POLYTYPIC FUNCTIONS OVER NESTED DATATYPES

RALF HINZE

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

March, 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:

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

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

How is a polytypic function defined?

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

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

The semantic approach—Modeling types

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

$$\begin{array}{rcl} List\ a & = & Nil \mid Cons\ a\ (List\ a) \\ & \Downarrow & \\ List & = & \mu\ Base \\ Base & = & K1 + Fst \times Snd \end{array}$$

The structure of functors is given by the following grammar.

$$\begin{array}{rcl} F & = & \mu B \\ B & = & K\,T \mid Fst \mid Snd \mid B + B \mid B \times B \mid F \cdot B \end{array}$$

The semantic approach—Polytypic definitions

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

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

NB $sum\langle f \rangle$ and $bsum\langle f \rangle$ are inductive on the structure of functors.

The syntactic approach—Modeling types

In our framework recursive types are modeled by *infinite* functor expressions (rational trees).

$$
\begin{array}{rcl}
List\ a & = & Nil \mid Cons\ a\ (List\ a) \\
 & \Downarrow & \\
List & = & K1 + Id \times List
\end{array}
$$

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

$$
F \quad = \quad K\,T \mid Id \mid F + F \mid F \times F \mid F \cdot F
$$

The syntactic approach—Polytypic definitions

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

$$
\begin{array}{lcl}
size\langle f\rangle & :: & f\ a \to Int \\
size\langle f\rangle & = & sum\langle f\rangle \circ map\langle f\rangle\ (const\ 1) \\
sum\langle f\rangle & :: & f\ Int \to Int \\
sum\langle Id\rangle & = & id \\
sum\langle K\,t\rangle & = & const\ 0 \\
sum\langle f_1 + f_2\rangle & = & sum\langle f_1\rangle \triangledown sum\langle f_2\rangle \\
sum\langle f_1 \times f_2\rangle & = & sum\langle f_1\rangle \boxplus sum\langle f_2\rangle \\
sum\langle f \cdot g\rangle & = & sum\langle f\rangle \circ map\langle f\rangle\ (sum\langle g\rangle) \\
\varphi_1 \boxplus \varphi_2 & = & (+) \circ (\varphi_1 \times \varphi_2)
\end{array}
$$

NB $sum\langle f\rangle$ is inductive on the structure of functors.

Specialization—Mutually recursive types

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

$$
\begin{aligned}
\mathit{Rose} &= \mathit{Id} \times \mathit{Forest} \\
\mathit{Forest} &= \mathit{K1} + \mathit{Rose} \times \mathit{Forest}
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
\mathit{sum}\langle \mathit{Rose} \rangle &= \mathit{id} \boxplus \mathit{size}\langle \mathit{Forest} \rangle \\
\mathit{sum}\langle \mathit{Forest} \rangle &= \mathit{const}\ 0 \triangledown (\mathit{sum}\langle \mathit{Rose} \rangle \boxplus \mathit{sum}\langle \mathit{Forest} \rangle)
\end{aligned}
$$

Specialization—Nested datatypes

$$\begin{aligned}
\Delta &= Id \times Id \\
Perfect &= Id + Perfect \cdot \Delta
\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}
sum\langle\Delta\rangle &= id \boxplus id \\
sum\langle Perfect\rangle &= id \triangledown (sum\langle Perfect\rangle \circ map\langle Perfect\rangle\ (sum\langle\Delta\rangle))
\end{aligned}$$

For nested datatypes efficiency becomes a concern: for some functors $sum\langle f\rangle$ has a quadratic running time!

Efficiency

Specification:

$$summap\langle f\rangle\ \varphi \quad = \quad sum\langle f\rangle \circ map\langle f\rangle\ \varphi \ .$$

Implementation:

$$\begin{array}{lcl}
summap\langle f\rangle & :: & (a \to Int) \to (f\ \ a \to Int) \\
summap\langle Id\rangle\ \varphi & = & \varphi \\
summap\langle K\,t\rangle\ \varphi & = & const\ 0 \\
summap\langle f_1 + f_2\rangle\ \varphi & = & summap\langle f_1\rangle\ \varphi \triangledown summap\langle f_2\rangle\ \varphi \\
summap\langle f_1 \times f_2\rangle\ \varphi & = & summap\langle f_1\rangle\ \varphi \boxplus summap\langle f_2\rangle\ \varphi \\
summap\langle f \cdot g\rangle\ \varphi & = & summap\langle f\rangle\ (summap\langle g\rangle\ \varphi)\ .
\end{array}$$

NB $summap\langle f\rangle$ has a linear running time.

Efficiency—Specialization

$$\begin{aligned}
\Delta &= Id \times Id \\
Perfect &= Id + Perfect \cdot \Delta
\end{aligned}$$

$$\Downarrow$$

$$\begin{aligned}
summap\langle \Delta \rangle\ \varphi &= \varphi \boxplus \varphi \\
summap\langle Perfect \rangle\ \varphi &= \varphi \triangledown summap\langle Perfect \rangle\ (summap\langle \Delta \rangle\ \varphi)
\end{aligned}$$

NB $sum\langle f \rangle$ proceeds top-down, $summap\langle f \rangle$ proceeds bottom-up.

Pros and Cons

The semantic approach

- $-$ 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.

The syntactic approach

- $+$ The definition of polytypic functions is simpler.
- $+$ The approach works for arbitrary unary functors.
- $\pm$ The extension to n-ary functors is straightforward (except for functor composition, $F \cdot (F_1, \ldots, F_n)$).
- $\pm$ Since type recursion is implicit, cata- and anamorphisms cannot be defined (requires a simple extension).

Current work

The syntactic approach becomes even simpler if recursive types are modeled by *algebraic trees* (instead of *rational trees*).

Main difference: The user need not supply equations for functor composition.

$$
\begin{array}{lcl}
\mathit{size}\langle f\rangle & :: & f\ a \to \mathit{Int} \\
\mathit{size}\langle \mathit{Id}\rangle & = & \mathit{const}\ 1 \\
\mathit{size}\langle K\, t\rangle & = & \mathit{const}\ 0 \\
\mathit{size}\langle f_1 + f_2\rangle & = & \mathit{size}\langle f_1\rangle \triangledown \mathit{size}\langle f_2\rangle \\
\mathit{size}\langle f_1 \times f_2\rangle & = & \mathit{size}\langle f_1\rangle \boxplus \mathit{size}\langle f_2\rangle
\end{array}
$$

NB $\mathit{size}\langle f\rangle$ need not be defined in terms of $\mathit{sum}\langle f\rangle$.

Future work

Extension of the approach to higher-order polymorphic datatypes.

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

Basic idea: higher-order polymorphic datatypes are modeled by possibly infinite expressions of the simply typed λ-calculus (with kinds playing the rôle of types).