Generic Haskell—Practice and Theory

RALF HINZE

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

August, 2002

(Pick the slides at `.../~ralf/talks.html#T33`.)

Overview

- Generic Haskell—Introduction
- Generic Haskell—Practice
- Generic Haskell—Theory

Prerequisites

A basic knowledge of Haskell is desirable, as all the examples are given either in Haskell or in Generic Haskell, which is an extension of Haskell (and which is the subject of this lecture).

Generic Haskell—Introduction

- Type systems
- Haskell's **data** construct
- Towards generic programming
- Towards Generic Haskell

Safe languages

We probably all agree that language safety is a good thing.

A few definitions stressing different aspects (taken from Pierce's "Types and Programming Languages"):

- A safe language is one that makes it impossible to shoot yourself in the foot while programming.
- A safe language is one that protects its own abstractions.
- A safe language is one that that prevents untrapped errors at run time.
- A safe language is completely defined by its programmer's manual.

Language safety can be achieved by static type checking, by dynamic type checking, or by a combination of static and dynamic checks.

Static typing

Static type checking has a number of benefits:

- Programming errors are detected at an early stage.
- Type systems enforce disciplined programming.
- Types promote abstraction (abstract data types, module systems).
- Types provide machine-checkable documentation.

However, type systems are always conservative: they must necessarily reject programs that behave well at run time.

Dynamic typing

This course has little to offer for addicts of dynamically typed languages.

Polymorphic type systems

Polymorphism complements safety by flexibility.

Polymorphism allows the definition of functions that behave uniformly over all types.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{List}\ a & = & \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \\
\mathit{length} & :: & \forall a\,.\,\mathit{List}\ a \to \mathit{Int} \\
\mathit{length}\ \mathit{Nil} & = & 0 \\
\mathit{length}\ (\mathit{Cons}\ a\ \mathit{as}) & = & 1 + \mathit{length}\ \mathit{as}
\end{array}
$$

The function length happens to be insensitive to the type of the list elements.

However, . . .

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

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

$eq :: \forall a . a \to a \to Bool$ -- does not work

Parametricity 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.

Haskell **data** construct

In Haskell new types are introduced via **data** declarations.

A Haskell data type is essentially a sum of products.

$$\textbf{data}\ String\ =\ Nil \mid Cons\ Char\ String$$

The type $String$ is a binary sum. The first summand, Nil, is a nullary product and the second summand, $Cons$, is a binary product.

Haskell's data construct

Data types may have type arguments, that is, we have (a simple form of) type abstraction and type application.

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

The type $List$ is obtained from $String$ by abstracting over $Char$.

Haskell's data construct

Type arguments may also range over type constructors.

$$\begin{aligned} \textbf{data}\ GRose\ f\ a &= Branch\ a\ (f\ (GRose\ f\ a)) \\ \textbf{data}\ Fix\ f &= In\ (f\ (Fix\ f)) \end{aligned}$$

Haskell's kind system ensures that type terms are well-formed. We have $GRose :: (* \to *) \to (* \to *)$ and $Fix :: (* \to *) \to *$.

The '$*$' kind represents manifest types such as $Char$ or Int.

The kind $k \to l$ represents type constructors that map type constructors of kind k to those of kind l.

Towards generic programming

Now, let's define equality functions for the types above.

$$eqString \;::\; String \rightarrow String \rightarrow Bool$$

$$\begin{aligned}
&eqString\ Nil\ Nil &&= True \\
&eqString\ Nil\ (Cons\ c'\ s') &&= False \\
&eqString\ (Cons\ c\ s)\ Nil &&= False \\
&eqString\ (Cons\ c\ s)\ (Cons\ c'\ s') &&= eqChar\ c\ c' \wedge eqString\ s\ s'
\end{aligned}$$

The function $eqChar :: Char \rightarrow Char \rightarrow Bool$ is equality of characters.

Towards generic programming

The type $List$ is obtained from $String$ by abstracting over $Char$. Likewise, $eqList$ is obtained from $eqString$ by abstracting over $eqChar$.

$$eqList \; :: \; \forall a \, . \, (a \rightarrow a \rightarrow Bool) \rightarrow (List \; a \rightarrow List \; a \rightarrow Bool)$$

$$\begin{array}{lcl}
eqList \; eqa \; Nil \; Nil & = & True \\
eqList \; eqa \; Nil \; (Cons \; a' \; x') & = & False \\
eqList \; eqa \; (Cons \; a \; x) \; Nil & = & False \\
eqList \; eqa \; (Cons \; a \; x) \; (Cons \; a' \; x') & = & eqa \; a \; a' \wedge eqList \; eqa \; x \; x'
\end{array}$$

Towards generic programming

The type $GRose$ abstracts over a type constructor (of kind $* \to *$) and over a type (of kind $*$). The equality function $eqGRose$ follows the type structure.

$$
\begin{aligned}
& eqGRose :: \forall f\,.\,(\forall a\,.\,(a \to a \to Bool) \to (f\ a \to f\ a \to Bool)) \\
& \qquad\qquad \to (\forall a\,.\,(a \to a \to Bool) \\
& \qquad\qquad\qquad \to (GRose\ f\ a \to GRose\ f\ a \to Bool)) \\
& eqGRose\ eqf\ eqa\ (Branch\ a\ f)\ (Branch\ a'\ f') \\
& \qquad\qquad\qquad = eqa\ a\ a' \wedge eqf\ (eqGRose\ eqf\ eqa)\ f\ f'
\end{aligned}
$$

$$
\begin{aligned}
& eqFix :: \forall f\,.\,(\forall a\,.\,(a \to a \to Bool) \to (f\ a \to f\ a \to Bool)) \\
& \qquad\qquad \to (Fix\ f \to Fix\ f \to Bool) \\
& eqFix\ eqf\ (In\ f)\ (In\ f') = eqf\ (eqFix\ eqf)\ f\ f'
\end{aligned}
$$

Towards Generic Haskell

- Observation: the type of eqT depends on the kind of T. The more complicated the kind of T, the more complicated the type of eqT.
- Apart from the typings, it's crystal clear what the definition of eqT looks like.
- Coding the equality function is boring and error-prone.
- Generic Haskell allows to capture the commonality.
- The generic equality function works for all types of all kinds (except, of course, for functional types).

Kind-indexed types

The type of the generic equality function is captured by the following kind-indexed type (the part enclosed in $\{\![\cdot]\!\}$ is the kind index).

$$
\begin{array}{lll}
\textbf{type } Eq\{\![*]\!\}\ t & = & t \to t \to Bool \\
\textbf{type } Eq\{\![k \to l]\!\}\ t & = & \forall a\,.\,Eq\{\![k]\!\}\ a \to Eq\{\![l]\!\}\ (t\ a)
\end{array}
$$

We have $eqString :: Eq\{\![*]\!\}\ String$, $eqList :: Eq\{\![* \to *]\!\}\ List$, and $eqFix :: Eq\{\![(* \to *) \to *]\!\}\ Fix$.

Sums and products

- Recall that Haskell's data types are essentially sums of products.
- To cover data types the generic programmer only has to define the generic function for binary sums and binary products (and nullary products).
- To this end Generic Haskell provides the following data types.

$$
\begin{array}{lll}
\mathbf{data}\ \mathit{Unit} & = & \mathit{Unit} \\
\mathbf{data}\ a :\!*\!: b & = & a :\!*\!: b \\
\mathbf{data}\ a :\!+\!: b & = & \mathit{Inl}\ a \mid \mathit{Inr}\ b
\end{array}
$$

Type-indexed values

The definition of generic equality is straightforward (eq is a type-indexed value; the part enclosed in $\{|\cdot|\}$ is the type index).

$$
\begin{array}{lcl}
eq\{|t :: k|\} & :: & Eq\{[k]\}\ t \\
eq\{|Char|\} & = & eqChar \\
eq\{|Int|\} & = & eqInt \\
eq\{|Unit|\}\ Unit\ Unit & = & True \\
eq\{|{:+:}|\}\ eqa\ eqb\ (Inl\ a)\ (Inl\ a') & = & eqa\ a\ a' \\
eq\{|{:+:}|\}\ eqa\ eqb\ (Inl\ a)\ (Inr\ b') & = & False \\
eq\{|{:+:}|\}\ eqa\ eqb\ (Inr\ b)\ (Inl\ a') & = & False \\
eq\{|{:+:}|\}\ eqa\ eqb\ (Inr\ b)\ (Inr\ b') & = & eqb\ b\ b' \\
eq\{|{:*:}|\}\ eqa\ eqb\ (a :*: b)\ (a' :*: b') & = & eqa\ a\ a' \wedge eqb\ b\ b'
\end{array}
$$

Generic Haskell takes care of type abstraction, type application and type recursion.

Generic application

Given the definition above we can use generic equality at any type of any kind.

$eq\{\!|List\ Char|\!\}$ `"hello" "Hello"`
$\Longrightarrow False$
$\textbf{let}\ sim\ c\ c' = eqChar\ (toUpper\ c)\ (toUpper\ c')$
$eq\{\!|List|\!\}\ sim$ `"hello" "Hello"`
$\Longrightarrow True$

Generic abstraction

Common idioms can be captured using generic abstractions.

$$
\begin{aligned}
&\mathit{similar}\{\!|t :: * \to *|\!\} &&:: \forall t\,.\, t\ \mathit{Char} \to t\ \mathit{Char} \to \mathit{Bool} \\
&\mathit{similar}\{\!|t|\!\} &&= \mathit{eq}\{\!|t|\!\}\ \mathit{sim}
\end{aligned}
$$

Note that $\mathit{similar}$ is only applicable to type constructors of kind $* \to *$.

Stocktaking

Modern functional programming languages such as Haskell 98 typically have a three level structure (ignoring the module system).

- values
- types — imposing structure on the value level
- kinds — imposing structure on the type level

Stocktaking

In ‘ordinary’ programming we define

- values depending on values (called functions),
- types depending on types (called type constructors).

Generic programming adds to this list the possibility of defining

- values depending on types (called generic functions or type-indexed values),
- types depending on kinds (called kind-indexed types).

Type-safety is not compromised.

* * *

Overview

- √ Generic Haskell: Introduction
- ▶ Generic Haskell: Practice
- ▶ Generic Haskell: Theory

Generic Haskell—Practice

- Mapping functions
- Kind-indexed types and type-indexed values
- Reductions
- Pretty printing

Mapping functions

A mapping function for a type constructor F of kind $* \longrightarrow *$ lifts a given function of type $a \longrightarrow b$ to a function of type $F\ a \longrightarrow F\ b$.

The all-time favourite:

$$\begin{array}{lcl} mapList & :: & \forall a\ b\,.\,(a \longrightarrow b) \longrightarrow (List\ a \longrightarrow List\ b) \\ mapList\ f\ Nil & = & Nil \\ mapList\ f\ (Cons\ a\ as) & = & Cons\ (f\ a)\ (mapList\ f\ as) \end{array}$$

The mapping function for lists applies the function to each list element.

Mapping functions

Can we generalize mapping functions so that they work for all types of all kinds?

Yes!

Let's tackle the type first. A first attempt:

$$
\begin{aligned}
&\textbf{type } Map\{[*]\}\ t &&= t \to t \qquad \text{-- WRONG}\\
&\textbf{type } Map\{[k \to l]\}\ t &&= \forall a\,.\, Map\{[k]\}\ a \to Map\{[l]\}\ (t\ a)
\end{aligned}
$$

Alas, we have $Map\{[* \to *]\}\ List = \forall a\,.\,(a \to a) \to (List\ a \to List\ a)$, which is not general enough.

Mapping functions

We need two type arguments:

$$
\begin{aligned}
&\textbf{type}\ \mathit{Map}\{\![*]\!\}\ t_1\ t_2 && = && t_1 \to t_2 \\
&\textbf{type}\ \mathit{Map}\{\![k \to l]\!\}\ t_1\ t_2 && = && \forall a_1\ a_2\,.\ \mathit{Map}\{\![k]\!\}\ a_1\ a_2 \\
& && && \qquad\qquad \to \mathit{Map}\{\![l]\!\}\ (t_1\ a_1)\ (t_2\ a_2) \\
&\mathit{map}\{\![t :: k]\!\} && :: && \mathit{Map}\{\![k]\!\}\ t\ t
\end{aligned}
$$

Now, $\mathit{Map}\{\![* \to *]\!\}\ \mathit{List}\ \mathit{List} = \forall a_1\ a_2\,.\,(a_1 \to a_2) \to (\mathit{List}\ a_1 \to \mathit{List}\ a_2)$ as desired.

Mapping functions

The definition of map itself is straightforward (really!):

$$
\begin{array}{lcl}
map\{\!|t :: k|\!\} & :: & Map\{\![k]\!\}\ t\ t \\
map\{\!|Char|\!\}\ c & = & c \\
map\{\!|Int|\!\}\ i & = & i \\
map\{\!|Unit|\!\}\ Unit & = & Unit \\
map\{\!|{:+:}|\!\}\ mapa\ mapb\ (Inl\ a) & = & Inl\ (mapa\ a) \\
map\{\!|{:+:}|\!\}\ mapa\ mapb\ (Inr\ b) & = & Inr\ (mapb\ b) \\
map\{\!|{:*:}|\!\}\ mapa\ mapb\ (a\ {:*:}\ b) & = & mapa\ a\ {:*:}\ mapb\ b
\end{array}
$$

Mapping functions

Generic applications:

$$map\{\!|List\ Char|\!\}\ \texttt{"hello world"}$$
$$\Longrightarrow\ \texttt{"hello world"}$$
$$map\{\!|List|\!\}\ toUpper\ \texttt{"hello world"}$$
$$\Longrightarrow\ \texttt{"HELLO WORLD"}$$

Generic abstraction:

$$\begin{aligned} distribute\{\!|t :: * \rightarrow *|\!\} &\ ::\ \forall a\ b\ .\ t\ a \rightarrow b \rightarrow t\ (a, b) \\ distribute\{\!|t|\!\}\ x\ b &\ =\ map\{\!|t|\!\}\ (\lambda a \rightarrow (a, b))\ x \end{aligned}$$

Kind-indexed types

In general, a kind-indexed type is defined as follows:

$$
\begin{aligned}
&\textbf{type } \mathit{Poly}\{\![*]\!\}\ t_1\ \ldots\ t_n\ =\ \ldots \\
&\textbf{type } \mathit{Poly}\{\![k \to l]\!\}\ t_1\ \ldots\ t_n \\
&\qquad = \forall a_1\ \ldots\ a_n\,.\ \mathit{Poly}\{\![k]\!\}\ a_1\ \ldots\ a_n \\
&\qquad\qquad \to \mathit{Poly}\{\![l]\!\}\ (t_1\ a_1)\ \ldots\ (t_n\ a_n)
\end{aligned}
$$

The second clause is the same for all kind-indexed types.

NB. Generic Haskell allows a slightly more general form (see below).

Type-indexed values

A type-indexed value is defined as follows:

$$
\begin{array}{lcl}
poly\{\!|t :: k|\!\} & :: & Poly\{\![k]\!\}\ t\ \ldots\ t \\
poly\{\!|Char|\!\} & = & \ldots \\
poly\{\!|Int|\!\} & = & \ldots \\
poly\{\!|Unit|\!\} & = & \ldots \\
poly\{\!|{:}{+}{:}|\!\}\ polya\ polyb & = & \ldots \\
poly\{\!|{:}{*}{:}|\!\}\ polya\ polyb & = & \ldots
\end{array}
$$

We have one clause for each primitive type ($Char$, Int etc) and one clause for each of the three type constructors $Unit$, :+:, and :*:.

NB. The type signature can be more elaborate (we will see examples of this).

Equality, revisited

Recall the type of the generic equality function:

$$\begin{array}{lcl} \textbf{type}\ Eq\{[\![*]\!]\}\ t & = & t \to t \to Bool \\ \textbf{type}\ Eq\{[\![k \to l]\!]\}\ t & = & \forall a\,.\,Eq\{[\![k]\!]\}\ a \to Eq\{[\![l]\!]\}\ (t\ a) \end{array}$$

In fact, the two arguments need not be of the same type.

$$\begin{array}{lcl} \textbf{type}\ Eq\{[\![*]\!]\}\ t_1\ t_2 & = & t_1 \to t_2 \to Bool \\ \textbf{type}\ Eq\{[\![k \to l]\!]\}\ t_1\ t_2 & = & \forall a_1\ a_2\,.\,Eq\{[\![k]\!]\}\ a_1\ a_2 \to Eq\{[\![l]\!]\}\ (t_1\ a_1)\ (t_2\ a_2) \end{array}$$

The definition of eq is not affected by this change!

Reductions

The Haskell standard library defines a vast number of list processing functions. Among others:

$$\begin{array}{lcl} sum, product & :: & (Num\ a) \Rightarrow [a] \rightarrow a \\ and, or & :: & [Bool] \rightarrow Bool \\ all, any & :: & (a \rightarrow Bool) \rightarrow [a] \rightarrow Bool \\ length & :: & [a] \rightarrow Int \\ minimum, maximum & :: & (Ord\ a) \Rightarrow [a] \rightarrow a \\ concat & :: & [[a]] \rightarrow [a] \end{array}$$

These are examples of so-called reductions. A reductions reduces (or crushes) a list of something to something. Reductions can be generalized from lists to arbitrary data types.

A simple case: summing up

Let's start with a simple instance.

$$
\begin{array}{lcl}
\textbf{type } \mathit{Sum}\{[*]\}\ t & = & t \to \mathit{Int} \\
\textbf{type } \mathit{Sum}\{[k \to l]\}\ t & = & \forall a\,.\,\mathit{Sum}\{[k]\}\ a \to \mathit{Sum}\{[l]\}\ (t\ a) \\
\mathit{sum}\{|t :: k|\} & :: & \mathit{Sum}\{[k]\}\ t \\
\mathit{sum}\{|\mathit{Char}|\}\ c & = & 0 \\
\mathit{sum}\{|\mathit{Int}|\}\ i & = & 0 \\
\mathit{sum}\{|\mathit{Unit}|\}\ \mathit{Unit} & = & 0 \\
\mathit{sum}\{|{:}{+}{:}|\}\ \mathit{suma}\ \mathit{sumb}\ (\mathit{Inl}\ a) & = & \mathit{suma}\ a \\
\mathit{sum}\{|{:}{+}{:}|\}\ \mathit{suma}\ \mathit{sumb}\ (\mathit{Inr}\ b) & = & \mathit{sumb}\ b \\
\mathit{sum}\{|{:}{*}{:}|\}\ \mathit{suma}\ \mathit{sumb}\ (a\ {:}{*}{:}\ b) & = & \mathit{suma}\ a + \mathit{sumb}\ b
\end{array}
$$

A simple case: summing up

Generic applications.

$$sum\{\!|List\ Int|\!\}\ [2, 7, 1965]$$
$$\Longrightarrow 0$$
$$sum\{\!|List|\!\}\ id\ [2, 7, 1965]$$
$$\Longrightarrow 1974$$
$$sum\{\!|List|\!\}\ (const\ 1)\ [2, 7, 1965]$$
$$\Longrightarrow 3$$

Generic abstractions.

$$\begin{aligned}
fsum\{\!|t :: * \to *|\!\} &:: t\ Int \to Int \\
fsum\{\!|t|\!\} &= sum\{\!|t|\!\}\ id \\
fsize\{\!|t :: * \to *|\!\} &:: \forall a\,.\,t\ a \to Int \\
fsize\{\!|t|\!\} &= sum\{\!|t|\!\}\ (const\ 1)
\end{aligned}$$

Reductions

We abstract away from Int, 0 and '$+$'.

$$\begin{aligned}
&\textbf{type}\ Reduce\{\![*]\!\}\ t\ x &&= x \to (x \to x \to x) \to t \to x \\
&\textbf{type}\ Reduce\{\![k \to l]\!\}\ t\ x &&= \forall a\,.\, Reduce\{\![k]\!\}\ a\ x \to Reduce\{\![l]\!\}\ (t\ a)\ x
\end{aligned}$$

Note that the type argument x is passed unchanged to the recursive calls (x can be seen as being global to the definition).

Reductions

The generic function *reduce* generalizes *sum*.

$$
\begin{array}{lcl}
reduce\{\!|t :: k|\!\} & :: & \forall x \,.\, Reduce\{\![k]\!\}\ t\ x \\
reduce\{\!|Char|\!\}\ e\ op\ c & = & e \\
reduce\{\!|Int|\!\}\ e\ op\ i & = & e \\
reduce\{\!|Unit|\!\}\ e\ op\ Unit & = & e \\
reduce\{\!|{:}{+}{:}|\!\}\ reda\ redb\ e\ op\ (Inl\ a) & = & reda\ e\ op\ a \\
reduce\{\!|{:}{+}{:}|\!\}\ reda\ redb\ e\ op\ (Inr\ b) & = & redb\ e\ op\ b \\
reduce\{\!|{:}{*}{:}|\!\}\ reda\ redb\ e\ op\ (a\ {:}{*}{:}\ b) & = & reda\ e\ op\ a\ `op`\ redb\ e\ op\ b
\end{array}
$$

Reductions

$$
\begin{array}{lcl}
\mathit{freduce}\{\!|t :: * \to *|\!\} & :: & \forall x \,.\, x \to (x \to x \to x) \to t\; x \to x \\
\mathit{freduce}\{\!|t|\!\} & = & \mathit{reduce}\{\!|t|\!\}\; (\lambda e\; \mathit{op}\; a \to a) \\
\mathit{fsum}\{\!|t|\!\} & = & \mathit{freduce}\{\!|t|\!\}\; 0\; (+) \\
\mathit{fproduct}\{\!|t|\!\} & = & \mathit{freduce}\{\!|t|\!\}\; 1\; (*) \\
\mathit{fand}\{\!|t|\!\} & = & \mathit{freduce}\{\!|t|\!\}\; \mathit{True}\; (\wedge) \\
\mathit{for}\{\!|t|\!\} & = & \mathit{freduce}\{\!|t|\!\}\; \mathit{False}\; (\vee) \\
\mathit{fall}\{\!|t|\!\}\; f & = & \mathit{fand}\{\!|t|\!\} \cdot \mathit{map}\{\!|t|\!\}\; f \\
\mathit{fany}\{\!|t|\!\}\; f & = & \mathit{for}\{\!|t|\!\} \cdot \mathit{map}\{\!|t|\!\}\; f \\
\mathit{fminimum}\{\!|t|\!\} & = & \mathit{freduce}\{\!|t|\!\}\; \mathit{maxBound}\; \mathit{min} \\
\mathit{fmaximum}\{\!|t|\!\} & = & \mathit{freduce}\{\!|t|\!\}\; \mathit{minBound}\; \mathit{max} \\
\mathit{fflatten}\{\!|t|\!\} & = & \mathit{freduce}\{\!|t|\!\}\; [\,]\; (+\!\!+)
\end{array}
$$

Pretty printing

Let's reimplement (a simple version of) Haskell's $shows$ function.

Problem: we need to know the constructor names.

Solution: we introduce an additional case:

$$poly\{\!|Con\ c|\!\}\ polya\ =\ \ldots$$

This case is invoked whenever we pass by a constructor.

The variable c is bound to a value of type $ConDescr$ and provides information about the name of a constructor, its arity etc.

Pretty printing

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{ConDescr} & = & \mathit{ConDescr}\{\mathit{conName} :: \mathit{String},\\
& & \qquad\qquad\quad \mathit{conType} :: \mathit{String},\\
& & \qquad\qquad\quad \mathit{conArity} :: \mathit{Int},\\
& & \qquad\qquad\quad \mathit{conLabels} :: \mathit{Bool},\\
& & \qquad\qquad\quad \mathit{conFixity} :: \mathit{Fixity}\}\\
\textbf{data}\ \mathit{Fixity} & = & \mathit{Nonfix}\\
& \mid & \mathit{Infix}\{\mathit{prec} :: \mathit{Int}\}\\
& \mid & \mathit{Infixl}\{\mathit{prec} :: \mathit{Int}\}\\
& \mid & \mathit{Infixr}\{\mathit{prec} :: \mathit{Int}\}
\end{array}
$$

Pretty printing

Via ':+:' we get to the constructors, Con signals that we hit a constructor, and via ':∗:' we get to the arguments of a constructor.

$$
\begin{array}{lcl}
\mathbf{type}\ \mathit{Shows}\{[*]\}\ t & = & t \to \mathit{ShowS} \\
\mathbf{type}\ \mathit{Shows}\{[k \to l]\}\ t & = & \forall a\,.\,\mathit{Shows}\{[k]\}\ a \to \mathit{Shows}\{[l]\}\ (t\ a) \\
\mathit{gshows}\{|t :: k|\} & :: & \mathit{Shows}\{[k]\}\ t \\
\mathit{gshows}\{|{:+:}|\}\ sa\ sb\ (\mathit{Inl}\ a) & = & sa\ a \\
\mathit{gshows}\{|{:+:}|\}\ sa\ sb\ (\mathit{Inr}\ b) & = & sb\ b \\
\mathit{gshows}\{|\mathit{Con}\ c|\}\ sa\ (\mathit{Con}\ a) & & \\
\quad \mid\ \mathit{conArity}\ c == 0 & = & \mathit{showString}\ (\mathit{conName}\ c) \\
\quad \mid\ \mathit{otherwise} & = & \mathit{showChar}\ \texttt{'('} \cdot \mathit{showString}\ (\mathit{conName}\ c) \\
 & & \cdot\ \mathit{showChar}\ \texttt{' '} \cdot sa\ a \cdot \mathit{showChar}\ \texttt{')'} \\
\mathit{gshows}\{|{:*:}|\}\ sa\ sb\ (a :*: b) & = & sa\ a \cdot \mathit{showChar}\ \texttt{' '} \cdot sb\ b \\
\mathit{gshows}\{|\mathit{Unit}|\}\ \mathit{Unit} & = & \mathit{showString}\ \texttt{""} \\
\mathit{gshows}\{|\mathit{Char}|\} & = & \mathit{shows} \\
\mathit{gshows}\{|\mathit{Int}|\} & = & \mathit{shows}
\end{array}
$$

Pretty printing

The generic programmer views, for instance, the list data type

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

as if it were given by the following type definition.

$$\textbf{type}\ List\ a\ =\ (Con\ Unit)\ {:+:}\ (Con\ (a\ {:*:}\ List\ a))$$

Pretty printing

The *shows* function generates one long string.

We can do better using pretty printing combinators.

$$
\begin{array}{lll}
empty & :: & Doc \\
(\diamond) & :: & Doc \rightarrow Doc \rightarrow Doc \\
string & :: & String \rightarrow Doc \\
nl & :: & Doc \\
nest & :: & Int \rightarrow Doc \rightarrow Doc \\
group & :: & Doc \rightarrow Doc \\
ppParen & :: & Bool \rightarrow Doc \rightarrow Doc
\end{array}
$$

Pretty printing

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Pretty}\{[*]\}\ t & = & \mathit{Int} \to t \to \mathit{Doc} \\
\textbf{type}\ \mathit{Pretty}\{[k \to l]\}\ t & = & \forall a\,.\ \mathit{Pretty}\{[k]\}\ a \to \mathit{Pretty}\{[l]\}\ (t\ a) \\
\mathit{ppPrec}\{\mid t :: k \mid\} & :: & \mathit{Pretty}\{[k]\}\ t \\
\mathit{ppPrec}\{\mid :\!+\!: \mid\}\ \mathit{ppa}\ \mathit{ppb}\ d\ (\mathit{Inl}\ a) & = & \mathit{ppa}\ d\ a \\
\mathit{ppPrec}\{\mid :\!+\!: \mid\}\ \mathit{ppa}\ \mathit{ppb}\ d\ (\mathit{Inr}\ b) & = & \mathit{ppb}\ d\ b \\
\mathit{ppPrec}\{\mid \mathit{Con}\ c \mid\}\ \mathit{ppa}\ d\ (\mathit{Con}\ a) & & \\
\quad \mid \mathit{conArity}\ c == 0 & = & \mathit{string}\ (\mathit{conName}\ c) \\
\quad \mid \mathit{otherwise} & = & \mathit{group}\ (\mathit{nest}\ 2\ (\mathit{ppParen}\ (d > 9)\ \mathit{doc})) \\
\quad \textbf{where}\ \mathit{doc} & = & \mathit{string}\ (\mathit{conName}\ c) \diamond \mathit{nl} \diamond \mathit{ppa}\ 10\ a \\
\mathit{ppPrec}\{\mid :\!*\!: \mid\}\ \mathit{ppa}\ \mathit{ppb}\ d\ (a :\!*\!: b) & = & \mathit{ppa}\ d\ a \diamond \mathit{nl} \diamond \mathit{ppb}\ d\ b \\
\mathit{ppPrec}\{\mid \mathit{Unit} \mid\}\ d\ \mathit{Unit} & = & \mathit{empty} \\
\mathit{ppPrec}\{\mid \mathit{Int} \mid\}\ d\ i & = & \mathit{string}\ (\mathit{show}\ i) \\
\mathit{ppPrec}\{\mid \mathit{Char} \mid\}\ d\ c & = & \mathit{string}\ (\mathit{show}\ c)
\end{array}
$$

Stocktaking

- A generic function works for all types of all kinds.
- A type-indexed value has a kind-indexed type.
- Constructor names are accessed via the Con case.

* * *

Overview

- √ Generic Haskell: Introduction
- √ Generic Haskell: Practice
- ▶ Generic Haskell: Theory

Generic Haskell—Theory

- Modelling data types
- The simply typed lambda calculus
- The polymorphic lambda calculus
- Specialization as an interpretation
- Bridging the gap

Specialization

Generic Haskell takes a transformational approach: a generic function is translated into a family of polymorphic functions.

This transformation can be phrased as an interpretation of the simply typed lambda calculus (types are simply typed lambda terms with kinds playing the role of types).

To make this precise we switch from Haskell to the polymorphic lambda calculus (also known as $F\omega$).

The polymorphic lambda calculus uses structural equivalence of types, whereas Haskell's type system is based on name equivalence. We have to do a bit of extra work to bridge the gap.

Modelling data types

Recall: the generic programmer views the data type

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

as if it were given by the following type definition.

$$\mathbf{type}\ List\ a \ = \ Unit :+: (a :*: List\ a)$$

NB. For simplicity, we omit the Con types.

Modelling data types

Haskell offers (a simple form of) type abstraction and type application. Thus, types can be modelled by terms of the simply typed lambda calculus.

$$\textbf{type}\ \mathit{List}\ =\ \Lambda A\,.\,\mathit{Fix}\ (\Lambda L\,.\,\mathit{Unit}\ \mathsf{:+:}\ (A\ \mathsf{:*:}\ L))$$

Here, Fix is the fixed point combinator and Unit, ':+:', and ':∗:' are type constants.

NB. We are cheating a bit here. In Haskell each **data** declaration introduces a new type (which is not equal to a sum of products). We will address this point later.

The simply typed lambda calculus

Kinds.

$\mathfrak{T}, \mathfrak{U} \in \mathit{Kind}$	$::=$	$*$		base kind
	$\mid$	$(\mathfrak{T} \to \mathfrak{U})$		function kind

Types.

$C \in \mathit{Const}$

$T, U \in \mathit{Type}$	$::=$	C	type constant
	$\mid$	A	type variable
	$\mid$	$(\Lambda A :: \mathfrak{U} \,.\, T)$	type abstraction
	$\mid$	$(T\ \ U)$	type application

We assume that Const contains at least Unit, ':+:', ':∗:', and a family of fixed point combinators $\mathit{Fix}_{\mathfrak{T}} :: (\mathfrak{T} \to \mathfrak{T}) \to \mathfrak{T}$.

Applicative structures

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

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

An applicative structure is extensional if $\mathbf{app}_{\mathfrak{T},\mathfrak{U}}\ \phi_1 = \mathbf{app}_{\mathfrak{T},\mathfrak{U}}\ \phi_2$ implies $\phi_1 = \phi_2$ (that is, $\mathbf{app}_{\mathfrak{T},\mathfrak{U}}$ is one-to-one).

Environment models

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

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

Extensionality ensures that there is at most one ϕ; there is at least one ϕ is $\mathcal{E}$ has 'enough points' (so that S and K combinators can be defined).

The polymorphic lambda calculus

Type schemes.

$$
\begin{array}{rcll}
R, S \in \mathit{Scheme} & ::= & T & \text{type term} \\
& \mid & (R \rightarrow S) & \text{functional type} \\
& \mid & (\forall A :: \mathfrak{U} \, . \, S) & \text{polymorphic type}
\end{array}
$$

The polymorphic lambda calculus

Terms.

$$
\begin{array}{llll}
c \in \mathit{const} & & & \\
t, u \in \mathit{Term} & ::= & c & \text{constant} \\
 & \mid & a & \text{variable} \\
 & \mid & (\lambda a :: S \,.\, t) & \text{abstraction} \\
 & \mid & (t\ u) & \text{application} \\
 & \mid & (\lambda A :: \mathfrak{U} \,.\, t) & \text{universal abstraction} \\
 & \mid & (t\ R) & \text{universal application}
\end{array}
$$

We assume that const includes at least the polymorphic fixed point operator $\mathit{fix} :: \forall A \,.\, (A \rightarrow A) \rightarrow A$ and suitable constants for each type constant.

Generic functions as models

Here is the definition of map using the syntax of the polymorphic lambda calculus.

$$
\begin{array}{lcl}
Map\{\![*]\!\}\ T_1\ T_2 & = & T_1 \to T_2 \\
Map\{\![\mathfrak{T} \to \mathfrak{U}]\!\}\ T_1\ T_2 & = & \forall A_1\ A_2 \,.\, Map\{\![\mathfrak{T}]\!\}\ A_1\ A_2 \\
 & & \qquad\qquad \to Map\{\![\mathfrak{U}]\!\}\ (T_1\ A_1)\ (T_2\ A_2) \\
map\{\!|\,Unit\,|\!\} & = & \lambda u \,.\, u \\
map\{\!|\,{:+:}\,|\!\} & = & \lambda A_1\ A_2 \,.\, \lambda map_A :: (A_1 \to A_2) \,. \\
 & & \lambda B_1\ B_2 \,.\, \lambda map_B :: (B_1 \to B_2) \,. \\
 & & \quad \lambda s \,.\, \mathbf{case}\ s\ \mathbf{of}\ \{\, inl\ a \Rightarrow inl\ (map_A\ a); \\
 & & \qquad\qquad\qquad\quad inr\ b \Rightarrow inr\ (map_B\ b)\,\} \\
map\{\!|\,{:*:}\,|\!\} & = & \lambda A_1\ A_2 \,.\, \lambda map_A :: (A_1 \to A_2) \,. \\
 & & \lambda B_1\ B_2 \,.\, \lambda map_B :: (B_1 \to B_2) \,. \\
 & & \quad \lambda p \,.\, (map_A\ (outl\ p), map_B\ (outr\ p))
\end{array}
$$

Generic functions as models

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

$$\mathbf{M}^{\mathfrak{T}} = \langle T_1, T_2 :: \mathfrak{T}; Map\{[\mathfrak{T}]\}\ T_1\ T_2 \rangle$$

$$\mathbf{app}_{\mathfrak{T},\mathfrak{U}} \langle F_1, F_2; f \rangle \langle A_1, A_2; a \rangle = \langle F_1\ A_1, F_2\ A_2; f\ A_1\ A_2\ a \rangle$$

$$\mathbf{const}(C) = \langle C, C; map\{[C]\} \rangle$$

is an environment model. Here, $\langle T_1, T_2 :: \mathfrak{T}; F\ T_1\ T_2 \rangle$ denotes a dependent product.

Formally, one has to work with equivalence classes of types and terms.

Fixed points

To model recursion the set of type constants includes a family of fixed point combinators: $Fix_{\mathfrak{T}} :: (\mathfrak{T} \to \mathfrak{T}) \to \mathfrak{T}$.

They can be interpreted generically, that is, the interpretation is the same for each generic function (of the same 'arity').

$$\textbf{const}(Fix_{\mathfrak{T}}) \;=\; \langle Fix_{\mathfrak{T}}, Fix_{\mathfrak{T}}; \lambda F_1\ F_2 \,.\, \lambda f :: Map_{\mathfrak{T}\to\mathfrak{T}}\ F_1\ F_2 \,.\\ lfp\ (f\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2))\rangle,$$

where $lfp :: \forall A \,.\, (A \to A) \to A$ is the fixed point combinator on the term level (its type argument $Map_{\mathfrak{T}}\ (Fix_{\mathfrak{T}}\ F_1)\ (Fix_{\mathfrak{T}}\ F_2)$ is omitted above).

An example

As a simple example let us specialize map for the type $Matrix$.

$$\begin{aligned} Matrix &:: * \to * \\ Matrix &= \Lambda A \,.\, List\ (List\ A) \end{aligned}$$

$$\mathcal{M}[\![Matrix]\!] = \langle Matrix, Matrix; mapMatrix \rangle$$

$$\begin{aligned} mapMatrix &:: \forall A_1\ A_2 .\ (A_1 \to A_2) \to (Matrix\ A_1 \to Matrix\ A_2) \\ mapMatrix &= \lambda A_1\ A_2 .\ \lambda map_A :: (A_1 \to A_2)\ . \\ &\quad\ mapList\ (List\ A_1)\ (List\ A_2)\ (mapList\ A_1\ A_2\ map_A) \end{aligned}$$

Bridging the gap

In Haskell, the type $List\ A$ is not equal to $Unit$:+: $(a$:*: $List\ a)$. We have to perform some impedance-matching.

We introduce generic representation types, which mediate between the two representations. For instance, the generic representation type for $List$ is given by

$$\textbf{type}\ List^{\circ}\ a\ =\ \ Unit\ \text{:+:}\ a * List\ a.$$

NB. $List^{\circ}$ is not recursive.

Idea: generate code for $poly\{\!|List^{\circ}|\!\}$ and then implement $poly\{\!|List|\!\}$ by applying a representation transformer.

Conversion

The type $\mathit{List}^{\circ}\ A$ is isomorphic to $\mathit{List}\ A$.

$$
\begin{array}{lcl}
\mathit{fromList} & :: & \forall A\,.\,\mathit{List}\ A \longrightarrow \mathit{List}^{\circ}\ A \\
\mathit{fromList}\ \mathit{Nil} & = & \mathit{Inl}\ \mathit{Unit} \\
\mathit{fromList}\ (\mathit{Cons}\ x\ xs) & = & \mathit{Inr}\ (x\ {:}{*}{:}\ xs) \\
\mathit{toList} & :: & \forall A\,.\,\mathit{List}^{\circ}\ A \longrightarrow \mathit{List}\ A \\
\mathit{toList}\ (\mathit{Inl}\ \mathit{Unit}) & = & \mathit{Nil} \\
\mathit{toList}\ (\mathit{Inr}\ (x\ {:}{*}{:}\ xs)) & = & \mathit{Cons}\ x\ xs
\end{array}
$$

Embedding-projection maps

The conversion functions must be applied at the appropriate places.

Take as examples:

$$\begin{aligned}\textbf{type}\ \mathit{GShows} &= \Lambda T\,.\,T \to \mathit{String} \to \mathit{String}\\ \textbf{type}\ \mathit{GReads} &= \Lambda T\,.\,\mathit{String} \to [(T, \mathit{String})].\end{aligned}$$

We have to convert a function of type $\mathit{GShows}\ (\mathit{List}^{\circ}\ A)$ to a function of type $\mathit{GShows}\ (\mathit{List}\ A)$ and a function of type $\mathit{GReads}\ (\mathit{List}^{\circ}\ A)$ to a function of type $\mathit{GReads}\ (\mathit{List}\ A)$.

That's exactly what a mapping function is good for.

Embedding-projection maps

We need functions that convert back and fro (the operators '$+$', '$*$', '$\rightarrow$' denote the 'ordinary' mapping functions).

$$
\begin{array}{lcl}
\textbf{data}\ EP\ A_1\ A_2 & = & EP\{\mathit{from} :: A_1 \rightarrow A_2, \mathit{to} :: A_2 \rightarrow A_1\} \\
id_E & :: & \forall A\,.\, EP\ A\ A \\
id_E & = & EP\{\mathit{from} = id, \mathit{to} = id\} \\
(+_E) & :: & \forall A_1\ A_2\,.\, EP\ A_1\ A_2 \rightarrow \forall B_1\ B_2\,.\, EP\ B_1\ B_2 \\
 & & \quad \rightarrow EP\ (A_1 :+: B_1)\ (A_2 :+: B_2) \\
f +_E g & = & EP\{\mathit{from} = \mathit{from}\ f + \mathit{from}\ g, \mathit{to} = \mathit{to}\ f + \mathit{to}\ g\} \\
(*_E) & :: & \forall A_1\ A_2\,.\, EP\ A_1\ A_2 \rightarrow \forall B_1\ B_2\,.\, EP\ B_1\ B_2 \\
 & & \quad \rightarrow EP\ (A_1 :*: B_1)\ (A_2 :*: B_2) \\
f *_E g & = & EP\{\mathit{from} = \mathit{from}\ f * \mathit{from}\ g, \mathit{to} = \mathit{to}\ f * \mathit{to}\ g\} \\
(\rightarrow_E) & :: & \forall A_1\ A_2\,.\, EP\ A_1\ A_2 \rightarrow \forall B_1\ B_2\,.\, EP\ B_1\ B_2 \\
 & & \quad \rightarrow EP\ (A_1 \rightarrow B_1)\ (A_2 \rightarrow B_2) \\
f \rightarrow_E g & = & EP\{\mathit{from} = \mathit{to}\ f \rightarrow \mathit{from}\ g, \mathit{to} = \mathit{from}\ f \rightarrow \mathit{to}\ g\}
\end{array}
$$

Embedding-projection maps

$$
\begin{array}{lcl}
MapE\{\![*]\!\}\ T_1\ T_2 & = & EP\ T_1\ T_2 \\
MapE\{\![\mathfrak{T} \to \mathfrak{U}]\!\}\ T_1\ T_2 & = & \forall A_1\ A_2 \,.\, MapE\{\![\mathfrak{T}]\!\}\ A_1\ A_2 \\
 & & \qquad\qquad \to MapE\{\![\mathfrak{U}]\!\}\ (T_1\ A_1)\ (T_2\ A_2) \\
mapE\{\!|\, T :: \mathfrak{T} \,|\!\} & :: & MapE\{\![\mathfrak{T}]\!\}\ T\ T \\
mapE\{\!|\, Char \,|\!\} & = & id_E \\
mapE\{\!|\, Int \,|\!\} & = & id_E \\
mapE\{\!|\, Unit \,|\!\} & = & id_E \\
mapE\{\!|\, {:}{+}{:} \,|\!\}\ mA\ mB & = & mA +_E mB \\
mapE\{\!|\, {:}{*}{:} \,|\!\}\ mA\ mB & = & mA *_E mB \\
mapE\{\!|\, \to \,|\!\}\ mA\ mB & = & mA \to_E mB
\end{array}
$$

The grand final

$$
\begin{aligned}
convList &:: \forall A\,.\,EP\ (List\ A)\ (List^{\circ}\ A) \\
convList &= EP\{from = fromList, to = toList\}
\end{aligned}
$$

$$
\begin{aligned}
gshows\{\!|List|\!\}\ sa &= mapE\{\!|GShows|\!\}\ convList\ (gshows\{\!|List^{\circ}|\!\}\ sa) \\
greads\{\!|List|\!\}\ sa &= mapE\{\!|GReads|\!\}\ convList\ (greads\{\!|List^{\circ}|\!\}\ sa)
\end{aligned}
$$

NB. Of course, $mapE\{\!|Poly|\!\}$ has to be generated 'by hand'.

Stocktaking

- Generic Haskell takes a transformational approach: a generic function is translated into a family of polymorphic functions.
- Specialization can be seen as an interpretation of type terms.
- Adapting the techniques to Haskell involves systematic application of representation changers.
- The basic proof method of the simply typed lambda calculus, based on so-called logical relations, can be used to show properties of generic functions.

Concluding remarks

- Generic programming considerably adds to the expressive power of polymorphic type systems.
- A generic program can be made to work for all types of all kinds. A type-indexed value is assigned a kind-indexed type.
- Generic Haskell is a full implementation of the theory. Moreover, it offers several extensions: access to constructor names, generic abstractions etc.