Generic data types — Or: Know your school math

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

September, 2004

(Pick the slides at .../~ralf/talks.html#T36.)

Generic programming

Generic programming is a matter of making programs more adaptable by making them more general. It consists of allowing a wider range of entities as parameters than is available in more traditional programming languages.

Datatype-generic programming is an instantiation of the idea of generic programming: it allows programs to be parameterised by a data type or a type functor.

The purpose of this talk is to show that this idea not only applies to algorithms but also to data structures.

Prerequisites

Some basic knowledge of functional programming languages such as Haskell or ML is most useful. I will use a Haskell-like language throughout.

Example: binary digits, binary strings, and string concatenation.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{Bit} &&= 0 + 1\\
&\mathbf{data}\ \mathit{String} &&= \mathit{Nil} + \mathit{Cons}\ (\mathit{Bit} \times \mathit{String})\\
&(+\!\!+) &&:: \mathit{String} \to \mathit{String} \to \mathit{String}\\
&\mathit{Nil} +\!\!+ s_2 &&= s_2\\
&\mathit{Cons}\ (b, s_1) +\!\!+ s_2 &&= \mathit{Cons}\ (b, s_1 +\!\!+ s_2)
\end{aligned}
$$

Polymorphic type systems

Polymorphic type systems combine security (well-typed programs cannot 'go wrong') with flexibility (polymorphism allows the definition of functions that behave uniformly over all types).

However, polymorphic type systems are sometimes less flexible than one would wish: for instance, it is not possible to define a polymorphic compress function that works for all types.

$$compress :: \forall \alpha \,.\, \alpha \rightarrow String$$

Parametricity implies that a function of this type must necessarily be constant: the function is insensitive to what type the values of the first argument are.

Generic programming

Generic programming saves the day.

Idea: define the compress function by induction on the structure of types (for emphasis the type argument is enclosed in angle brackets).

$$
\begin{array}{lll}
compress\langle \tau :: \star \rangle & & :: \tau \to String \\
compress\langle 1 \rangle & () & = Nil \\
compress\langle \alpha + \beta \rangle & (Inl\ a) & = Cons\ (0, compress\langle \alpha \rangle\ a) \\
compress\langle \alpha + \beta \rangle & (Inr\ b) & = Cons\ (1, compress\langle \beta \rangle\ b) \\
compress\langle \alpha \times \beta \rangle & (a, b) & = compress\langle \alpha \rangle\ a +\!\!+ compress\langle \beta \rangle\ b
\end{array}
$$

This simple definition contains all the ingredients needed to derive specialisations for compressing elements of arbitrary data types (assuming that a data type is a sum of products as in Haskell).

Overview

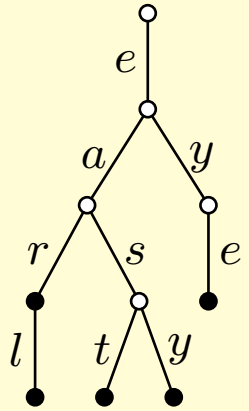
- ✘ Digital search trees (7–20)
- ✘ Trees with a focus (22–33)

Digital search trees

Digital search trees, also known as tries, employ the structure of search keys to organise information.

Tries were originally devised to represent sets of strings (A. Thue, Über die gegenseitige lage gleicher teile gewisser zeichenreihen, 1912).

$\{ear, earl, east, easy, eye\} \Longrightarrow$



Task

Digital search trees can also be used to implement finite maps (aka dictionaries, look-up tables).

☞ Finite maps are a wildly used abstraction. Digital search trees feature access that is proportional to the size of the key; they are superior to ordinary search trees and hash tables.

We are seeking a generic definition, that is, one that works for arbitrary key types.

A concrete instance: binary strings

Let us first consider two special instances.

$$
\begin{array}{lll}
\textbf{data}\ \mathit{MapBit} & :: \star \to \star & \\
\mathit{lookupBit} & :: \forall \nu\,.\,\mathit{Bit} \to \mathit{MapBit}\ \nu \to \mathit{Maybe}\ \nu \\
\textbf{data}\ \mathit{MapString} & :: \star \to \star & \\
\mathit{lookupString} & :: \forall \nu\,.\,\mathit{String} \to \mathit{MapString}\ \nu \to \mathit{Maybe}\ \nu
\end{array}
$$

The data type $\mathit{MapString}\ V$ represents the set of finite maps from String to V, that is, $\mathit{String} \to_{\mathrm{fin}} V$.

The *Maybe* data type

The mathematical treatment of finite maps usually assumes that V contains a distinguished element $\bullet$. Then a finite map is a function that sends only a finite number of keys to a value different from $\bullet$.

To avoid this restrictive assumption we adjoin a distinguished element using Haskell's *Maybe* data type.

$$\mathbf{data}\ \mathit{Maybe}\ \alpha = \mathit{Nothing} \mid \mathit{Just}\ \alpha$$

Forward composition of maps:

$$\begin{array}{l}(\diamond) \qquad :: (\alpha \to \mathit{Maybe}\ \beta) \to (\beta \to \mathit{Maybe}\ \gamma) \to (\alpha \to \mathit{Maybe}\ \gamma) \\ (f \diamond g)\ a = \mathbf{case}\ f\ a\ \mathbf{of}\ \{\mathit{Nothing} \to \mathit{Nothing}; \mathit{Just}\ b \to g\ b\}\end{array}$$

Implementation of $MapBit$

Finite maps over bits are simply pairs:

$$\begin{array}{l}\textbf{data}\ MapBit\ \nu = NodeBit\ (Maybe\ \nu \times Maybe\ \nu)\\ lookupBit\ \texttt{0}\ (NodeBit\ (t0, t1)) = t0\\ lookupBit\ \texttt{1}\ (NodeBit\ (t0, t1)) = t1\end{array}$$

Implementation of $MapString$

Finite maps over binary strings are compositions of finite maps:

$$
\begin{aligned}
&\textbf{data}\ MapString\ \nu = NodeString\ (Maybe\ \nu && \text{-- } Nil\\
&\qquad\qquad \times MapBit\ (MapString\ \nu)) && \text{-- } Cons\\
&lookupString\ Nil\ \ (NodeString\ (tn, tc))\\
&\quad = tn\\
&lookupString\ (Cons\ b\ s)\ (NodeString\ (tn, tc))\\
&\quad = (lookupBit\ b \diamond lookupString\ s)\ tc
\end{aligned}
$$

☞ Unfolding the definition of $MapBit$ yields the familiar data structure of binary tries.

Some school math

Digital search trees are based on the laws of exponentials.

$$
\begin{array}{ll}
1 & \to_{\mathrm{fin}} \nu \cong \nu \\
(\kappa_1 + \kappa_2) & \to_{\mathrm{fin}} \nu \cong (\kappa_1 \to_{\mathrm{fin}} \nu) \times (\kappa_2 \to_{\mathrm{fin}} \nu) \\
(\kappa_1 \times \kappa_2) & \to_{\mathrm{fin}} \nu \cong \kappa_1 \to_{\mathrm{fin}} (\kappa_2 \to_{\mathrm{fin}} \nu)
\end{array}
$$

This observation is due to Wadsworth (R.H. Connelly and F.L. Morris, A generalisation of the trie data structure, 1995).

Generic finite maps

$$
\begin{array}{ll}
1 & \to_{\text{fin}} \nu \cong \nu \\
(\kappa_1 + \kappa_2) & \to_{\text{fin}} \nu \cong (\kappa_1 \to_{\text{fin}} \nu) \times (\kappa_2 \to_{\text{fin}} \nu) \\
(\kappa_1 \times \kappa_2) & \to_{\text{fin}} \nu \cong \kappa_1 \to_{\text{fin}} (\kappa_2 \to_{\text{fin}} \nu)
\end{array}
$$

Using the laws of exponentials we can define a generic type of finite maps: $Map\langle K\rangle\ V$ represents $K \to_{\text{fin}} V$.

$$
\begin{array}{lll}
\textbf{data}\ Map\langle \kappa :: \star\rangle & :: \star \to \star & \\
\textbf{data}\ Map\langle 1\rangle & \nu = Maybe\ \nu & \\
\textbf{data}\ Map\langle \alpha + \beta\rangle & \nu = Map\langle \alpha\rangle\ \nu \times Map\langle \beta\rangle\ \nu & \\
\textbf{data}\ Map\langle \alpha \times \beta\rangle & \nu = Map\langle \alpha\rangle\ (Map\langle \beta\rangle\ \nu) &
\end{array}
$$

☞ The two type arguments of Map play different rôles: $Map\langle K\rangle\ V$ is defined by induction on the structure of K, but is parametric in V.

Generic finite maps

$$
\begin{aligned}
&\textbf{data } Map\langle 1\rangle \qquad\quad \nu = Maybe\ \nu\\
&\textbf{data } Map\langle \alpha+\beta\rangle\ \nu = Map\langle\alpha\rangle\ \nu \times Map\langle\beta\rangle\ \nu\\
&\textbf{data } Map\langle \alpha\times\beta\rangle\ \nu = Map\langle\alpha\rangle\ (Map\langle\beta\rangle\ \nu)
\end{aligned}
$$

The definition of Map can be written more succinctly using a point-free style:

$$
\begin{aligned}
&\textbf{data } Map\langle 1\rangle \qquad\quad = Maybe\\
&\textbf{data } Map\langle \alpha+\beta\rangle = Map\langle\alpha\rangle \times Map\langle\beta\rangle\\
&\textbf{data } Map\langle \alpha\times\beta\rangle = Map\langle\alpha\rangle\ \cdot\ Map\langle\beta\rangle
\end{aligned}
$$

NB. '$\times$' denotes lifted products.

Generic look-up

$$
\begin{array}{lllll}
lookup\langle \kappa :: \star \rangle & & & :: \forall \nu \,.\, \kappa \to Map\langle \kappa \rangle\ \nu \to Maybe\ \nu \\
lookup\langle 1 \rangle & () & t & = t \\
lookup\langle \alpha + \beta \rangle & (Inl\ a) & (ta, tb) & = lookup\langle \alpha \rangle\ a\ ta \\
lookup\langle \alpha + \beta \rangle & (Inr\ b) & (ta, tb) & = lookup\langle \beta \rangle\ b\ tb \\
lookup\langle \alpha \times \beta \rangle & (a, b) & t & = (lookup\langle \alpha \rangle\ a \diamond lookup\langle \beta \rangle\ b)\ t
\end{array}
$$

Generic look-up

The definition of $lookup$ can be written more succinctly using a point-free style:

$$
\begin{array}{lll}
lookup\langle \kappa :: \star \rangle & & :: \forall \nu \,.\, \kappa \to Map\langle \kappa \rangle\ \nu \to Maybe\ \nu \\
lookup\langle 1 \rangle & () & = id \\
lookup\langle \alpha + \beta \rangle & (Inl\ a) & = lookup\langle \alpha \rangle\ a \cdot outl \\
lookup\langle \alpha + \beta \rangle & (Inr\ b) & = lookup\langle \beta \rangle\ b \cdot outr \\
lookup\langle \alpha \times \beta \rangle & (a, b) & = lookup\langle \alpha \rangle\ a \diamond lookup\langle \beta \rangle\ b
\end{array}
$$

Some instances

Let's pick the fruit and specialise Map to some data types.

☞ The generic definition can be specialised to arbitrary data types (R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types, 2004). Perhaps surprisingly, the specialisation also works for parameterised data types:

$$\mathbf{data}\ Pair\ \alpha = Pair\ (\alpha \times \alpha)$$

The trie for $Pair\ \alpha$ is parameterised by the trie for α:

$$\begin{aligned}
&\mathbf{data}\ MapPair\ map\alpha\ \nu = NodePair\ (map\alpha\ (map\alpha\ \nu)) \\
&lookupPair :: \forall\alpha\ map\alpha\ .\ (\forall\nu\ .\ \alpha \to map\alpha\ \nu \to Maybe\ \nu) \\
&\qquad\qquad \to (\forall\nu\ .\ Pair\ \alpha \to MapPair\ map\alpha\ \nu \to Maybe\ \nu) \\
&lookupPair\ lookup\alpha\ (Pair\ (a_1, a_2))\ (NodePair\ t) \\
&\quad = (lookup\alpha\ a_1 \diamond lookup\alpha\ a_2)\ t
\end{aligned}$$

Some instances

Specialising parameterised recursive data types works, as well.

$$\mathbf{data}\ \mathit{Tree}\ \alpha = \mathit{Empty} + \mathit{Node}\ (\mathit{Pair}\ (\mathit{Tree}\ \alpha))$$

The trie is recursive where the key type is:

$$\begin{aligned}
&\mathbf{data}\ \mathit{MapTree}\ \mathit{map}\alpha\ \nu = \mathit{NodeTree}\ (\mathit{Maybe}\ \nu \\
&\qquad\qquad \times \mathit{MapPair}\ (\mathit{MapTree}\ \mathit{map}\alpha)\ \nu) \\
&\mathit{lookupTree} :: \forall\alpha\ \mathit{map}\alpha\ .\ (\forall\nu\ .\ \alpha \to \mathit{map}\alpha\ \nu \to \mathit{Maybe}\ \nu) \\
&\qquad\qquad \to (\forall\nu\ .\ \mathit{Tree}\ \alpha \to \mathit{MapTree}\ \mathit{map}\alpha\ \nu \to \mathit{Maybe}\ \nu) \\
&\mathit{lookupTree}\ \mathit{lookup}\alpha\ \mathit{Empty}\quad (\mathit{NodeTree}\ (te, tn)) \\
&\quad = te \\
&\mathit{lookupTree}\ \mathit{lookup}\alpha\ (\mathit{Node}\ p)\ (\mathit{NodeTree}\ (te, tn)) \\
&\quad = \mathit{lookupPair}\ (\mathit{lookupTree}\ \mathit{lookup}\alpha)\ p\ tn
\end{aligned}$$

Some instances

Finally, we can derive tries for so-called nested data types.

$$\mathbf{data}\ Pow\ \alpha = Zero\ \alpha + Succ\ (Pow\ (Pair\ \alpha))$$

The trie is nested, as well:

$$\begin{aligned}
&\mathbf{data}\ MapPow\ map\alpha\ \nu = NodePow\ (map\alpha\ \nu \\
&\qquad\qquad\qquad\qquad\qquad\qquad \times MapPow\ (MapPair\ map\alpha)\ \nu) \\
&lookupPow :: \forall\alpha\ map\alpha\ .\ (\forall\nu\ .\ \alpha \to map\alpha\ \nu \to Maybe\ \nu) \\
&\qquad\qquad\qquad\qquad \to (\forall\nu\ .\ Pow\ \alpha \to MapPow\ map\alpha\ \nu \to Maybe\ \nu) \\
&lookupPow\ lookup\alpha\ (Zero\ a)\ (NodePow\ (tz, ts)) \\
&\quad = lookup\alpha\ a\ tz \\
&lookupPow\ lookup\alpha\ (Succ\ p)\ (NodePow\ (tz, ts)) \\
&\quad = lookupPow\ (lookupPair\ lookup\alpha)\ p\ ts
\end{aligned}$$

☞ $MapPow$ is a stream-like data structure; $lookupPow$ is tail-recursive.

Overview

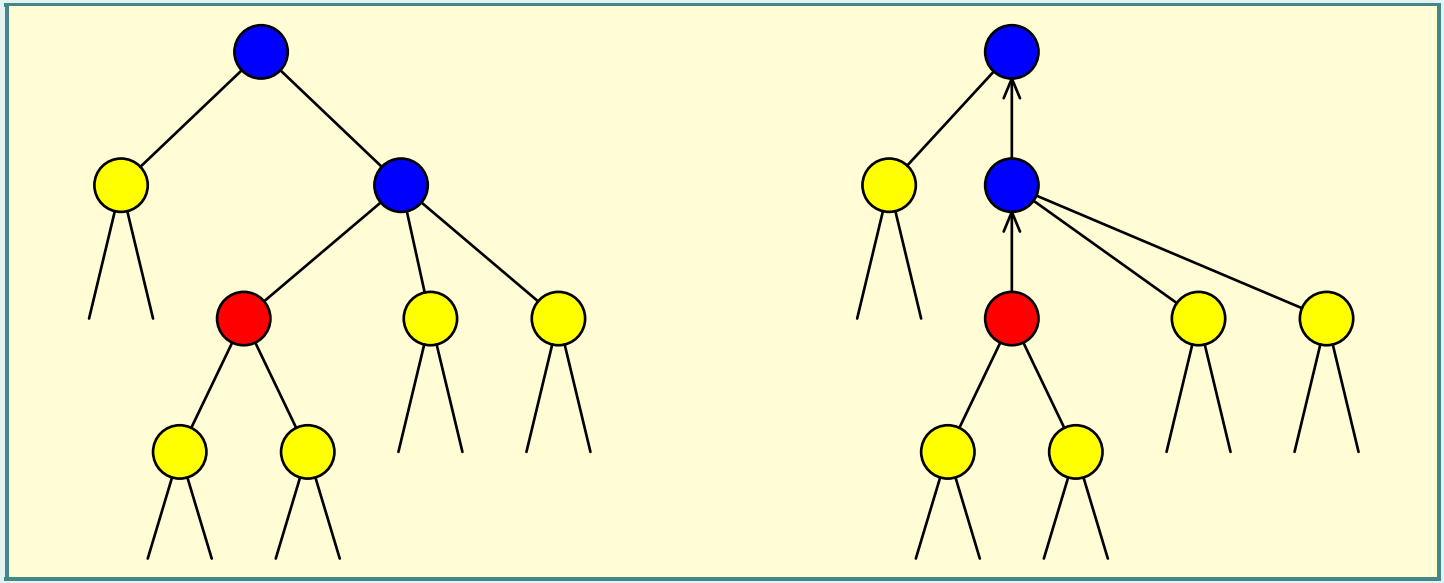
✔ Digital search trees (7–20)

✘ Trees with a focus (22–33)

Task

Represent a tree together with a focus of interest (zipper, finger, pointer reversal).

☞ Focused trees have a variety of applications: structured editors, theorem provers etc.



Again, we are seeking a generic definition, that is, one that works for arbitrary data types.

A concrete instance: 2-3 trees

$$\begin{aligned}\textbf{data}\ \mathit{Tree23} = {} & \mathit{Empty} \\ & + \mathit{Node2}\ (\mathit{Tree23} \times \mathit{Int} \times \mathit{Tree23}) \\ & + \mathit{Node3}\ (\mathit{Tree23} \times \mathit{Int} \times \mathit{Tree23} \times \mathit{Int} \times \mathit{Tree23})\end{aligned}$$

A concrete instance: 2-3 trees

A 2-3 tree with a focus of interest consists of the focused tree and a path leading to the root.

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Focus23} & = & \mathit{Path23} \times \mathit{Tree23} \\
\textbf{data}\ \mathit{Path23} & = & \mathit{Top} + \mathit{Step}\ (\mathit{Path23} \times \mathit{Seg23}) \\
\textbf{data}\ \mathit{Seg23} & = & \mathit{Node2}_1\ (\bullet \times \mathit{Int} \times \mathit{Tree23}) \\
 & & + \mathit{Node2}_2\ (\mathit{Tree23} \times \mathit{Int} \times \bullet) \\
 & & + \mathit{Node3}_1\ (\bullet \times \mathit{Int} \times \mathit{Tree23} \times \mathit{Int} \times \mathit{Tree23}) \\
 & & + \mathit{Node3}_2\ (\mathit{Tree23} \times \mathit{Int} \times \bullet \times \mathit{Int} \times \mathit{Tree23}) \\
 & & + \mathit{Node3}_3\ (\mathit{Tree23} \times \mathit{Int} \times \mathit{Tree23} \times \mathit{Int} \times \bullet)
\end{array}
$$

☞ $\mathit{Path23}$ is a snoc list of $\mathit{Seg23}$s.

NB. $\bullet$ is the unit type (representing a hole).

Operations

Moving up a 2-3 tree:

$$
\begin{aligned}
&up && :: Focus23 \to Focus23 \\
&up\ (Top, && t) = (Top, t) \\
&up\ (Step\ (p, s), t) &&= (p, \quad plugin\ (s, t)) \\
&plugin && :: Seg23 \times Tree23 \to Tree23 \\
&plugin\ (Node2_1\ (\bullet, a, r), && t) = Node2\ (t, a, r) \\
&plugin\ (Node2_2\ (l, a, \bullet), && t) = Node2\ (l, a, t) \\
&plugin\ (Node3_1\ (\bullet, a, m, b, r), t) &&= Node3\ (t, a, m, b, r) \\
&plugin\ (Node3_2\ (l, a, \bullet, b, r), t) &&= Node3\ (l, a, t, \ b, r) \\
&plugin\ (Node3_3\ (l, a, m, b, \bullet), t) &&= Node3\ (l, a, m, b, t)
\end{aligned}
$$

Making recursive components explicit

We write the type $Tree23$ as a fixed point of a functor, the so-called base functor, using a point-free style.

$$
\begin{aligned}
Tree23 &= Fix\ Base23 \\
Base23 &= Empty \\
&\quad + Node2\ (Id \times K\ Int \times Id) \\
&\quad + Node3\ (Id \times K\ Int \times Id \times K\ Int \times Id)
\end{aligned}
$$

NB. $K\ T$ is the constant functor, Id is the identity functor, and '$+$' and '$\times$' denote lifted sums and products.

The fixed point operator, Fix, is given by

$$\mathbf{data}\ Fix\ \phi = In\ (\phi\ (Fix\ \phi))$$

Making recursive components explicit

$$
\begin{aligned}
Focus23 &= Path23 \times Tree23 \\
Path23 &= Top + Step\ (Path23 \times Seg23)
\end{aligned}
$$

$$
\begin{aligned}
Seg23 &= Base23'\ Tree23 \\
Base23' &= Node2_1\ (K\ \bullet \times K\ Int \times Id) \\
&\quad + Node2_2\ (Id\ \ \times K\ Int \times K\ \bullet) \\
&\quad + Node3_1\ (K\ \bullet \times K\ Int \times Id\ \ \times K\ Int \times Id) \\
&\quad + Node3_2\ (Id\ \ \times K\ Int \times K\ \bullet \times K\ Int \times Id) \\
&\quad + Node3_3\ (Id\ \ \times K\ Int \times Id\ \ \times K\ Int \times K\ \bullet)
\end{aligned}
$$

Generic paths and segments

Let T be the fixed point of F, that is, $T = Fix\ F$. We parameterise the generic types by the base functor F.

$$
\begin{array}{lll}
Focus\ F & = Path\ F \times Fix\ F & \text{-- } = Path\ F \times T \\
Path\ F & = Top + Step\ (Path\ F \times Seg\ F) & \\
Seg\ F & = F'\ (Fix\ F) &
\end{array}
$$

☞ Now, what is the relationship between F and F'?

More school math

The functor F' is the derivative of the base functor F. We define F' by induction on the structure of F.

$$
\begin{array}{lll}
(F :: \star \to \star)' & :: \star \to \star & \\
(K\ C)' & = K\ 0 & \\
Id' & = K\ 1 & \text{-- } = K\ \bullet \\
(F_1 + F_2)' & = F_1' + F_2' & \\
(F_1 \times F_2)' & = F_1' \times F_2 + F_1 \times F_2' &
\end{array}
$$

Since F is a functor, we must distinguish 4 cases rather than 3.

☞ The observation that a one-point context corresponds to the derivative of a functor is due to McBride; the definition above was given independently by Hinze and Jeuring.

Examples of derivatives

$$
\begin{aligned}
(Id + Id)' &= K\ 1 + K\ 1 \\
(K\ n \times Id)' &\cong K\ n \\
(Id \times Id)' &= K\ 1 \times Id + Id \times K\ 1 \\
(Id^n)' &\cong K\ n \times Id^{n-1}
\end{aligned}
$$

The derivative of the list functor is a pair of lists: the prefix and the suffix of the hole.

$$
\begin{aligned}
List &= K\ 1 + Id \times List \\
List' &= K\ 0 + (K\ 1 \times List + Id \times List') \\
List' &\cong List \times List
\end{aligned}
$$

The chain rule

One can show that $(-)'$ satisfies the chain rule, which we all know and love:

$$(F \cdot G)' \cong F' \cdot G \times G'$$

The chain rule states, that the derivative of a composition of two functors is the derivative of the outer functor (composed with the inner functor) times the derivative of the inner functor.

More examples of derivatives

$$(List \cdot List)' \cong List' \cdot List \times List'$$
$$(List \cdot List)' \cong List^2 \cdot List \times List^2$$

The chain rule is convenient for calculating the derivatives of nested data types.

$$\begin{aligned} Pair &= Id \times Id \\ Pow &= Id + Pow \cdot Pair \\ Pow' &\cong K\ 1 + Pow' \cdot Pair \times K\ 2 \times Id \\ Pow' &\cong K\ 1 + Pow' \cdot Pair \times Id + Pow' \cdot Pair \times Id \end{aligned}$$

Generic operations

$$
\begin{array}{lll}
up\langle F :: \star \to \star \rangle & & :: Focus\ F \to Focus\ F \\
up\langle F \rangle\ (Top, & t) & = (Top, t) \\
up\langle F \rangle\ (Step\ (p, s), & t) & = (p, In\ (plugin\langle F \rangle\ (s, t))) \\
plugin\langle F :: \star \to \star \rangle & & :: \forall \alpha\ .\ F'\ \alpha \times \alpha \to F\ \alpha \\
plugin\langle Id \rangle\ (\bullet, & t) & = t \\
plugin\langle F_1 + F_2 \rangle\ (Inl\ s_1, & t) & = Inl\ (plugin\langle F_1 \rangle\ (s_1, t)) \\
plugin\langle F_1 + F_2 \rangle\ (Inr\ s_2, & t) & = Inr\ (plugin\langle F_2 \rangle\ (s_2, t)) \\
plugin\langle F_1 \times F_2 \rangle\ (Inl\ (s, r), & t) & = (plugin\langle F_1 \rangle\ (s, t), r) \\
plugin\langle F_1 \times F_2 \rangle\ (Inr\ (l, s), & t) & = (l, plugin\langle F_2 \rangle\ (s, t))
\end{array}
$$

NB. We need not define $plugin\langle K\ C \rangle$ as $(K\ C)' = K\ 0$.

Overview

- ✔ Digital search trees (7–20)
- ✔ Trees with a focus (22–33)

Conclusion

- Generic functions and data types are defined by induction on the structure of types.
- Generic definitions can be specialised to arbitrary data types.
- Generic programming, albeit more abstract, is often simpler than ordinary programming (because we only have to provide instances for three simple, non-recursive data types).
- Generalisations of textbook data structures reveal familiar mathematical structures.