ t_3) & = & \mathit{size}\ t_1 + 1 + \mathit{size}\ t_2 + 1 + \mathit{size}\ t_3
\end{array}
$$

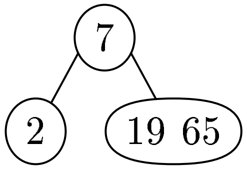
Why?—Nested 2-3 trees and $size$

So-called "nested datatypes" allow to capture the data-structural invariants of 2-3 trees.

$$
\begin{aligned}
&\textbf{type } \mathit{Tree23}\ a && = && \mathit{Perfect23}\ ()\ a \\
&\textbf{data } \mathit{Node23}\ t\ a && = && \mathit{Node2}\ t\ a\ t \mid \mathit{Node3}\ t\ a\ t\ a\ t \\
&\textbf{data } \mathit{Perfect23}\ t\ a && = && \mathit{Zero}\ t \mid \mathit{Succ}\ (\mathit{Perfect23}\ (\mathit{Node23}\ t\ a)\ a)
\end{aligned}
$$

NB The type recursion is *not regular*.

Here is a 2-3 tree and its representation.



$$
\Longrightarrow \quad
\begin{aligned}
\mathit{Succ}\ (\mathit{Succ}\ (\mathit{Zero}\ (\mathit{Node2}\ &(\mathit{Node2}\ ()\ 2\ ()) \\
&7 \\
&(\mathit{Node3}\ ()\ 19\ ()\ 65\ ())))))
\end{aligned}
$$

The definition of $size$ is not as easy as before . . .

Why?—Nested 2-3 trees and $size$

Again, the definitions follow the structure of types.

$$
\begin{array}{lcl}
size & :: & Tree23\ a \to Int \\
size & = & sizeP\ (const\ 0) \\
sizeN & :: & (t \to Int) \to (Node23\ t\ a \to Int) \\
sizeN\ s\ (Node2\ t_1\ a_1\ t_2) & = & s\ t_1 + 1 + s\ t_2 \\
sizeN\ s\ (Node3\ t_1\ a_1\ t_2\ a_2\ t_3) & = & s\ t_1 + 1 + s\ t_2 + 1 + s\ t_3 \\
sizeP & :: & (t \to Int) \to (Perfect23\ t\ a \to Int) \\
sizeP\ s\ (Zero\ t) & = & s\ t \\
sizeP\ s\ (Succ\ t) & = & sizeP\ (sizeN\ s)\ t
\end{array}
$$

NB The recursion is *not regular* ($sizeP$ uses polymorphic recursion).

NB Our approach to polytypic programming handles nested datatypes.

Why?—Résumé

- Implementing functions like $size$ is in many cases tiresomely routine and in same cases a challenge.

- But note, the meaning of size is crystal clear: $size :: f\ \ a \rightarrow Int$ counts the number of values of type a in a given value of type $f\ \ a$.

A brief guide to the lecture

$\surd$ Why? (2 – 7)

☐ How? (9 – 17)

☐ Examples (19 – 26)

☐ Behind the scenes (28 – 36)

☐ Résumé

How?

A polytypic programming language enables the user to program $size$ once and for all times.

A polytypic function is defined by *induction on the structure of types*.

[By contrast, a monotypic function is typically defined by induction on the structure of values.]

Hence, we must first explicit the structure of types . . .

How?—Datatypes

Haskell's **data** construct combines several features in a single coherent form: sums, products, and recursion.

The structure of datatypes becomes more apparent if we omit constructor names and use '$+$' for sums and '$\times$' for products.

$$
\boxed{
\begin{array}{lcl}
\mathit{Bintree}\ a & = & 1 + \mathit{Bintree}\ a \times a \times \mathit{Bintree}\ a \\
\mathit{Node23}\ a_1\ a_2 & = & a_1 \times a_2 \times a_1 + a_1 \times a_2 \times a_1 \times a_2 \times a_1 \\
\mathit{Perfect23}\ a_1\ a_2 & = & a_1 + \mathit{Perfect23}\ (\mathit{Node23}\ a_1\ a_2)\ a_2
\end{array}
}
$$

NB 1 denotes the unit type.

How?—Modeling datatypes

Given a fixed set of type variables $A = \{a_1, a_2, a_3, \ldots\}$ and a set of primitive type constructors $P = \{1, Int, +, \times\}$ *type expressions* are formed according to

$$T \quad = \quad A \mid P\;(T, \ldots, T) \; .$$

Notation: we write $c\,()$ as c and $t_1 \oplus t_2$ for $\oplus\,(t_1, t_2)$.

We treat 1, '$+$', and '$\times$' as if they were given by the following datatype declarations.

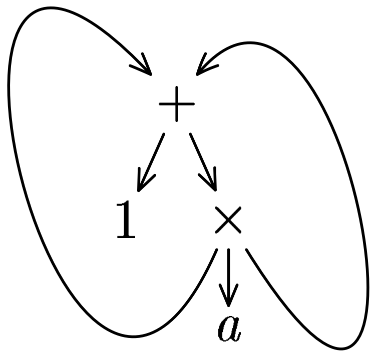
$$\begin{array}{lcl}
\mathbf{data}\; 1 & = & () \\
\mathbf{data}\; a_1 + a_2 & = & inl\;\; a_1 \mid inr\;\; a_2 \\
\mathbf{data}\; a_1 \times a_2 & = & (a_1, a_2)
\end{array}$$

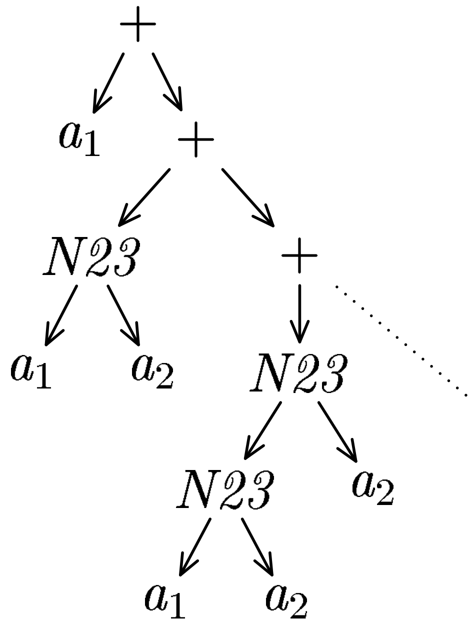
How?—Recursive types

What about recursive types?

Well, recursive types are modeled by infinite type expressions!

$Bintree\ a$





$Perfect23\ a_1\ a_2$

NB $Bintree\ a$ is a *rational tree*, $Perfect23\ a_1\ a_2$ an *algebraic tree*.

How?—Polytypic $size$

Here is the generic definition of $size$—the type parameter is enclosed in angle brackets.

$$
\begin{array}{lcl}
size\langle f\rangle & :: & \forall a.f\ a \to Int \\
size\langle a\rangle\ a & = & 1 \\
size\langle 1\rangle\ () & = & 0 \\
size\langle Int\rangle\ n & = & 0 \\
size\langle f_1 + f_2\rangle\ (inl\ a_1) & = & size\langle f_1\rangle\ a_1 \\
size\langle f_1 + f_2\rangle\ (inr\ a_2) & = & size\langle f_2\rangle\ a_2 \\
size\langle f_1 \times f_2\rangle\ (a_1, a_2) & = & size\langle f_1\rangle\ a_1 + size\langle f_2\rangle\ a_2
\end{array}
$$

NB Each equation is more or less inevitable.

How?—General combining forms

Here are some general combining forms which capture common programming idioms.

$$
\begin{array}{lcl}
id\ a & = & a \\
const\ a\ b & = & a \\
(f_1 \bigtriangledown f_2)\ (inl\ a_1) & = & f_1\ a_1 \\
(f_1 \bigtriangledown f_2)\ (inr\ a_2) & = & f_2\ a_2 \\
f_1 + f_2 & = & (inl \circ f_1) \bigtriangledown (inr \circ f_2) \\
outl\ (a_1, a_2) & = & a_1 \\
outr\ (a_1, a_2) & = & a_2 \\
(f_1 \bigtriangleup f_2)\ a & = & (f_1\ a, f_2\ a) \\
f_1 \times f_2 & = & (f_1 \circ outl) \bigtriangleup (f_2 \circ outr)
\end{array}
$$

How?—Polytypic $size$, revisited

Using the combining forms $size$ can be written more succinctly as

$$
\boxed{
\begin{array}{lcl}
size\langle f\rangle & :: & \forall a.f\ \ a \to Int \\
size\langle a\rangle & = & const\ 1 \\
size\langle K\,t\rangle & = & const\ 0 \\
size\langle f_1 + f_2\rangle & = & size\langle f_1\rangle \ \triangledown\ size\langle f_2\rangle \\
size\langle f_1 \times f_2\rangle & = & size\langle f_1\rangle \boxplus size\langle f_2\rangle \\
(\boxplus) & :: & (a_1 \to Int) \times (a_2 \to Int) \to (a_1 \times a_2 \to Int) \\
\varphi_1 \boxplus \varphi_2 & = & (+) \circ (\varphi_1 \times \varphi_2) \ \ .
\end{array}
}
$$

The first definition of $size\langle f\rangle$ is written in an *applicative style*, this one uses a *compositional style*.

NB The “type pattern” $K\,t$ matches all primitive types (ie 1, Int).

How?—Polytypic definitions

In general, a polytypic function is given by the following data.

$$\boxed{\begin{array}{rcl} poly\langle f\rangle & :: & \tau(f) \\ poly\langle a_i\rangle & = & poly_{a_i} \\ poly\langle F(f_1,\ldots,f_k)\rangle & = & poly_F\ (poly\langle f_1\rangle,\ldots,poly\langle f_k\rangle) \end{array}}$$

We have one equation for each type variable $(a_1, a_2, \ldots, a_n)$ and one equation for each primitive type constructor $(1, Int, +, \times)$.

How?—Résumé

- Type expressions are made up of type variables $(a_1, a_2, \ldots)$ and primitive type constructors $(1, Int, +, \times)$. Recursive types are modeled by infinite type expressions.

- A polytypic function is defined by induction on the structure of type expressions. An equation must be provided for each type variable $(a_1, a_2, \ldots, a_n)$ and for each primitive type constructor $(1, Int, +, \times)$.

A brief guide to the lecture

$\surd$ Why? (2 – 7)

$\surd$ How? (9 – 17)

$\square$ Examples (19 – 26)

$\square$ Behind the scenes (28 – 36)

$\square$ Résumé

Examples—Polytypic map

The function map takes a function of type $a \to b$ and a data structure of type $f\ a$ and applies the function to each element of type a within the data structure (while preserving its shape).

Here is $List$'s map.

$$
\boxed{
\begin{array}{lcl}
map & :: & (a \to b) \to (List\ a \to List\ b) \\
map\ \varphi\ Nil & = & Nil \\
map\ \varphi\ (Cons\ a\ as) & = & Cons\ (\varphi\ a)\ (map\ \varphi\ as)
\end{array}
}
$$

Here is $Bintree$'s map.

$$
\boxed{
\begin{array}{lcl}
map & :: & (a \to b) \to (Bintree\ a \to Bintree\ b) \\
map\ \varphi\ Empty & = & Empty \\
map\ \varphi\ (Node\ \ell\ a\ r) & = & Node\ (map\ \varphi\ \ell)\ (\varphi\ a)\ (map\ \varphi\ r)
\end{array}
}
$$

Examples—Polytypic map

Here is the polytypic definition.

$$
\begin{array}{lcl}
map\langle f \rangle & :: & \forall a.\forall b.(a \to b) \to (f\ \ a \to f\ \ b) \\
map\langle a \rangle\ \varphi & = & \varphi \\
map\langle K\, t \rangle\ \varphi & = & id \\
map\langle f_1 + f_2 \rangle\ \varphi & = & map\langle f_1 \rangle\ \varphi + map\langle f_2 \rangle\ \varphi \\
map\langle f_1 \times f_2 \rangle\ \varphi & = & map\langle f_1 \rangle\ \varphi \times map\langle f_2 \rangle\ \varphi
\end{array}
$$

NB map is the action of a functor on arrows (ie functions).

Examples—Polytypic reduction

The function $size$ is an instance of a more general concept termed *reduction*. A reduction is a function of type $f\ a \to a$. To define a reduction we require two ingredients: a value $e :: a$ and a binary operation $op :: a \times a \to a$.

Here is $List$'s $reduce$.

$$
\begin{array}{lcl}
reduce & :: & a \to (a \times a \to a) \to (List\ a \to a) \\
reduce\ e\ op\ Nil & = & e \\
reduce\ e\ op\ (Cons\ a\ as) & = & op\ (a, reduce\ e\ op\ as)
\end{array}
$$

Here is $Bintree$'s $reduce$.

$$
\begin{array}{lcl}
reduce & :: & a \to (a \times a \to a) \to (Bintree\ a \to a) \\
reduce\ e\ op\ Empty & = & e \\
reduce\ e\ op\ (Node\ \ell\ a\ r) & = & op\ (reduce\ e\ op\ \ell, op\ (a, reduce\ e\ op\ r))
\end{array}
$$

Examples—Polytypic reduction

Here is the polytypic definition.

$$
\begin{array}{lll}
\mathit{reduce}\langle f\rangle & :: & \forall a . a \to (a \times a \to a) \to (f\ a \to a) \\
\mathit{reduce}\langle f\rangle\ e\ \mathit{op} & = & \mathit{red}\langle f\rangle \\
\quad \mathbf{where}\ \mathit{red}\langle f\rangle & :: & f\ a \to a \\
\qquad\qquad \mathit{red}\langle a\rangle & = & \mathit{id} \\
\qquad\qquad \mathit{red}\langle K\,t\rangle & = & \mathit{const}\ e \\
\qquad\qquad \mathit{red}\langle f_1 + f_2\rangle & = & \mathit{red}\langle f_1\rangle \triangledown \mathit{red}\langle f_2\rangle \\
\qquad\qquad \mathit{red}\langle f_1 \times f_2\rangle & = & \mathit{op} \circ (\mathit{red}\langle f_1\rangle \times \mathit{red}\langle f_2\rangle)
\end{array}
$$

Examples—$reduce\langle f \rangle$ applied

A number of functions can be defined in terms of $reduce\langle f \rangle$ and $map\langle f \rangle$.

$$
\begin{array}{lcl}
sum\langle f\rangle & :: & (Num\ n) \Rightarrow f\ n \to n \\
sum\langle f\rangle & = & reduce\langle f\rangle\ 0\ (+) \\
and\langle f\rangle & :: & f\ Bool \to Bool \\
and\langle f\rangle & = & reduce\langle f\rangle\ True\ (\wedge) \\
minimum\langle f\rangle & :: & (Bounded\ a, Ord\ a) \Rightarrow f\ a \to a \\
minimum\langle f\rangle & = & reduce\langle f\rangle\ maxBound\ min \\
size\langle f\rangle & :: & (Num\ n) \Rightarrow f\ a \to n \\
size\langle f\rangle & = & sum\langle f\rangle \circ map\langle f\rangle\ (const\ 1) \\
all\langle f\rangle & :: & (a \to Bool) \to (f\ a \to Bool) \\
all\langle f\rangle\ p & = & and\langle f\rangle \circ map\langle f\rangle\ p \\
flatten\langle f\rangle & :: & f\ a \to List\ a \\
flatten\langle f\rangle & = & reduce\langle f\rangle\ [\,]\ (+\!\!+) \circ map\langle f\rangle\ wrap\ \mathbf{where}\ wrap\ a = [\,a\,]
\end{array}
$$

Examples—Polytypic equality

In Haskell $(==)$ can be automatically derived for each user-defined datatype. In other words, equality is a polytypic function.

$$
\begin{array}{lcl}
eq\langle t\rangle & :: & t \times t \to Bool \\
eq\langle 1\rangle\ ((),()) & = & True \\
eq\langle Int\rangle\ (m,n) & = & eqInt\ (m,n) \\
eq\langle t_1 + t_2\rangle\ (inl\ a_1, inl\ b_1) & = & eq\langle t_1\rangle\ (a_1,b_1) \\
eq\langle t_1 + t_2\rangle\ (inl\ a_1, inr\ b_2) & = & False \\
eq\langle t_1 + t_2\rangle\ (inr\ a_2, inl\ b_1) & = & False \\
eq\langle t_1 + t_2\rangle\ (inr\ a_2, inr\ b_2) & = & eq\langle t_2\rangle\ (a_2,b_2) \\
eq\langle t_1 \times t_2\rangle\ ((a_1,a_2),(b_1,b_2)) & = & eq\langle t_1\rangle\ (a_1,b_1) \wedge eq\langle t_2\rangle\ (a_2,b_2)
\end{array}
$$

Examples—Binary encoding

The function $encode\langle t\rangle$ encodes a value of type t into a sequence of bits.

$$
\begin{array}{lcl}
\mathbf{data}\ Bit & = & O \mid I \\
encode\langle t\rangle & :: & t \to [Bit] \\
encode\langle 1\rangle\ () & = & [\,] \\
encode\langle Int\rangle\ n & = & encodeInt\ n \\
encode\langle t_1 + t_2\rangle\ (inl\ a_1) & = & O : encode\langle t_1\rangle\ a_1 \\
encode\langle t_1 + t_2\rangle\ (inr\ a_2) & = & I : encode\langle t_2\rangle\ a_2 \\
encode\langle t_1 \times t_2\rangle\ (a_1, a_2) & = & encode\langle t_1\rangle\ a_1 +\!\!+\ encode\langle t_2\rangle\ a_2
\end{array}
$$

NB $encode\langle t\rangle$ can be seen as a very simple *pretty-printer*.

Examples—Binary decoding

Here is $encode\langle t\rangle$'s inverse.

$$
\begin{array}{lcl}
decode\langle t\rangle & :: & [Bit] \to (t, [Bit]) \\
decode\langle 1\rangle\ bs & = & ((), bs) \\
decode\langle Int\rangle\ bs & = & decodeInt\ bs \\
decode\langle t_1 + t_2\rangle\ (O : bs) & = & inl\ (decode\langle t_1\rangle\ bs) \\
decode\langle t_1 + t_2\rangle\ (I : bs) & = & inr\ (decode\langle t_2\rangle\ bs) \\
decode\langle t_1 \times t_2\rangle\ bs & = & \mathbf{let}\ (a_1, bs_1) = decode\langle t_1\rangle\ bs \\
 & & \phantom{\mathbf{let}}\ (a_2, bs_2) = decode\langle t_2\rangle\ bs_1 \\
 & & \mathbf{in}\ ((a_1, a_2), bs_2)
\end{array}
$$

NB $decode\langle t\rangle$ can be seen as a very simple *parser*.

A brief guide to the lecture

$\surd$ Why? (2 – 7)

$\surd$ How? (9 – 17)

$\surd$ Examples (19 – 26)

□ Behind the scenes (28 – 36)

□ Résumé

Behind the scenes

By now we have seen several examples for polytypic functions.

It remains to explain how a polytypic function is specialized.

Behind the scenes—Datatypes and datatype definitions

The specialization must be based on the *representation* of type expressions.

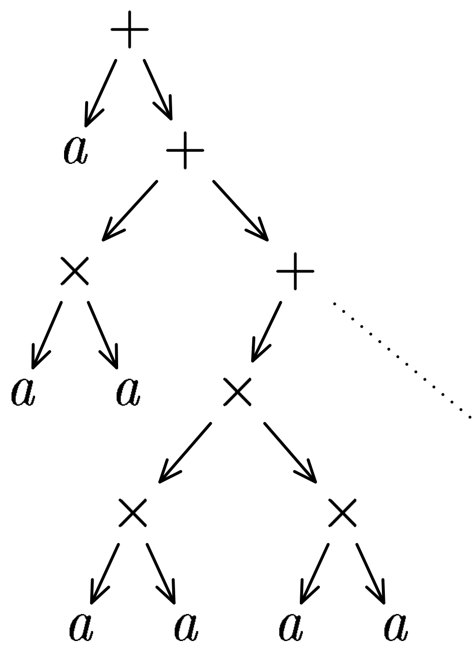
The datatype definition

$$\begin{array}{lcl} \mathbf{data}\ \mathit{Perfect}\ a & = & \mathit{Zero}\ a \\ & \mid & \mathit{Succ}\ (\mathit{Perfect}\ (a, a)) \end{array}$$

alias

$$\mathit{Perfect}\ a \quad = \quad a + \mathit{Perfect}\ (a \times a)$$

represents the type expression on the right.



Behind the scenes—Functor expressions

The structure of datatype definitions becomes more apparent if we rewrite them as *functor equations*. *Functor expressions* of arity n are given by

$$\boxed{\quad F^n \quad = \quad \Pi^n_i \mid P^n \mid F^k \cdot (F^n_1, \ldots, F^n_k) \ . \quad}$$

Here, Π^n_i is the n-ary projection functor selecting its i-th component; $F \cdot (F_1, \ldots, F_k)$ is the composition of a k-ary functor F with functors F_i, all of arity n.

Notation: we write $K t$ for $t \cdot ()$ and $f_1 \oplus f_2$ for $\oplus \cdot (f_1, f_2)$. For unary and binary projection functors we use the following more familiar names: $Id = \Pi^1_1$, $Fst = \Pi^2_1$, and $Snd = \Pi^2_2$.

Behind the scenes—Functor equations

Here are the datatype definitions rewritten as functor equations.

$$\begin{array}{lcl}
\mathit{Bintree} & = & K1 + \mathit{Bintree} \times \mathit{Id} \times \mathit{Bintree} \\
\mathit{Node23} & = & \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} + \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} \\
\mathit{Perfect23} & = & \mathit{Fst} + \mathit{Perfect23} \cdot (\mathit{Node23}\,, \mathit{Snd}) \\
\mathit{Perfect} & = & \mathit{Id} + \mathit{Perfect} \cdot (\mathit{Id} \times \mathit{Id})
\end{array}$$

Functor equations are written in a *compositional style* while **data** definitions are written in an *applicative style*.

Behind the scenes—Specializing polytypic functions

The central idea is to define, for each arity n, an n-ary function $poly_n\langle f\rangle$ satisfying

$$\boxed{poly_n\langle f\rangle\ (poly\langle t_1\rangle, \ldots, poly\langle t_n\rangle) \quad = \quad poly\langle f(t_1, \ldots, t_n)\rangle\ .}$$

Note the close correspondence between functors and functions:

- f is an n-ary functor sending t_1, . . . , t_n to $f(t_1, \ldots, t_n)$;
- $poly_n\langle f\rangle$ is an n-ary function sending $poly\langle t_1\rangle$, . . . , $poly\langle t_n\rangle$ to $poly\langle f(t_1, \ldots, t_n)\rangle$.

Behind the scenes—Specializing polytypic functions

The following definition satisfies the specification.

$$
\boxed{
\begin{array}{rcl}
poly\langle f\rangle & :: & \tau(f) \\
poly\langle f\rangle & = & poly_n\langle f\rangle\ (poly_{a_1}, \ldots, poly_{a_n}) \\
poly_n\langle f\rangle & :: & (\tau(t_1), \ldots, \tau(t_n)) \to \tau(f(t_1, \ldots, t_n)) \\
poly_n\langle \Pi_i\rangle & = & \pi_i \\
poly_n\langle F\rangle & = & poly_F \\
poly_n\langle f \cdot (h_1, \ldots, h_k)\rangle & = & poly_k\langle f\rangle \star (poly_n\langle h_1\rangle, \ldots, poly_n\langle h_k\rangle)\ ,
\end{array}
}
$$

where $\pi_i\,(\varphi_1, \ldots, \varphi_n) = \varphi_i$ and $(\varphi \star (\varphi_1, \ldots, \varphi_n))\,a = \varphi\,(\varphi_1\ a, \ldots, \varphi_n\ a)$.

NB $poly_n$ is inductive on the structure of functor expressions.

Behind the scenes—Specializing polytypic functions

Specializing a polytypic function works by transforming a system of functor equations into a system of function definitions.

$$
\begin{array}{lcl}
f_1 & = & e_1 \\
f_2 & = & e_2 \\
\vdots & & \\
f_m & = & e_m
\end{array}
\quad \Longrightarrow \quad
\begin{array}{lcl}
\mathit{poly}_{k_1}\langle f_1 \rangle & = & \mathit{poly}_{k_1}\langle e_1 \rangle \\
\mathit{poly}_{k_2}\langle f_2 \rangle & = & \mathit{poly}_{k_2}\langle e_2 \rangle \\
\vdots & & \\
\mathit{poly}_{k_m}\langle f_m \rangle & = & \mathit{poly}_{k_m}\langle e_m \rangle
\end{array}
$$

Behind the scenes—Specializing *size*

$$
\begin{aligned}
\mathit{Tree23} &= \mathit{Perfect23} \cdot (K1, \mathit{Id}) \\
\mathit{Node23} &= \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} + \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} \times \mathit{Snd} \times \mathit{Fst} \\
\mathit{Perfect23} &= \mathit{Fst} + \mathit{Perfect23} \cdot (\mathit{Node23}, \mathit{Snd})
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
\mathit{size}\langle \mathit{Tree23} \rangle &= \mathit{size}_1\langle \mathit{Tree23} \rangle\ (\mathit{const}\ 1) \\
\mathit{size}_1\langle \mathit{Tree23} \rangle\ \varphi &= \mathit{size}_2\langle \mathit{Perfect23} \rangle\ (\mathit{const}\ 0, \varphi) \\
\mathit{size}_2\langle \mathit{Node23} \rangle\ (\varphi_1, \varphi_2) &= (\varphi_1 \boxplus \varphi_2 \boxplus \varphi_1) \triangledown (\varphi_1 \boxplus \varphi_2 \boxplus \varphi_1 \boxplus \varphi_2 \boxplus \varphi_1) \\
\mathit{size}_2\langle \mathit{Perfect23} \rangle\ (\varphi_1, \varphi_2) &= \varphi_1 \triangledown \mathit{size}_2\langle \mathit{Perfect23} \rangle\ (\mathit{size}_2\langle \mathit{Node23} \rangle\ (\varphi_1, \varphi_2), \varphi_2)
\end{aligned}
$$

Behind the scenes—Résumé

- Type expressions are represented by functor equations.
- The specialization of a polytypic function is defined on the representation of type expressions, ie it is inductive on the structure of functor expressions.

A brief guide to the lecture

$\surd$ Why? (2 – 7)

$\surd$ How? (9 – 17)

$\surd$ Examples (19 – 26)

$\surd$ Behind the scenes (28 – 36)

□ Résumé

Résumé

- A polytypic function is defined by *induction on the structure of types*.
- Types are modeled by possibly infinite trees.
- The specialization of a polytypic function is defined on the representation of types, ie it is inductive on the structure of functor expressions.
- Polytypic functions are generally more abstract than their monotypic counterparts. But their definition is also to a high degree inevitable.

References

Polytypic programming.

- P. Jansson and J. Jeuring. PolyP—a polytypic programming language extension. *POPL'97*. ACM Press.
- J. Jeuring and P. Jansson. Polytypic programming. *2nd International School on Advanced Functional Programming*, LNCS 1129.
- R. Hinze. Polytypic functions over nested datatypes. *3rd latin-american conference on functional programming (CLaPF'99)*.
- R. Hinze. Generalizing generalized tries. Tech. rept. IAI-TR-98-11.
- R. Hinze. Polytypic programming with ease. Tech. rept. IAI-TR-99-2.

Nested datatypes.

- R. Bird and L. Meertens. Nested datatypes. *MPC'98*, LNCS 1422.
- R. Bird and R. Paterson. Generalised folds for nested datatypes, 1998.

Prospects

- Polytypic reasoning.
- Making type recursion explicit.
- Polytypic datatypes.