POLYTYPIC PROGRAMMING—OR: Programming Language Theory is Helpful

RALF HINZE

Institute of Information and Computing Sciences
Utrecht University
Email: `ralf@cs.uu.nl`
Homepage: `http://www.cs.uu.nl/~ralf/`

March, 2001

(Pick the slides at `.../~ralf/talks.html#T23`.)

A brief outline of the talk

Polytypic programming is about making programming easier and more economical.

- ✘ Why? (2 – 7)
- ✘ How?—A programmer's perspective (9 – 13)
- ✘ How?—From an implementor's point of view (15 – 26)

Prerequisites: we use the functional programming language Haskell 98 for the examples.

Recap : Haskell 98

Haskell 98 is a statically typed language. New types are introduced via **data** declarations.

$$\mathbf{data}\ List\ A \quad = \quad nil \mid cons\ A\ (List\ A)$$

Here, $List$ is a *type constructor*, nil and $cons$ are *data constructors*.

Functions are defined via pattern matching.

$$\begin{array}{lcl} size & :: & \forall A\,.\,List\ A \to Int \\ size\ nil & = & 0 \\ size\ (cons\ a\ as) & = & 1 + size\ as \end{array}$$

☞ $size$ is a *polymorphic* function.

Polymorphic type systems

Many of us (most of us?) will probably agree that *static type systems*, especially, *polymorphic type systems* are a good thing. For reasons of

Security soundness results guarantee that well-typed programs cannot 'go wrong'.

Flexibility polymorphism allows the definition of functions that behave uniformly over all types (such as $size$).

☞ This is an uninteresting talk for addicts of untyped languages.

However, . . .

. . . even polymorphic type systems are sometimes less flexible than one would wish.

For instance, it is not possible to define a polymorphic *equality function*.

$$\boxed{eq \quad :: \quad \forall T \,.\, T \to T \to Bool}$$

A deep theorem, the *parametricity theorem*, implies that a function of this type must necessarily be constant—roughly speaking, the two arguments cannot be inspected.

☞ As a consequence, the programmer is forced to program a separate equality function for each type from scratch.

Here is how one would define equality for lists.

$$
\begin{array}{lcl}
eqList & :: & \forall A\,.\,(A \to A \to Bool) \\
 & & \to (List\ A \to List\ A \to Bool) \\
eqList\ eqA\ nil\ nil & = & True \\
eqList\ eqA\ nil\ (cons\ a_2\ as_2) & = & False \\
eqList\ eqA\ (cons\ a_1\ as_1)\ nil & = & False \\
\multicolumn{3}{l}{eqList\ eqA\ (cons\ a_1\ as_1)\ (cons\ a_2\ as_2)} \\
 & = & eqA\ a_1\ a_2 \wedge eqList\ as_1\ as_2
\end{array}
$$

☞ *eqList* is actually an 'equality transformer'.

Polytypic programming comes to rescue

Polytypic programming (also known as generic programming, type parametric programming, shape polymorphism, structural polymorphism) addresses this problem.

☞ *Basic idea:* define the equality function by *induction on the structure of types*.

Note that an instance of eq, such as $eqList$, is typically defined by induction on the structure of values.

Polytypic programs are ubiquitous

- Equality and comparison functions.
- Reductions (such as $size$, sum, $listify$).
- Pretty printers (such as Haskell's $show$ function).
- Parsers (such as Haskell's $read$ function).
- Data conversion (Jansson and Jeuring, ESOP'99).
- Digital searching (Hinze, JFP).

A brief outline of the talk

- ✔ Why? (2 – 7)
- ✘ How?—A programmer's perspective (9 – 13)
- ✘ How?—From an implementor's point of view (15 – 26)

Polytypic definitions

In order to define a polytypic value it suffices to give instances for the following three data types—plus an instance for every primitive type.

$$
\begin{aligned}
\mathbf{data}\ 1 &= () \\
\mathbf{data}\ A_1 + A_2 &= \mathit{inl}\ A_1 \mid \mathit{inr}\ A_2 \\
\mathbf{data}\ A_1 \times A_2 &= (A_1, A_2)
\end{aligned}
$$

Here, 1 is the unit data type, '$+$' is a binary sum, and '$\times$' is a binary product (pairs).

Polytypic definitions—equality

For emphasis, the type argument of eq is enclosed in angle brackets.

$$
\begin{array}{lcl}
eq\{\!|A|\!\} & :: & A \to A \to Bool \\
eq\{\!|1|\!\}\ ()\ () & = & True \\
eq\{\!|Int|\!\}\ i_1\ i_2 & = & eqInt\ i_1\ i_2 \\
eq\{\!|A + B|\!\}\ (inl\ a_1)\ (inl\ a_2) & = & eq\{\!|A|\!\}\ a_1\ a_2 \\
eq\{\!|A + B|\!\}\ (inl\ a_1)\ (inr\ b_2) & = & False \\
eq\{\!|A + B|\!\}\ (inr\ b_1)\ (inl\ a_2) & = & False \\
eq\{\!|A + B|\!\}\ (inr\ b_1)\ (inr\ b_2) & = & eq\{\!|B|\!\}\ b_1\ b_2 \\
eq\{\!|A \times B|\!\}\ (a_1, b_1)\ (a_2, b_2) & = & eq\{\!|A|\!\}\ a_1\ a_2 \land eq\{\!|B|\!\}\ b_1\ b_2
\end{array}
$$

This simple definition contains all ingredients needed to derive specializations for arbitrary data types.

Haskell's type classes

Haskell supports *overloading*, based on type classes.

$$\textbf{class}\ Eq\ T\ \textbf{where}$$
$$\quad eq \ :: \ T \to T \to Bool$$

☞ However, overloading is not polytypic programming. For each type, you have to give an explicit instance declaration, containing the code for equality.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ (Eq\ A) \Rightarrow Eq\ (List\ A)\ \textbf{where}} \\
\quad eq\ nil\ nil & = & True \\
\quad eq\ nil\ (cons\ a_2\ as_2) & = & False \\
\quad eq\ (cons\ a_1\ as_1)\ nil & = & False \\
\quad eq\ (cons\ a_1\ as_1)\ (cons\ a_2\ as_2) & = & eq\ a_1\ a_2 \wedge eq\ as_1\ as_2
\end{array}$$

deriving

However, for a handful of built-in classes Haskell provides special support—the so-called '**deriving**' mechanism.

$$\mathbf{data}\ \mathit{List}\ A \quad = \quad \mathit{nil} \mid \mathit{cons}\ A\ (\mathit{List}\ A)\ \mathbf{deriving}\ (\mathit{Eq})$$

The '$\mathbf{deriving}\ (\mathit{Eq})$' part tells Haskell to generate the 'obvious' code for equality.

☞ Alas, you cannot define your own derivable type classes.

Derivable type classes

Idea: use polytypic definitions to specify default method declarations.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{class}\ Eq\ T\ \textbf{where}} \\
\quad eq & :: & T \to T \to Bool \\
\quad eq\{\!|1|\!\}\ ()\ () & = & True \\
\quad eq\{\!|A+B|\!\}\ (inl\ a_1)\ (inl\ a_2) & = & eq\ a_1\ a_2 \\
\quad eq\{\!|A+B|\!\}\ (inl\ a_1)\ (inr\ b_2) & = & False \\
\quad eq\{\!|A+B|\!\}\ (inr\ b_1)\ (inl\ a_2) & = & False \\
\quad eq\{\!|A+B|\!\}\ (inr\ b_1)\ (inr\ b_2) & = & eq\ b_1\ b_2 \\
\quad eq\{\!|A\times B|\!\}\ (a_1, b_1)\ (a_2, b_2) & = & eq\ a_1\ a_2 \wedge eq\ b_1\ b_2
\end{array}
$$

☞ This extension is implemented in the Glasgow Haskell Compiler (Version 5.00).

A brief outline of the talk

- ✔ Why? (2 – 7)
- ✔ How?—A programmer's perspective (9 – 13)
- ✘ How?—From an implementor's point of view (15 – 26)

A closer look at Haskell's data construct

It features *type abstraction*.

$$\mathbf{data}\ List\ A \quad = \quad nil \mid cons\ A\ (List\ A)$$

It features *type application*.

$$\mathbf{data}\ List\ A \quad = \quad nil \mid cons\ A\ (List\ A)$$

It features *type recursion*.

$$\mathbf{data}\ List\ A \quad = \quad nil \mid cons\ A\ (List\ A)$$

☞ The type language corresponds to the simply typed λ-calculus.

Kinds and types

Kinds—can be seen as the 'types of types'.

$$
\begin{array}{lcll}
\mathfrak{T}, \mathfrak{U} \in Kind & ::= & \star & \text{kind of types} \\
 & | & \mathfrak{T} \to \mathfrak{U} & \text{function kind}
\end{array}
$$

Types.

$$
\begin{array}{lcll}
T, U \in Type & ::= & C & \text{type constant} \\
 & | & A & \text{type variable} \\
 & | & \Lambda A \,.\, T & \text{type abstraction} \\
 & | & T \ U & \text{type application}
\end{array}
$$

Modelling data types

Assuming the following type constants

$$
\begin{array}{lcl}
1 & :: & \star \\
(+) & :: & \star \to \star \to \star \\
(\times) & :: & \star \to \star \to \star \\
\mathit{Fix} & :: & ((\star \to \star) \to (\star \to \star)) \to (\star \to \star),
\end{array}
$$

we can rewrite List as a lambda term:

$$
\mathit{List} \quad = \quad \mathit{Fix}\,(\Lambda L\,.\,\Lambda A\,.\,1 + A \times L\;A).
$$

Specializing: unfolding . . .

We could specialize a polytypic value by unfolding its definition:

$$\begin{array}{lcl} eq\{\![List\ Int]\!\} & = & eq\{\![1 + Int \times List\ Int]\!\} \\ & = & eq_{+}\ eq_{1}\ (eq_{\times}\ eq_{Int}\ (eq\{\![List\ Int]\!\})), \end{array}$$

where

$$\begin{array}{lcl} eq_{1}\ ()\ () & = & True \\ eq_{+}\ eq_{A}\ eq_{B}\ (inl\ a_1)\ (inl\ a_2) & = & eq_{A}\ a_1\ a_2 \\ eq_{+}\ eq_{A}\ eq_{B}\ (inl\ a_1)\ (inr\ b_2) & = & False \\ eq_{+}\ eq_{A}\ eq_{B}\ (inr\ b_1)\ (inl\ a_2) & = & False \\ eq_{+}\ eq_{A}\ eq_{B}\ (inr\ b_1)\ (inr\ b_2) & = & eq_{B}\ b_1\ b_2 \\ eq_{\times}\ eq_{A}\ eq_{B}\ (a_1, b_1)\ (a_2, b_2) & = & eq_{A}\ a_1\ a_2 \wedge eq_{B}\ b_1\ b_2. \end{array}$$

. . . does not work

Consider the following data type.

$$\textbf{data}\ \mathit{Power}\ A \;=\; \mathit{zero}\ A \mid \mathit{succ}\ (\mathit{Power}\ (A \times A))$$

Since the type recursion is nested, unfolding eq's definition will loop.

$$\begin{aligned}
& eq\{\!|\mathit{Power}\ \mathit{Int}|\!\} \\
&= eq\{\!|\mathit{Int} + \mathit{Power}\ (\mathit{Int} \times \mathit{Int})|\!\} \\
&= eq_{+}\ eq_{\mathit{Int}}\ (eq\{\!|\mathit{Power}\ (\mathit{Int} \times \mathit{Int})|\!\}) \\
&= eq_{+}\ eq_{\mathit{Int}}\ (eq_{+}\ (eq_{\times}\ eq_{\mathit{Int}}\ eq_{\mathit{Int}}) \\
&\qquad\qquad\qquad\qquad (eq\{\!|\mathit{Power}\ ((\mathit{Int} \times \mathit{Int}) \times (\mathit{Int} \times \mathit{Int}))|\!\})) \\
&= \ldots
\end{aligned}$$

Specializing polytypic values

☞ *Basic idea:* Let function follow type.

$$\begin{array}{lcl} eq\{\!|List\ Int|\!\} & = & eqList\ eqInt \\ eq\{\!|Power\ Int|\!\} & = & eqPower\ eqInt \end{array}$$

Since $List$ and $Power$ are functions on types, $eqList$ and $eqPower$ are consequently functions on equality functions: $eqList$ maps $eq\{\!|A|\!\}$ to $eq\{\!|List\ A|\!\}$.

Specializing generic values—continued

In general, the type of $eq\{\!|T :: \mathfrak{T}|\!\}$ is given by

$$eq\{\!|T :: \mathfrak{T}|\!\} \quad :: \quad Equal\{\!|\mathfrak{T}|\!\}\ T,$$

where $Equal\{\!|\mathfrak{T}|\!\}$ is defined by induction on the structure of kinds.

$$\begin{array}{lcl} Equal\{\!|\star|\!\}\ T & = & T \to T \to Bool \\ Equal\{\!|\mathfrak{A} \to \mathfrak{B}|\!\}\ T & = & \forall A\,.\,Equal\{\!|\mathfrak{A}|\!\}\ A \to Equal\{\!|\mathfrak{B}|\!\}\ (T\ A) \end{array}$$

☞ $eqList$ has type $Equal\{\!|\star \to \star|\!\}\ List$.

Specializing generic values—continued

The specialization of eq to types of arbitrary kinds is given by

$$\begin{array}{lcl} eq\{\!|A|\!\} & = & eq_A \\ eq\{\!|C|\!\} & = & eq_c \\ eq\{\!|T_1\ T_2|\!\} & = & (eq\{\!|T_1|\!\})\ T_2\ (eq\{\!|T_2|\!\}) \\ eq\{\!|\Lambda A\,.\,T|\!\} & = & \lambda A\,.\,\lambda eq_A\,.\,eq\{\!|T|\!\}. \end{array}$$

Type application is mapped to value application, type abstraction to value abstraction, and type recursion to value recursion ($eq_{Fix} = fix$).

☞ This is very similar to an interpretation of the simply typed λ-calculus.

Recap: applicative structures

An *applicative structure* $\mathcal{E}$ is a triple $(\mathbf{E}, \mathbf{app}, \mathbf{const})$ such that

- $\mathbf{E} = (\mathbf{E}^{\mathfrak{T}} \mid \mathfrak{T} \in Kind)$,
- $\mathbf{app} = (\mathbf{app}_{\mathfrak{T},\mathfrak{U}} : \mathbf{E}^{\mathfrak{T} \to \mathfrak{U}} \to (\mathbf{E}^{\mathfrak{T}} \to \mathbf{E}^{\mathfrak{U}}) \mid \mathfrak{T}, \mathfrak{U} \in Kind)$, and
- $\mathbf{const} : const \to \mathbf{E}$ with $\mathbf{const}(C :: \mathfrak{T}) \in \mathbf{E}^{\mathfrak{T}}$.

Recap: environment models

An applicative structure $\mathcal{E} = (\mathbf{E}, \mathbf{app}, \mathbf{const})$ is an *environment model* if it is *extensional* (**app** is one-to-one) and if the clauses below define a total meaning function.

$$
\begin{array}{lcl}
\mathcal{E}[\![C :: \mathfrak{T}]\!]\eta & = & \mathbf{const}(C) \\
\mathcal{E}[\![A :: \mathfrak{T}]\!]\eta & = & \eta(A) \\
\mathcal{E}[\![(\Lambda A \,.\, T) :: (\mathfrak{S} \to \mathfrak{T})]\!]\eta & & \\
 & = & \text{the unique } \varphi \in \mathbf{E}^{\mathfrak{S} \to \mathfrak{T}} \text{ such that} \\
 & & \quad \mathbf{app}_{\mathfrak{S},\mathfrak{T}}\, \varphi\, \delta = \mathcal{E}[\![T :: \mathfrak{T}]\!]\eta(A := \delta) \\
\mathcal{E}[\![(T\ \ U) :: \mathfrak{V}]\!]\eta & = & \mathbf{app}_{\mathfrak{U},\mathfrak{V}}\, (\mathcal{E}[\![T :: \mathfrak{U} \to \mathfrak{V}]\!]\eta)\, (\mathcal{E}[\![U :: \mathfrak{U}]\!]\eta)
\end{array}
$$

Specialization as an interpretation

The applicative structure $\mathcal{E} = (\mathbf{E}, \mathbf{app}, \mathbf{const})$ with

$$\begin{array}{lcl} \mathbf{E}^{\mathfrak{T}} & = & (T :: \mathfrak{T}; Equal\{\!|\mathfrak{T}|\!\}\ T) \\ \mathbf{app}_{\mathfrak{T},\mathfrak{U}}\,(F; f)\,(A; a) & = & (F\ A; f\ A\ a) \\ \mathbf{const}(C) & = & (C; eq_C) \end{array}$$

is an environment model. Here, $(T :: \mathfrak{T}; F\ T)$ denotes a dependent product.

Benefits

- Since the definition of **app** and the interpretation of Fix are the same for all polytypic functions, the polytypic programmer merely has to specify the interpretation of the remaining constants: ‘1’, ‘$+$’, ‘$\times$’, Int, and so on.

- We can use the basic proof method of the simply typed λ-calculus, which is based on so-called *logical relations* to prove properties of polytypic values.

A brief outline of the talk

- ✔ Why? (2 – 7)
- ✔ How?—A programmer's perspective (9 – 13)
- ✔ How?—From an implementor's point of view (15 – 26)