POLYTYPIC PROGRAMMING WITH EASE
(Extended Abstract)

RALF HINZE

Institut für Informatik III, Universität Bonn, Römerstraße 164
53117 Bonn, Germany
Email: `ralf@informatik.uni-bonn.de`
Hompage: `http://www.informatik.uni-bonn.de/~ralf`

November, 1999

What is a polytypic function?

A polytypic function is parameterised by datatype and is defined by induction on the structure of types.

The archetypical example:

$$\boxed{size\langle f\rangle \quad :: \quad \forall a.f \ \ a \to Int \quad .}$$

Meaning: $size\langle f\rangle$ counts the number of values of type a in a given value of type $f \ \ a$.

Example types—for which we may want to define $size$

Lists and rose trees (multiway branching trees).

$$
\begin{array}{lcl}
\mathbf{data}\ List\ a & = & Nil \mid Cons\ a\ (List\ a) \\
\mathbf{data}\ Rose\ a & = & Branch\ a\ (List\ (Rose\ a))
\end{array}
$$

Perfectly balanced, binary leaf trees.

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

NB. $Perfect$ is a so-called *nested datatype* (while $List$, $Rose$, and $Fork$ are *regular* datatypes).

How is a polytypic function defined?

Well, we must explicit the structure of types . . .

The central question: How do we model *recursive types*?

PolyP—Modeling types

In the framework of PolyP (P. Jansson and J. Jeuring) recursive types are modeled by so-called *type functors*.

$$
\begin{array}{lcl}
List & = & \mu\ BaseList \\
BaseList & = & K1 + Fst \times Snd \\
Rose & = & \mu\ BaseRose \\
BaseRose & = & Fst \times List \cdot Snd
\end{array}
$$

The structure of functors is given by the following grammar.

$$
\begin{array}{lcl}
F & = & \mu B \\
B & = & Fst \mid Snd \mid K1 \mid B + B \mid B \times B \mid F \cdot B
\end{array}
$$

PolyP—Polytypic definitions

Here is the definition of $size\langle f\rangle$ in PolyP (modulo change of notation).

$$
\begin{array}{lcl}
size\langle f\rangle & :: & f\ a \to Int \\
size\langle f\rangle & = & sum\langle f\rangle \circ map\langle f\rangle\ (const\ 1) \\
sum\langle f\rangle & :: & f\ Int \to Int \\
sum\langle \mu b\rangle & = & cata\langle \mu b\rangle\ (bsum\langle b\rangle) \\
bsum\langle b\rangle & :: & b\ Int\ Int \to Int \\
bsum\langle Fst\rangle & = & id \\
bsum\langle Snd\rangle & = & id \\
bsum\langle K1\rangle & = & const\ 0 \\
bsum\langle b_1 + b_2\rangle & = & bsum\langle b_1\rangle \bigtriangledown bsum\langle b_2\rangle \\
bsum\langle b_1 \times b_2\rangle & = & (+) \circ (bsum\langle b_1\rangle \times bsum\langle b_2\rangle) \\
bsum\langle f \cdot b\rangle & = & sum\langle f\rangle \circ map\langle f\rangle\ (bsum\langle b\rangle)
\end{array}
$$

Our approach—Modeling types

In our framework recursive types are modeled by *infinite* functor expressions (so-called *algebraic trees*).

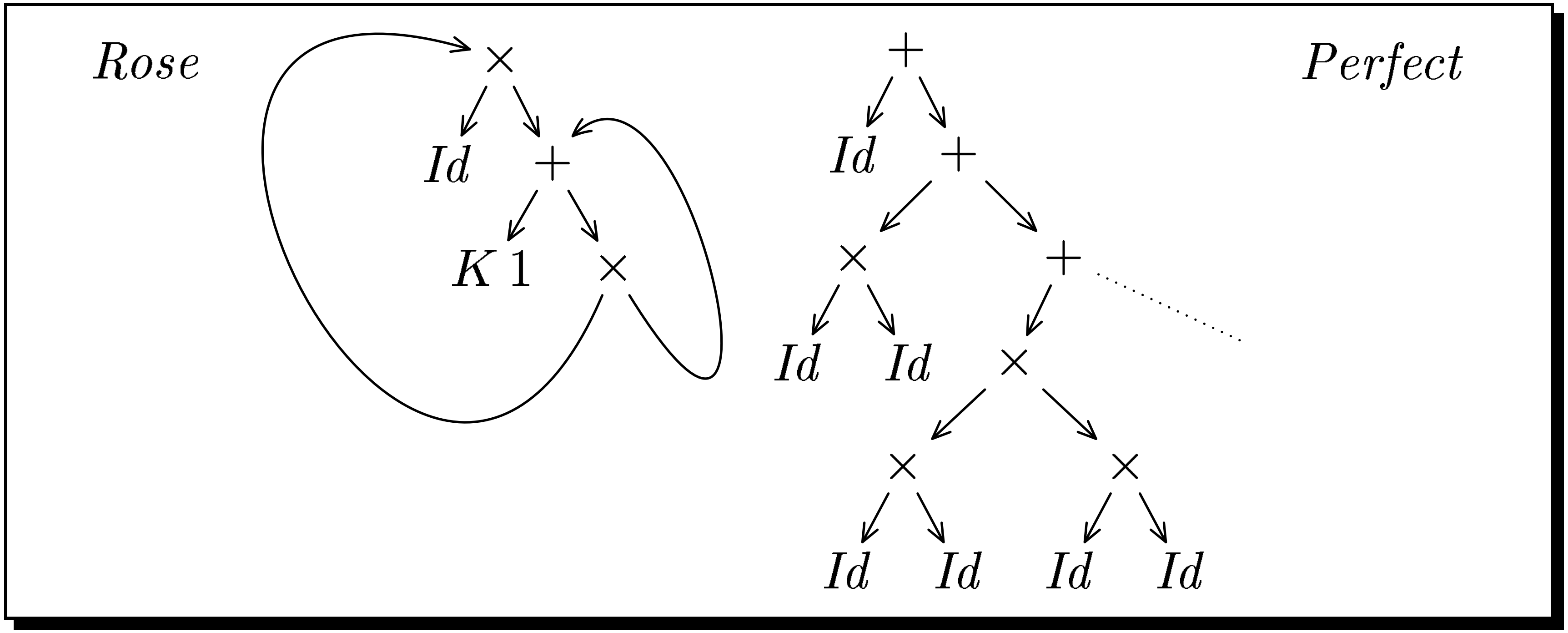
$$
\begin{array}{lcl}
\mathit{List} & = & K1 + \mathit{Id} \times \mathit{List} \\
\mathit{Rose} & = & \mathit{Id} \times \mathit{List} \cdot \mathit{Rose} \\
\mathit{Perfect} & = & \mathit{Id} + \mathit{Perfect} \cdot \mathit{Fork} \\
\mathit{Fork} & = & \mathit{Id} \times \mathit{Id}
\end{array}
$$

The structure of functors is now given by the following grammar.

$$
F \quad = \quad \mathit{Id} \mid K1 \mid F + F \mid F \times F
$$

Our approach—Recursive types

Regular datatypes like $Rose$ are modeled by *rational trees*; nested datatypes like $Perfect$ by *algebraic trees*.



Our approach—Polytypic definitions

Here is the definition of $size\langle f \rangle$ in our framework.

$$
\begin{array}{lcl}
size\langle f \rangle & :: & \forall a.f\ \ a \to Int \\
size\langle Id \rangle & = & const\ 1 \\
size\langle K1 \rangle & = & const\ 0 \\
size\langle f_1 + f_2 \rangle & = & size\langle f_1 \rangle \ \triangledown\ size\langle f_2 \rangle \\
size\langle f_1 \times f_2 \rangle & = & size\langle f_1 \rangle \boxplus size\langle f_2 \rangle \\
\varphi_1 \boxplus \varphi_2 & = & (+) \circ (\varphi_1 \times \varphi_2)
\end{array}
$$

A standard result from the theory of infinite trees guarantees that $size$ possesses a unique least extension in the realm of infinite trees.

Our approach—Polytypic definitions

NB. $size\langle f\rangle$ can also be defined in a pointwise style.

$$
\begin{array}{lcl}
size\langle f\rangle & :: & \forall a . f\ a \to Int \\
size\langle Id\rangle\ x & = & 1 \\
size\langle K1\rangle\ x & = & 0 \\
size\langle f_1 + f_2\rangle\ (Inl\ x_1) & = & size\langle f_1\rangle\ x_1 \\
size\langle f_1 + f_2\rangle\ (Inr\ x_2) & = & size\langle f_2\rangle\ x_2 \\
size\langle f_1 \times f_2\rangle\ (x_1, x_2) & = & size\langle f_1\rangle\ x_1 + size\langle f_2\rangle\ x_2
\end{array}
$$

Examples—Polytypic map

A generic definition of the mapping function (ie the functorial action on arrows).

$$
\begin{array}{lcl}
map\langle f\rangle & :: & \forall a.\forall b.(a \to b) \to (f\ \ a \to f\ \ b) \\
map\langle Id\rangle\ \varphi\ x & = & \varphi\ x \\
map\langle K1\rangle\ \varphi\ x & = & x \\
map\langle f_1 + f_2\rangle\ \varphi\ (Inl\ x_1) & = & Inl\ (map\langle f_1\rangle\ \varphi\ x_1) \\
map\langle f_1 + f_2\rangle\ \varphi\ (Inr\ x_2) & = & Inr\ (map\langle f_2\rangle\ \varphi\ x_2) \\
map\langle f_1 \times f_2\rangle\ \varphi\ (x_1, x_2) & = & (map\langle f_1\rangle\ \varphi\ x_1, map\langle f_2\rangle\ \varphi\ x_2)
\end{array}
$$

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 $op :: a \to a \to a$.

$$
\begin{array}{lcl}
reduce\langle f\rangle & :: & \forall a.a \to (a \to a \to a) \to (f\ a \to a) \\
reduce\langle Id\rangle\ e\ op\ x & = & x \\
reduce\langle K1\rangle\ e\ op\ x & = & e \\
reduce\langle f_1 + f_2\rangle\ e\ op\ (Inl\ x_1) & = & reduce\langle f_1\rangle\ e\ op\ x_1 \\
reduce\langle f_1 + f_2\rangle\ e\ op\ (Inr\ x_2) & = & reduce\langle f_2\rangle\ e\ op\ x_2 \\
reduce\langle f_1 \times f_2\rangle\ e\ op\ (x_1, x_2) & = & reduce\langle f_1\rangle\ e\ op\ x_1 \\
 & & \quad `op`\,reduce\langle f_2\rangle\ e\ op\ x_2
\end{array}
$$

The function op is inserted between adjacent elements of type a and non-a elements are replaced by e.

Examples—$reduce\langle f \rangle$ applied

$$
\begin{array}{lcl}
sum\langle f\rangle & :: & \forall n.(Num\ n) \Rightarrow f\ n \to n \\
sum\langle f\rangle & = & reduce\langle f\rangle\ 0\ (+) \\
and\langle f\rangle & :: & f\ Bool \to Bool \\
and\langle f\rangle & = & reduce\langle f\rangle\ True\ (\wedge) \\
size\langle f\rangle & :: & \forall a.\forall n.(Num\ n) \Rightarrow f\ a \to n \\
size\langle f\rangle & = & sum\langle f\rangle \circ map\langle f\rangle\ (const\ 1) \\
all\langle f\rangle & :: & \forall a.(a \to Bool) \to (f\ a \to Bool) \\
all\langle f\rangle\ p & = & and\langle f\rangle \circ map\langle f\rangle\ p \\
flatten\langle f\rangle & :: & \forall a.f\ a \to List\ a \\
flatten\langle f\rangle & = & reduce\langle f\rangle\ [\,]\ (+\!\!+) \circ map\langle f\rangle\ wrap\ \mathbf{where}\ wrap\ a = [\,a\,] \\
\mathbf{data}\ Tree\ a & = & Empty \mid Leaf\ a \mid Fork\ (Tree\ a)\ (Tree\ a) \\
shape\langle f\rangle & :: & \forall a.f\ a \to Tree\ a \\
shape\langle f\rangle & = & reduce\langle f\rangle\ Empty\ Fork \circ map\langle f\rangle\ Leaf
\end{array}
$$

Specialization—Basic idea

The specialization should be *compositional*. Assume that we require $size$ functions for the following types

$$List \qquad\qquad List \cdot Rose \quad .$$

The code generated for $List$ should be shared.

Idea: Let function follow type.

$$size_1\langle List\rangle \qquad\qquad size_1\langle List\rangle \circ size_1\langle Rose\rangle$$

Since $List$ is a function on types, $size_1\langle List\rangle$ is consequently a function on 'size functions': it maps $size\langle f\rangle$ to $size\langle List \cdot f\rangle$. We have, in particular, $size\langle List\rangle = size_1\langle List\rangle\ (size\langle Id\rangle)$.

Specialization—Generating code

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

$$\begin{array}{rcl} List\ a & = & 1 + a \times List\ a \\ Rose\ a & = & a \times List\ (Rose\ a) \end{array}$$

$$\Downarrow$$

$$\begin{array}{rcl} size_1\langle List\rangle\ \varphi & = & const\ 0 \bigtriangledown (\varphi \boxplus size_1\langle List\rangle\ \varphi) \\ size_1\langle Rose\rangle\ \varphi & = & \varphi \boxplus size_1\langle List\rangle\ (size_1\langle Rose\rangle\ \varphi) \end{array}$$

NB. The structure of types is reflected on the value level.

Specialization—Type signatures

Let $Size\langle f\rangle = \forall a.f\ \ a \to Int$, then $size_1\langle List\rangle$ and $size_1\langle Rose\rangle$ have the following types.

$$
\begin{array}{lcl}
size_1\langle List\rangle & :: & \forall f\,.Size\langle f\rangle \to Size\langle List \cdot f\rangle \\
size_1\langle List\rangle\ \varphi & = & const\ 0\ \triangledown\ (\varphi \boxplus size_1\langle List\rangle\ \varphi) \\
size_1\langle Rose\rangle & :: & \forall f\,.Size\langle f\rangle \to Size\langle Rose \cdot f\rangle \\
size_1\langle Rose\rangle\ \varphi & = & \varphi \boxplus size_1\langle List\rangle\ (size_1\langle Rose\rangle\ \varphi)
\end{array}
$$

Since $size_1\langle List\rangle$ takes polymorphic functions to polymorphic functions, it has a so-called *rank-2* type signature.

Specialization—$size$ in Haskell

Haskell 98 does not provide rank-2 type signatures. Furthermore, Haskell only offers a limited form of *type constructor polymorphism*. Both typing problems disappear if we replace functor variables by type variables.

$$
\begin{array}{lcl}
\mathit{size1List} & :: & (x \to \mathit{Int}) \to (\mathit{List}\ x \to \mathit{Int}) \\
\mathit{size1List}\ \varphi\ \mathit{Nil} & = & 0 \\
\mathit{size1List}\ \varphi\ (\mathit{Cons}\ x\ \mathit{xs}) & = & \varphi\ x + \mathit{size1List}\ \varphi\ \mathit{xs} \\
\mathit{size1Rose} & :: & (x \to \mathit{Int}) \to (\mathit{Rose}\ x \to \mathit{Int}) \\
\mathit{size1Rose}\ \varphi\ (\mathit{Branch}\ x\ \mathit{xs}) & = & \varphi\ x + \mathit{size1List}\ (\mathit{size1Rose}\ \varphi)\ \mathit{xs}
\end{array}
$$

NB. This *trick* works for arbitrary type signatures.

Specialization—Nested datatypes

The specialization works equally well for nested datatypes.

$$
\begin{aligned}
\mathit{Perfect}\ a &= a + \mathit{Perfect}\ (\mathit{Fork}\ a)\\
\mathit{Fork}\ a &= a \times a
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
\mathit{size}_1\langle \mathit{Perfect}\rangle\ \varphi &= \varphi \triangledown \mathit{size}_1\langle \mathit{Perfect}\rangle\ (\mathit{size}_1\langle \mathit{Fork}\rangle\ \varphi)\\
\mathit{size}_1\langle \mathit{Fork}\rangle\ \varphi &= \varphi \boxplus \varphi
\end{aligned}
$$

Specialization—Nested datatypes

The corresponding Haskell code.

$$
\begin{array}{lcl}
\mathit{size1Perfect} & :: & (x \to \mathit{Int}) \to (\mathit{Perfect}\ x \to \mathit{Int}) \\
\mathit{size1Perfect}\ \varphi\ (\mathit{Zero}\ x) & = & \varphi\ x \\
\mathit{size1Perfect}\ \varphi\ (\mathit{Succ}\ x) & = & \mathit{size1Perfect}\ (\mathit{size1Fork}\ \varphi)\ x \\
\mathit{size1Fork} & :: & (x \to \mathit{Int}) \to (\mathit{Fork}\ x \to \mathit{Int}) \\
\mathit{size1Fork}\ \varphi\ (\mathit{Fork}\ x_1\ x_2) & = & \varphi\ x_1 + \varphi\ x_2
\end{array}
$$

NB. $\mathit{size1Perfect}$ operates in two stages: while recursing it constructs a tailor-made size function $\mathit{size1Fork}^i\ \varphi$ of type $\mathit{Fork}^i\ x \to \mathit{Int}$, which is eventually applied in the base case.

Pros and Cons

PolyP

- $-$ The definition of polytypic functions is quite involved.
- $-$ The approach is restricted to regular unary functors.
- $-$ The extension to n-ary functors and mutually recursive datatypes is (at least) unwieldy.
- $+$ Since type recursion is explicit, cata- and anamorphisms can be defined.

Our approach

- $+$ The definition of polytypic functions is much simpler.
- $+$ The approach works for arbitrary unary functors (including nests).
- $+$ The extension to n-ary functors is entirely straightforward.
- $\pm$ Since type recursion is implicit, cata- and anamorphisms cannot be defined (requires a simple extension).

Current work

- We have seen *type-indexed values*. In "Generalizing Generalized Tries" (appears in JFP) the approach is applied to define *type-indexed datatypes*. We show how to define *tries* and operations on tries generically.

- Haskell 98 allows to define types that are parameterized by type constructors.

$$\boxed{\mathbf{data}\ GRose\ f\ a \quad = \quad Node\ a\ (f\ (GRose\ f\ a))}$$

In "A New Approach to Generic Functional Programming" (POPL'00) we show how to extend the approach to higher-order kinded types. *Basic idea:* datatypes are modeled by terms of the simply typed λ-calculus (with kinds playing the rôle of types).

Future work

Mapping functions cannot only be defined for unary functors, but also for binary functors, ternary functors, . . .

$$
\begin{array}{lll}
map\langle f :: * \to * \rangle & :: & (a \to b) \\
 & & \to (f\ a \to f\ b) \\
map\langle f :: * \to * \to * \rangle & :: & (a_1 \to b_1) \\
 & & \to (a_2 \to b_2) \\
 & & \to (f\ a_1\ a_2 \to f\ b_1\ b_2)
\end{array}
$$

Can we define a polytypic mapping function that works for *all types of all kinds*? It appears that such a function requires a *kind-indexed type*.