The Fun of Programming

Oege de Moor,
Jeremy Gibbons
and Geraint Jones

macmillan.eps

Contents

Fun with phantom types

R. Hinze

Haskell is renowned for its many extensions to the Hindley-Milner type system (type classes, polymorphic recursion, rank-n types, existential types, functional dependencies—just to name a few). In this chapter we look at yet another extension. I can hear you groaning but this is quite a mild extension and one that fits nicely within the Hindley-Milner framework. Of course, whenever you add a new feature to a language, you should throw out an existing one (especially if the language at hand is named after a logician). Now, for this chapter we abandon type classes—judge for yourself how well we get along without Haskell's most beloved feature.

1 Introducing phantom types

Suppose you want to embed a programming language, say, a simple expression language in Haskell. Since you are a firm believer of *static typing*, you would like your embedded language to be statically typed, as well. This requirement rules out a simple *Term* data type as this choice would allow us to freely mix terms of different types. The next idea is to parameterize the *Term* type so that *Term t* comprises only terms of type t. The different compartments of *Term* are then inhabited by declaring *constructors* of the appropriate types (we confine ourselves to a few basic operations):

$$\begin{array}{lcl}
\mathit{Zero} & :: & \mathit{Term}\ \mathit{Int} \\
\mathit{Succ}, \mathit{Pred} & :: & \mathit{Term}\ \mathit{Int} \to \mathit{Term}\ \mathit{Int} \\
\mathit{IsZero} & :: & \mathit{Term}\ \mathit{Int} \to \mathit{Term}\ \mathit{Bool} \\
\mathit{If} & :: & \forall a \,.\, \mathit{Term}\ \mathit{Bool} \to \mathit{Term}\ a \to \mathit{Term}\ a \to \mathit{Term}\ a.
\end{array}$$

The types are essentially those of the corresponding Haskell functions except that every argument and every result type has *Term* wrapped around it. For instance, the Haskell function $\mathit{succ} :: \mathit{Int} \to \mathit{Int}$ corresponds to the constructor $\mathit{Succ} :: \mathit{Term}\ \mathit{Int} \to \mathit{Term}\ \mathit{Int}$.

This term representation meets the typing requirement: we can apply *Succ* only to an arithmetic expression; applying *Succ* to a Boolean expression results in a type error. Unfortunately, the above signature cannot be translated into a **data** declaration (Haskell's linguistic construct for introducing constructors). The reason is simply that all constructors of a data type must share the same result type, namely, the declared type on the left-hand side. Thus, we can assign *Zero* the type *Term t* but not *Term Int*. Of course, using the first type would defeat the purpose of the whole exercise. The only constructor that fits into the scheme is *If*, which has the desired general result type.

If only we had the means to constrain the type argument of *Term* to a certain type. Now, this is exactly what the aforementioned 'mild' extension allows us to do. Given this extension we declare the *Term* data type as follows:

$$
\begin{array}{lcll}
\textbf{data}\ \mathit{Term}\ t & = & \mathit{Zero} & \textbf{with}\ t = \mathit{Int} \\
 & \mid & \mathit{Succ}\ (\mathit{Term}\ \mathit{Int}) & \textbf{with}\ t = \mathit{Int} \\
 & \mid & \mathit{Pred}\ (\mathit{Term}\ \mathit{Int}) & \textbf{with}\ t = \mathit{Int} \\
 & \mid & \mathit{IsZero}\ (\mathit{Term}\ \mathit{Int}) & \textbf{with}\ t = \mathit{Bool} \\
 & \mid & \mathit{If}\ (\mathit{Term}\ \mathit{Bool})\ (\mathit{Term}\ a)\ (\mathit{Term}\ a) & \textbf{with}\ t = a.
\end{array}
$$

The **with** clause that it attached to each constructor records its type constraints. For instance, *Zero* has *Type t* with the additional constraint $t = \mathit{Int}$. Note that the **with** clause of the *If* constructor is not strictly necessary. We could have simply replaced a by t. Its main purpose is to illustrate that the type equation may contain type variables that do not appear on the left-hand side of the declaration. These variables can be seen as being *existentially quantified*.

Let us move on to defining an interpreter for the expression language. The interpreter takes an expression of type *Term t* to a value of type t. The definition proceeds by straightforward structural recursion.

$$
\begin{array}{lcl}
\mathit{eval} & :: & \forall t\,.\ \mathit{Term}\ t \to t \\
\mathit{eval}\ (\mathit{Zero}) & = & 0 \\
\mathit{eval}\ (\mathit{Succ}\ e) & = & \mathit{eval}\ e + 1 \\
\mathit{eval}\ (\mathit{Pred}\ e) & = & \mathit{eval}\ e - 1 \\
\mathit{eval}\ (\mathit{IsZero}\ e) & = & \mathit{eval}\ e == 0 \\
\mathit{eval}\ (\mathit{If}\ e_1\ e_2\ e_3) & = & \textbf{if}\ \mathit{eval}\ e_1\ \textbf{then}\ \mathit{eval}\ e_2\ \textbf{else}\ \mathit{eval}\ e_3
\end{array}
$$

Even though *eval* is assigned the type $\forall t\,.\ \mathit{Term}\ t \to t$, each equation—with the notable exception of the last one—has a more specific type as dictated by the type constraints. As an example, the first equation has type $\mathit{Term}\ \mathit{Int} \to \mathit{Int}$ as *Zero* constrains t to *Int*.

The interpreter is quite noticeable in that it is *tag free*. If it receives a Boolean expression, then it returns a Boolean. By contrast, a more conventional interpreter of type $\mathit{Term} \to \mathit{Val}$ has to inject the Boolean into the *Val* data type. Conversely, when evaluating a conditional it has to untag the evaluated condition and furthermore it has to check whether the value is actually a Boolean. To make a long story short, we are experiencing the benefits of static typing. Here is a short interactive session that shows the interpreter in action (:**type** displays the type of an expression).

$Main\rangle$ **let** $one = Succ\ Zero$
$Main\rangle$:**type** one
$Term\ Int$
$Main\rangle$ $eval\ one$
1
$Main\rangle$ $eval\ (IsZero\ one)$
$False$
$Main\rangle$ $IsZero\ (IsZero\ one)$
`Type error: couldn't match 'Int' against 'Bool'`
$Main\rangle$ $eval\ (If\ (IsZero\ one)\ Zero\ one)$
1
$Main\rangle$ **let** $true = IsZero\ Zero$
$Main\rangle$ **let** $false = IsZero\ one$
$Main\rangle$ $eval\ (If\ true\ true\ false)$
$True$

Thinking of it, the type $Term\ t$ is quite unusual. Though $Term$ is parameterized, it is not a container type: an element of $Term\ Int$, for instance, is an expression that evaluates to an integer; it is not a data structure that contains integers. This means, in particular, that we cannot define a mapping function $(a \to b) \to (Term\ a \to Term\ b)$ as for many other data types. How could we possibly turn expressions of type $Term\ a$ into expression of type $Term\ b$? The type $Term\ b$ might not even be inhabited: there are, for instance, no terms of type $Term\ String$. Clearly, types of this characteristic deserve a special name. Since the type argument of $Term$ is not related to any component, we call $Term$ a *phantom type*. The purpose of this chapter is to demonstrate the usefulness and the beauty of phantom types.

Exercise 1 (Language design) Whenever we define a function that involves a phantom type, we will provide an explicit type signature. Can you imagine why? *Hint:* is it possible to infer the types of functions that involve phantom types? □

Exercise 2 (Language design) In Haskell, constructors are introduced via **data** declarations. An alternative is to abandon the **data** construct and to introduce constructors simply by listing their signatures. Discuss the pros and cons of the two alternatives. □

2 Generic functions

Suppose you are developing an application where the need arises to compress data to strings of bits. As it happens, you have data of many different types and you want to program a compression function that works for all of these types. This sounds like a typical case for Haskell's type classes. Alas, I promised to do without type classes. Fortunately, phantom types offer an intriguing alternative.

The basic idea is to define a type whose elements represent types. For concreteness, assume that we need compressing functions for types built from *Int* and *Char* using the list and the pair type constructor.

$$
\begin{array}{lcll}
\mathbf{data}\ \mathit{Type}\ t & = & \mathit{RInt} & \mathbf{with}\ t = \mathit{Int} \\
 & \mid & \mathit{RChar} & \mathbf{with}\ t = \mathit{Char} \\
 & \mid & \mathit{RList}\ (\mathit{Type}\ a) & \mathbf{with}\ t = [a] \\
 & \mid & \mathit{RPair}\ (\mathit{Type}\ a)\ (\mathit{Type}\ b) & \mathbf{with}\ t = (a, b) \\
\mathit{rString} & :: & \mathit{Type}\ \mathit{String} & \\
\mathit{rString} & = & \mathit{RList}\ \mathit{RChar} &
\end{array}
$$

An element *rt* of type *Type t* is a representation of *t*. For instance, *Int* is represented by *RInt*, the type (*String*, *Int*) is represented by *RPair rString RInt*.

Now, the compression function takes a type representation as a first argument and the to-be-compressed value as the second argument. The following interactive session illustrates the use of *compress* (note that integers require 32 bits and characters 7 bits).

```
Main⟩ :type compress RInt
Int → [Bit]
Main⟩ compress RInt 60
<00111100000000000000000000000000>
Main⟩ :type compress rString
[Char] → [Bit]
Main⟩ compress rString "Richard"
<1010010111001011111000111000101111000011101001111001001100>
```

The definition of *compress* itself is straightforward: it pattern matches on the type representation and then takes the appropriate action.

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Bit} & = & \mathtt{0} \mid \mathtt{1} \\
\mathit{compress} & :: & \forall t\,.\ \mathit{Type}\ t \to t \to [\mathit{Bit}] \\
\mathit{compress}\ (\mathit{RInt})\ i & = & \mathit{compressInt}\ i \\
\mathit{compress}\ (\mathit{RChar})\ c & = & \mathit{compressChar}\ c \\
\mathit{compress}\ (\mathit{RList}\ \mathit{ra})\ [\,] & = & \mathtt{0} : [\,] \\
\mathit{compress}\ (\mathit{RList}\ \mathit{ra})\ (a : \mathit{as}) & = & \mathtt{1} : \mathit{compress}\ \mathit{ra}\ a +\!\!+ \mathit{compress}\ (\mathit{RList}\ \mathit{ra})\ \mathit{as} \\
\mathit{compress}\ (\mathit{RPair}\ \mathit{ra}\ \mathit{rb})\ (a, b) & = & \mathit{compress}\ \mathit{ra}\ a +\!\!+ \mathit{compress}\ \mathit{rb}\ b
\end{array}
$$

We assume that $\mathit{compressInt} :: \mathit{Int} \to [\mathit{Bit}]$ and $\mathit{compressChar} :: \mathit{Char} \to [\mathit{Bit}]$ are given. Consider the definition of *compress* (*RList ra*). Since the list data type has two constructors, we emit one bit to distinguish between the two cases. In the case of a non-empty list, we recursively encode the head and the tail. As an aside, if we extend *compress* to data types with more than two constructors, we must ensure that the codes for the constructors have the *unique prefix property*, that is, no code is the prefix of another code. However, we *can* use the same code for constructors of different types as compression (as well as decompression) is driven by type.

We can view *Type* as representing a *family of types* and *compress* as implementing a *family of functions*. Through the first argument of *compress* we specify which member of the family we wish to apply. Functions that work for a family of types are commonly called *generic functions*. Using a phantom type of type representations, generic functions are easy to define. Typical examples of generic functions include equality and comparison functions, pretty printers and parsers. Actually, pretty printing is quite a nice example, so let us consider this next.

In Haskell, the *Show* class takes care of converting values into string representations. We will define a variant of its *show* method building upon the pretty-printing combinators of Chapter **??**. The implementation of the *Show* class is complicated by the desire to print lists of characters different from lists of other types: a list of characters is shown using string syntax whereas any other list is shown as a comma-separated sequence of elements enclosed in square brackets. Using type representations we can easily single out this special case by supplying an additional equation.

$$
\begin{array}{lcl}
\mathit{pretty} & :: & \forall t \, . \, \mathit{Type} \; t \to t \to \mathit{Doc} \\
\mathit{pretty} \; (\mathit{RInt}) \; i & = & \mathit{prettyInt} \; i \\
\mathit{pretty} \; (\mathit{RChar}) \; c & = & \mathit{prettyChar} \; c \\
\mathit{pretty} \; (\mathit{RList} \; \mathit{RChar}) \; s & = & \mathit{prettyString} \; s \\
\mathit{pretty} \; (\mathit{RList} \; \mathit{ra}) \; [\,] & = & \mathit{text} \; \texttt{"[]"} \\
\mathit{pretty} \; (\mathit{RList} \; \mathit{ra}) \; (a : \mathit{as}) & = & \mathit{block} \; 1 \; (\mathit{text} \; \texttt{"["} \; \langle\rangle \; \mathit{pretty} \; \mathit{ra} \; a \; \langle\rangle \; \mathit{prettyL} \; \mathit{as}) \\
\quad \textbf{where} \; \mathit{prettyL} \; [\,] & = & \mathit{text} \; \texttt{"]"} \\
\qquad\qquad \mathit{prettyL} \; (a : \mathit{as}) & = & \mathit{text} \; \texttt{","} \; \langle\rangle \; \mathit{line} \; \langle\rangle \; \mathit{pretty} \; \mathit{ra} \; a \; \langle\rangle \; \mathit{prettyL} \; \mathit{as} \\
\mathit{pretty} \; (\mathit{RPair} \; \mathit{ra} \; \mathit{rb}) \; (a, b) & = & \mathit{block} \; 1 \; (\mathit{text} \; \texttt{"("} \; \langle\rangle \; \mathit{pretty} \; \mathit{ra} \; a \; \langle\rangle \; \mathit{text} \; \texttt{","} \\
 & & \qquad\qquad \langle\rangle \; \mathit{line} \; \langle\rangle \; \mathit{pretty} \; \mathit{rb} \; b \; \langle\rangle \; \mathit{text} \; \texttt{")"}) \\
\mathit{block} & :: & \mathit{Int} \to \mathit{Doc} \to \mathit{Doc} \\
\mathit{block} \; i \; d & = & \mathit{group} \; (\mathit{nest} \; i \; d)
\end{array}
$$

Here, $\mathit{prettyInt} :: \mathit{Int} \to \mathit{Doc}$, $\mathit{prettyChar} :: \mathit{Int} \to \mathit{Doc}$, and $\mathit{prettyString} :: \mathit{String} \to \mathit{Doc}$ are predefined functions that pretty print integers, characters, and strings, respectively.

Exercise 3 Implement generic equality $\mathit{eq} :: \forall t \, . \, \mathit{Type} \; t \to t \to t \to \mathit{Bool}$ and a generic comparison function $\mathit{compare} :: \forall t \, . \, \mathit{Type} \; t \to t \to t \to \mathit{Ordering}$. □

Exercise 4 Families of type-indexed functions can be implemented either using type classes or using type representations. Discuss differences and commonalities of the two approaches. □

Exercise 5 Implement a function $\mathit{uncompress} :: \forall t \, . \, \mathit{Type} \; t \to [\mathit{Bit}] \to t$ that uncompresses a bit string. *Hint:* use tupling (see IFPH, Section 7.3). Implement a generic parser *parse* that converts a string to a value. The function *parse* should at least be able to read strings that were generated by *pretty*. □

3 Dynamic values

Even a programming language such as Haskell cannot guarantee the absence of run-time errors using static checks only. For instance, when we communicate with the environment, we have to check dynamically whether the imported values have the expected types. In this section we show how to embed dynamic checking in a statically typed language.

To this end we introduce a *universal data type*, the type *Dynamic*, which encompasses all static values (whose types are representable). To inject a static value into the universal type we bundle the value with a representation of its type.

$$\textbf{data}\ \mathit{Dynamic} \quad = \quad \mathit{Dyn}\ (\mathit{Type}\ t)\ t$$

It is important to note that the type variable t is existentially quantified: a dynamic value is a pair consisting of a type representation of *Type* t and a value of type t for *some* type t. The type *Dynamic* looks attractive but on a second thought we note a small deficiency: we can form a list of dynamic values but we cannot turn this list into a dynamic value itself, simply because the type *Dynamic* is not representable. This is, however, easily remedied: we simply add *Dynamic* to *Type* t.

$$\begin{array}{lcl}\textbf{data}\ \mathit{Type}\ t & = & \cdots \\ & | & \mathit{RDyn} \quad \textbf{with}\ t = \mathit{Dynamic}\end{array}$$

Note that *Type* and *Dynamic* are now defined by mutual recursion.

Dynamic values and generic functions go well together. In a sense, they are complementary concepts. It is not too difficult, for instance, to extend the generic functions of the previous section so that they also work for dynamic values (see Exercise 7 and 8): a dynamic value contains a type representation, which a generic function requires as a first argument. The following interactive session illustrates the use of dynamics and generics (note that the identifier *it* always refers to the previously evaluated expression).

$\mathit{Main}\rangle$ **let** $\mathit{ds} = [\mathit{Dyn}\ \mathit{RInt}\ 60, \mathit{Dyn}\ \mathit{rString}\ \texttt{"Bird"}]$
$\mathit{Main}\rangle$:**type** ds
$[\mathit{Dynamic}]$
$\mathit{Main}\rangle$ $\mathit{Dyn}\ (\mathit{RList}\ \mathit{RDyn})\ \mathit{ds}$
$\mathit{Dyn}\ (\mathit{RList}\ \mathit{RDyn})\ [\mathit{Dyn}\ \mathit{RInt}\ 60, \mathit{Dyn}\ (\mathit{RList}\ \mathit{RChar})\ \texttt{"Bird"}]$
$\mathit{Main}\rangle$ $\mathit{compress}\ \mathit{RDyn}\ \mathit{it}$
`<01010010000011110000000000000000000000000000010100011010000`
`1110010111010011110010011 00>`
$\mathit{Main}\rangle$ $\mathit{uncompress}\ \mathit{RDyn}\ \mathit{it}$
$\mathit{Dyn}\ (\mathit{RList}\ \mathit{RDyn})\ [\mathit{Dyn}\ \mathit{RInt}\ 60, \mathit{Dyn}\ (\mathit{RList}\ \mathit{RChar})\ \texttt{"Bird"}]$

By pairing a value with its type representation we turn a static into a dynamic value. The other way round involves a dynamic check. This operation, usually termed *cast*, takes a dynamic value and a type representation and checks whether the type representation of the dynamic value and the supplied one are identical. The equality check is defined

$$
\begin{array}{lcl}
\mathit{tequal} & :: & \forall t\ u\,.\,\mathit{Type}\ t \rightarrow \mathit{Type}\ u \rightarrow \mathit{Maybe}\ (t \rightarrow u)\\
\mathit{tequal}\ (\mathit{RInt})\ (\mathit{RInt}) & = & \mathit{return}\ \mathit{id}\\
\mathit{tequal}\ (\mathit{RChar})\ (\mathit{RChar}) & = & \mathit{return}\ \mathit{id}\\
\mathit{tequal}\ (\mathit{RList}\ \mathit{ra}_1)\ (\mathit{RList}\ \mathit{ra}_2) & = & \mathit{liftM}\ \mathit{list}\ (\mathit{tequal}\ \mathit{ra}_1\ \mathit{ra}_2)\\
\mathit{tequal}\ (\mathit{RPair}\ \mathit{ra}_1\ \mathit{rb}_1)\ (\mathit{RPair}\ \mathit{ra}_2\ \mathit{rb}_2) & & \\
\quad = \ \mathit{liftM2}\ \mathit{pair}\ (\mathit{tequal}\ \mathit{ra}_1\ \mathit{ra}_2)\ (\mathit{tequal}\ \mathit{rb}_1\ \mathit{rb}_2) & & \\
\mathit{tequal}\ \_\ \_ & = & \mathit{fail}\ \texttt{"cannot tequal"}.
\end{array}
$$

If the test succeeds, *tequal* returns a function that allows us to transform the dynamic value into a static value of the specified type. Of course, as the types are equal, this function is necessarily the identity! Turning to the implementation of *tequal*, the functions *list* and *pair* are the mapping functions of the list and the pair type constructor. Since the equality check may fail, we must lift the mapping functions into the *Maybe* monad (using *return*, *liftM*, and *liftM2*). The cast operation simply calls *tequal* and then applies the conversion function to the dynamic value.

$$
\begin{array}{lcl}
\mathit{cast} & :: & \forall t\,.\,\mathit{Dynamic} \rightarrow \mathit{Type}\ t \rightarrow \mathit{Maybe}\ t\\
\mathit{cast}\ (\mathit{Dyn}\ \mathit{ra}\ a)\ \mathit{rt} & = & \mathit{fmap}\ (\lambda f \rightarrow f\ a)\ (\mathit{tequal}\ \mathit{ra}\ \mathit{rt})
\end{array}
$$

Here is a short interactive session that illustrates its use.

$$
\begin{array}{l}
\mathit{Main}\rangle\ \ \mathbf{let}\ d = \mathit{Dyn}\ \mathit{RInt}\ 60\\
\mathit{Main}\rangle\ \ \mathit{cast}\ d\ \mathit{RInt}\\
\mathit{Just}\ 60\\
\mathit{Main}\rangle\ \ \mathit{cast}\ d\ \mathit{RChar}\\
\mathit{Nothing}
\end{array}
$$

Exercise 6 Define functions that compress and uncompress type representations. *Hint:* define an auxiliary data type

$$\mathbf{data}\ \mathit{Rep} \ = \ \mathit{Rep}\ (\mathit{Type}\ t)$$

and then implement functions $\mathit{compressRep} :: \mathit{Rep} \rightarrow [\mathit{Bit}]$ and $\mathit{uncompressRep} :: [\mathit{Bit}] \rightarrow \mathit{Rep}$ that compress and uncompress elements of type *Rep*. Why do we need the auxiliary data type? □

Exercise 7 Use the results of the previous exercise to implement functions that compress and uncompress dynamic values. To compress a dynamic value, first compress the type representation and then compress the static value. Conversely, to uncompress a dynamic value first uncompress the type representation and then use the type representation to read in a static value of this type. Finally, extend the generic functions *compress* and *uncompress* to take care of dynamic values. □

Exercise 8 Implement functions that pretty print and parse dynamic values and extend the definitions of *pretty* and *parse* accordingly. □

Exercise 9 Extend the type of type representations *Type* and the dynamic type equality check *tequal* to include functional types of the form $a \to b$. $\square$

4 Generic traversals and queries

Let us develop the theme of Section 2 a bit further. Suppose you have to write a function that traverses a complex data structure representing a university's organisational structure, and that increases the age of a given person. The interesting part of this function, namely the increase of age, is probably dominated by the *boilerplate code* that recurses over the data structure. The boilerplate code is not only tiresome to program, it is also highly vulnerable to changes in the underlying data structure. Fortunately, generic programming saves the day as it allows us to write the traversal code once and use it many times. Before we look at an example let us first introduce a data type of persons.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Name} && = && \mathit{String}\\
&\textbf{type}\ \mathit{Age} && = && \mathit{Int}\\
&\textbf{data}\ \mathit{Person} && = && \mathit{Person}\ \mathit{Name}\ \mathit{Age}
\end{aligned}
$$

To be able to apply generic programming techniques, we add *Person* to the type of representable types.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Type}\ t && = && \cdots\\
& && \mid && \mathit{RPerson}\quad \textbf{with}\ t = \mathit{Person}
\end{aligned}
$$

Now, the aforementioned function that increases the age can be programmed as follows (this is only the interesting part without the boilerplate code):

$$
\begin{aligned}
&\mathit{tick} && :: && \mathit{Name} \to \mathit{Traversal}\\
&\mathit{tick}\ s\ (\mathit{RPerson})\ (\mathit{Person}\ n\ a) && && \\
&\quad \mid s == n && = && \mathit{Person}\ n\ (a+1)\\
&\mathit{tick}\ s\ \mathit{rt}\ t && = && t
\end{aligned}
$$

The function *tick s* is a so-called *traversal*, which can be used to modify data of *any* type (the type *Traversal* will be defined shortly). In our case, *tick s* changes values of type *Person* whose name equals s; integers, characters, lists etc are left unchanged.

The following interactive session shows the traversal *tick* in action. The combinator *everywhere*, defined below, implements the generic part of the traversal: it applies its argument ‘everywhere’ in a given value.

$$
\begin{array}{l}
\mathit{Main}\rangle\ \ \textbf{let}\ \mathit{ps} = [\mathit{Person}\ \texttt{"Norma"}\ 50, \mathit{Person}\ \texttt{"Richard"}\ 59] \\
\mathit{Main}\rangle\ \ \mathit{everywhere}\ (\mathit{tick}\ \texttt{"Richard"})\ (\mathit{RList}\ \mathit{RPerson})\ \mathit{ps} \\
[\mathit{Person}\ \texttt{"Norma"}\ 50, \mathit{Person}\ \texttt{"Richard"}\ 60] \\
\mathit{Main}\rangle\ \ \mathit{total}\ \mathit{age}\ (\mathit{RList}\ \mathit{RPerson})\ \mathit{it} \\
110 \\
\mathit{Main}\rangle\ \ \mathit{total}\ \mathit{sizeof}\ \mathit{rString}\ \texttt{"Richard Bird"} \\
60
\end{array}
$$

The second and the third example illustrate *generic queries*: *age* computes the age of a person, *sizeof* yields the size of an object (the number of occupied memory cells), *total* applies an integer query to every component of a value and sums up the results.

Turning to the implementation the type of generic traversals is given by:

$$\textbf{type}\ \mathit{Traversal} \quad = \quad \forall t\,.\, \mathit{Type}\ t \to t \to t.$$

A generic traversal takes a type representation and transforms a value of the specified type. The universal quantifier makes explicit that the function works for *all* representable types. The simplest traversal is *copy*, which does nothing.

$$
\begin{array}{lcl}
\mathit{copy} & :: & \mathit{Traversal} \\
\mathit{copy}\ \mathit{rt} & = & \mathit{id}
\end{array}
$$

Traversals can be composed using the operator ‘$\circ$’, which has *copy* as its identity.

$$
\begin{array}{lcl}
(\circ) & :: & \mathit{Traversal} \to \mathit{Traversal} \to \mathit{Traversal} \\
(f \circ g)\ \mathit{rt} & = & f\ \mathit{rt} \cdot g\ \mathit{rt}
\end{array}
$$

The *everywhere* combinator is implemented in two steps. We first define a function that applies a traversal f to the *immediate* components of a value: $C\ t_1\ \ldots\ t_n$ is mapped to $C\ (f\ \mathit{rt}_1\ t_1)\ \ldots\ (f\ \mathit{rt}_n\ t_n)$ where rt_i is the representation of t_i's type.

$$
\begin{array}{lcl}
\mathit{imap} & :: & \mathit{Traversal} \to \mathit{Traversal} \\
\mathit{imap}\ f\ (\mathit{RInt})\ i & = & i \\
\mathit{imap}\ f\ (\mathit{RChar})\ c & = & c \\
\mathit{imap}\ f\ (\mathit{RList}\ \mathit{ra})\ [\,] & = & [\,] \\
\mathit{imap}\ f\ (\mathit{RList}\ \mathit{ra})\ (a : \mathit{as}) & = & f\ \mathit{ra}\ a : f\ (\mathit{RList}\ \mathit{ra})\ \mathit{as} \\
\mathit{imap}\ f\ (\mathit{RPair}\ \mathit{ra}\ \mathit{rb})\ (a, b) & = & (f\ \mathit{ra}\ a, f\ \mathit{rb}\ b) \\
\mathit{imap}\ f\ (\mathit{RPerson})\ (\mathit{Person}\ n\ a) & = & \mathit{Person}\ (f\ \mathit{rString}\ n)\ (f\ \mathit{RInt}\ a)
\end{array}
$$

The function *imap* can be seen as a ‘traversal transformer’. Note that *imap* has a so-called *rank-2 type*: it takes polymorphic functions to polymorphic functions. The combinator *everywhere* enjoys the same type.

$$
\begin{array}{lcl}
\mathit{everywhere}, \mathit{everywhere}' & :: & \mathit{Traversal} \to \mathit{Traversal} \\
\mathit{everywhere}\ f & = & f \circ \mathit{imap}\ (\mathit{everywhere}\ f) \\
\mathit{everywhere}'\ f & = & \mathit{imap}\ (\mathit{everywhere}'\ f) \circ f
\end{array}
$$

Actually, there are two flavours of the combinator: *everywhere* f applies f *after* the recursive calls (it proceeds bottom-up), whereas *everywhere′* applies f *before* (it proceeds top-down). And yes, *everywhere* and *everywhere′* have the structure of generic folds and unfolds—only the types are different (Chapter **??** treats folds and unfolds in detail).

Generic queries have a similar type except that they yield a value of some fixed type.

$$\textbf{type}\ \mathit{Query}\ x \quad = \quad \forall t\,.\,\mathit{Type}\ t \to t \to x$$

In the rest of this section we confine ourselves to queries of type *Query Int*. Exercise 11 deals with the general case. The definition of the combinator *total* follows the model of *everywhere*. We first define a non-recursive, auxiliary function that sums up the immediate components of a value and then tie the recursive knot.

$$\begin{array}{lcl}
\mathit{isum} & :: & \mathit{Query}\ \mathit{Int} \to \mathit{Query}\ \mathit{Int} \\
\mathit{isum}\ f\ (\mathit{RInt})\ a & = & 0 \\
\mathit{isum}\ f\ (\mathit{RChar})\ a & = & 0 \\
\mathit{isum}\ f\ (\mathit{RList}\ \mathit{ra})\ [\,] & = & 0 \\
\mathit{isum}\ f\ (\mathit{RList}\ \mathit{ra})\ (a : \mathit{as}) & = & f\ \mathit{ra}\ a + f\ (\mathit{RList}\ \mathit{ra})\ \mathit{as} \\
\mathit{isum}\ f\ (\mathit{RPair}\ \mathit{ra}\ \mathit{rb})\ (a, b) & = & f\ \mathit{ra}\ a + f\ \mathit{rb}\ b \\
\mathit{isum}\ f\ (\mathit{RPerson})\ (\mathit{Person}\ s\ i) & = & f\ \mathit{rString}\ s + f\ \mathit{RInt}\ i \\
\mathit{total} & :: & \mathit{Query}\ \mathit{Int} \to \mathit{Query}\ \mathit{Int} \\
\mathit{total}\ f\ \mathit{rt}\ t & = & f\ \mathit{rt}\ t + \mathit{isum}\ (\mathit{total}\ f)\ \mathit{rt}\ t
\end{array}$$

It remains to define the ad-hoc queries *age* and *sizeof*.

$$\begin{array}{lcl}
\mathit{age} & :: & \forall t\,.\,\mathit{Type}\ t \to t \to \mathit{Age} \\
\mathit{age}\ (\mathit{RPerson})\ (\mathit{Person}\ n\ a) & = & a \\
\mathit{age}\ _\ _ & = & 0 \\
\mathit{sizeof} & :: & \mathit{Query}\ \mathit{Int} \\
\mathit{sizeof}\ (\mathit{RInt})\ _ & = & 2 \\
\mathit{sizeof}\ (\mathit{RChar})\ _ & = & 2 \\
\mathit{sizeof}\ (\mathit{RList}\ \mathit{ra})\ [\,] & = & 0 \\
\mathit{sizeof}\ (\mathit{RList}\ \mathit{ra})\ (_ : _) & = & 3 \\
\mathit{sizeof}\ (\mathit{RPair}\ \mathit{ra}\ \mathit{rb})\ _ & = & 3 \\
\mathit{sizeof}\ (\mathit{RPerson})\ _ & = & 3
\end{array}$$

Using *total sizeof* we can compute the memory consumption of a data structure. Actually, the result is a conservative estimate since any sharing of subtrees is ignored. Note that the empty list consumes no memory since it need be represented only once (it can be globally shared).

Exercise 10 Prove the following properties of *imap* (which justify its name).

$$\begin{array}{rcl}
\mathit{imap}\ \mathit{copy} & = & \mathit{copy} \\
\mathit{imap}\ (f \circ g) & = & \mathit{imap}\ f \circ \mathit{imap}\ g
\end{array}$$

Does *everywhere* satisfy similar properties? □

Exercise 11 Generalize *isum* and *total* to functions

$$icrush, everything \quad :: \quad \forall x\,.\,(x \to x \to x) \to x \to Query\ x \to Query\ x$$

such that *icrush* $(+)\ 0 = isum$ and *everything* $(+)\ 0 = total$. □

5 Normalization by evaluation

Let us move on to one of the miracles of theoretical computer science. In Haskell, one cannot show values of functional types. For reasons of computability, there is no systematic way of showing functions and any ad-hoc approach would destroy *referential transparency* (except if *show* were a constant function). For instance, if *show* yielded the text of a function definition, we could distinguish, say, quick sort from merge sort. Substituting one for the other could then possibly change the meaning of a program.

However, what we *can* do is to print the *normal form* of a function. This does not work for Haskell in its full glory, but only for a very tiny subset, the simply typed lambda calculus. Nonetheless, the ability to do that is rather surprising. Let us consider an example first. Suppose you have defined the following Haskell functions (the famous SKI combinators)

$$\begin{aligned} s &= \lambda x\ y\ z \to (x\ z)\ (y\ z) \\ k &= \lambda x\ y \to x \\ i &= \lambda x \to x \end{aligned}$$

and you want to normalize combinator expressions. The function *reify*, defined below, allows you to do that: it takes a type representation (where b represents the base type and ':→' functional types) and yields the normal form of a Haskell value of this type, where the normal form is given as an element of a suitable expression data type.

$$\begin{aligned}
&Main\rangle\ \ reify\ (b :\to b)\ (s\ k\ k) \\
&Fun\ (\lambda a \to a) \\
&Main\rangle\ \ reify\ (b :\to (b :\to b))\ (s\ (k\ k)\ i) \\
&Fun\ (\lambda a \to Fun\ (\lambda b \to a)) \\
&Main\rangle\ \ \mathbf{let}\ e = (s\ ((s\ (k\ s))\ ((s\ (k\ k))\ i)))\ ((s\ ((s\ (k\ s))\ ((s\ (k\ k))\ i)))\ (k\ i)) \\
&Main\rangle\ \ \mathbf{:type}\ e \\
&\forall t\,.\,(t \to t) \to t \to t \\
&Main\rangle\ \ reify\ ((b :\to b) :\to (b :\to b))\ e \\
&Fun\ (\lambda a \to Fun\ (\lambda b \to App\ a\ (App\ a\ b)))
\end{aligned}$$

The last test case is probably the most interesting one as the expression e is quite involved. We first use Haskell's type inferencer to determine its type, then we call *reify* passing it a representation of the inferred type and e itself. And voilà:

the computed result shows that e normalizes to a function that applies its first argument twice to its second.

Now, since we want to represent simply typed lambda terms, we change the type of type representations to

$$
\begin{array}{lcll}
\textbf{infixr} :\rightarrow & & & \\
\textbf{data}\ \mathit{Type}\ t & = & \mathit{RBase} & \textbf{with}\ t = \mathit{Base} \\
 & \mid & \mathit{Type}\ a :\rightarrow \mathit{Type}\ b & \textbf{with}\ t = a \rightarrow b \\
b & :: & \mathit{Type}\ \mathit{Base} & \\
b & = & \mathit{RBase}. &
\end{array}
$$

Here, *Base* is the base type of the simply typed lambda calculus. We won't reveal its definition until later. To represent lambda terms we use *higher-order abstract syntax*. For instance, the lambda term $\lambda f.\lambda x.f\ (f\ x)$ is represented by the Haskell term $\mathit{Fun}\ (\lambda f \rightarrow \mathit{Fun}\ (\lambda x \rightarrow \mathit{App}\ f\ (\mathit{App}\ f\ x)))$, that is, abstractions are represented by Haskell functions.

$$
\begin{array}{lcll}
\textbf{data}\ \mathit{Term}\ t & = & \mathit{App}\ (\mathit{Term}\ (a \rightarrow b))\ (\mathit{Term}\ a) & \textbf{with}\ t = b \\
 & \mid & \mathit{Fun}\ (\mathit{Term}\ a \rightarrow \mathit{Term}\ b) & \textbf{with}\ t = a \rightarrow b
\end{array}
$$

Note that since we use higher-order abstract syntax there is no need to represent variables.

The function *reify* takes a Haskell value of type t to an expression of type $\mathit{Term}\ t$. It is defined by induction over the structure of types, that is, it is driven by the type representation of t. Let us consider functional types first. In this case, *reify* has to turn a value of type $a \rightarrow b$ into an expression of type $\mathit{Term}\ (a \rightarrow b)$. The constructor *Fun* constructs terms of this type, so we are left with converting an $a \rightarrow b$ value to a $\mathit{Term}\ a \rightarrow \mathit{Term}\ b$ value (unfortunately, *Term* does not give rise to a mapping function). Suppose that there is a transformation of type $\mathit{Term}\ a \rightarrow a$ available. Then we can reflect a $\mathit{Term}\ a$ to an a, apply the given function, and finally reify the resulting b to a $\mathit{Term}\ b$. In other words, to implement *reify* we need its converse, as well. Turning to the base case, this means that we require functions of type $\mathit{Base} \rightarrow \mathit{Term}\ \mathit{Base}$ and $\mathit{Term}\ \mathit{Base} \rightarrow \mathit{Base}$. Fortunately, we are still free in the choice of the base type. An intriguing option is to set *Base* to the *fixed point* of *Term*.

$$
\textbf{newtype}\ \mathit{Base} \quad = \quad \mathit{In}\{\mathit{out} :: \mathit{Term}\ \mathit{Base}\}
$$

Then the isomorphisms *out* and *In* constitute the required functions. Given these prerequisites we can finally define *reify* and its inverse *reflect*.

$$
\begin{array}{lcl}
\mathit{reify} & :: & \forall t\,.\,\mathit{Type}\ t \rightarrow (t \rightarrow \mathit{Term}\ t) \\
\mathit{reify}\ (\mathit{RBase})\ v & = & \mathit{out}\ v \\
\mathit{reify}\ (\mathit{ra} :\rightarrow \mathit{rb})\ v & = & \mathit{Fun}\ (\lambda x \rightarrow \mathit{reify}\ \mathit{rb}\ (v\ (\mathit{reflect}\ \mathit{ra}\ x))) \\
\mathit{reflect} & :: & \forall t\,.\,\mathit{Type}\ t \rightarrow (\mathit{Term}\ t \rightarrow t) \\
\mathit{reflect}\ (\mathit{RBase})\ e & = & \mathit{In}\ e \\
\mathit{reflect}\ (\mathit{ra} :\rightarrow \mathit{rb})\ e & = & \lambda x \rightarrow \mathit{reflect}\ \mathit{rb}\ (\mathit{App}\ e\ (\mathit{reify}\ \mathit{ra}\ x))
\end{array}
$$

Exercise 12 Implement a *show* function for $Term\ t$. *Hint:* augment the expression type $Term\ t$ by an additional constructor *Var* of type $String \to Term\ t$. □

6 Functional unparsing

Can we program C's *printf* function in a statically typed language such as Haskell? Yes, we can, provided we use a tailor-made type of format directives (rather than a string). Here is an interactive session that illustrates the puzzle (we renamed *printf* to *format*).

```
Main⟩ :type format (Lit "Richard")
String
Main⟩ format (Lit "Richard")
"Richard"
Main⟩ :type format Int
Int → String
Main⟩ format Int 60
"60"
Main⟩ :type format (String :^: Lit " is " :^: Int)
String → Int → String
Main⟩ format (String :^: Lit " is " :^: Int) "Richard" 60
"Richard is 60"
```

The format directive $Lit\ s$ means emit s literally. The directives *Int* and *String* instruct *format* to take an additional argument of the types *Int* and *String* respectively, which is then shown. The operator ':^:' is used to concatenate two directives.

The type of *format* depends on its first argument, the format directive. This is something we have already seen a number of times: the type of *compress*, for instance, depends on its first argument, the type representation. Of course, the dependence here is slightly more involved. Yet, this smells like a case for phantom types.

The format directive can be seen as a binary tree of type representations: $Lit\ s$, *Int*, *String* form the leaves, ':^:' constructs the inner nodes. The type of *format* is essentially obtained by linearizing the binary tree mapping, for instance, *String* :^: *Lit* `" is "` :^: *Int* to $String \to Int \to String$.

Before tackling the puzzle proper it is useful to reconsider flattening binary trees (see IFPH, Section 7.3.1). To avoid the repeated use of the expensive '++' operation, one typically defines an auxiliary function that makes use of an accumulating parameter.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Btree}\ a & = & \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{Btree}\ a)\ (\mathit{Btree}\ a) \\
\mathit{flatten} & :: & \forall a\,.\,\mathit{Btree}\ a \to [\,a\,] \\
\mathit{flatten}\ t & = & \mathit{flatcat}\ t\ [\,] \\
\mathit{flatcat} & :: & \forall a\,.\,\mathit{Btree}\ a \to [\,a\,] \to [\,a\,] \\
\mathit{flatcat}\ (\mathit{Leaf}\ a)\ \mathit{as} & = & a : \mathit{as} \\
\mathit{flatcat}\ (\mathit{Fork}\ \mathit{tl}\ \mathit{tr})\ \mathit{as} & = & \mathit{flatcat}\ \mathit{tl}\ (\mathit{flatcat}\ \mathit{tr}\ \mathit{as})
\end{array}
$$

The auxiliary function *flatcat* linearizes the given tree and additionally appends the accumulator to the result.

Now, this technique can be mirrored on the type level using a two-argument phantom type.

$$
\begin{array}{lcll}
\textbf{data}\ \mathit{Dir}\ x\ y & = & \mathit{Lit}\ \mathit{String} & \textbf{with}\ y = x \\
 & \mid & \mathit{Int} & \textbf{with}\ y = \mathit{Int} \to x \\
 & \mid & \mathit{String} & \textbf{with}\ y = \mathit{String} \to x \\
 & \mid & \mathit{Dir}\ y_1\ y_2 :\hat{}: \mathit{Dir}\ x\ y_1 & \textbf{with}\ y = y_2
\end{array}
$$

The first argument corresponds to the accumulating parameter and the second to the overall result. The binary tree is implicitly given by the value constructor. Forming a functional type in a **with** clause corresponds to consing an element to a list. The major difference to the definition of *flatcat* is that *Dir* employs a relational style! In fact, with a little bit of imagination you can read the **data** declaration as a relational program (see also Chapter **??**).

Now, using *Dir* we can assign *format* the type $\forall y\,.\,\mathit{Dir}\ \mathit{String}\ y \to y$: linearizing a directive d and plugging in *String* for the final result type, we obtain y as the type of $\mathit{format}\ d$. Unfortunately, we cannot define *format* directly since its type is not general enough to push the recursion through (see Exercise 13). We have to introduce an auxiliary function that takes a *continuation* and an accumulating string argument.

$$
\begin{array}{lcl}
\mathit{format}' & :: & \forall x\ y\,.\,\mathit{Dir}\ x\ y \to (\mathit{String} \to x) \to (\mathit{String} \to y) \\
\mathit{format}'\ (\mathit{Lit}\ s) & = & \lambda \mathit{cont}\ \mathit{out} \to \mathit{cont}\ (\mathit{out} +\!\!+\ s) \\
\mathit{format}'\ (\mathit{Int}) & = & \lambda \mathit{cont}\ \mathit{out} \to \lambda i \to \mathit{cont}\ (\mathit{out} +\!\!+\ \mathit{show}\ i) \\
\mathit{format}'\ (\mathit{String}) & = & \lambda \mathit{cont}\ \mathit{out} \to \lambda s \to \mathit{cont}\ (\mathit{out} +\!\!+\ s) \\
\mathit{format}'\ (d_1 :\hat{}: d_2) & = & \lambda \mathit{cont}\ \mathit{out} \to \mathit{format}'\ d_1\ (\mathit{format}'\ d_2\ \mathit{cont})\ \mathit{out} \\
\mathit{format} & :: & \forall y\,.\,\mathit{Dir}\ \mathit{String}\ y \to y \\
\mathit{format}\ d & = & \mathit{format}'\ d\ \mathit{id}\ \texttt{""}
\end{array}
$$

Note that $\mathit{format}'\ (d_1 :\hat{}: d_2)$ can be simplified to $\mathit{format}'\ d_1 \cdot \mathit{format}'\ d_2$, where '$\cdot$' is ordinary function composition. This is not a coincidence. In fact, the type $(\mathit{String} \to x) \to (\mathit{String} \to y) = \mathit{MapTrans}\ \mathit{String}\ x\ y$ constitutes an arrow (see Chapter **??**).

Exercise 13 Try to implement $\mathit{format} :: \forall y\,.\,\mathit{Dir}\ \mathit{String}\ y \to y$ directly. Where does the attempt fail? □

Exercise 14 The function $format'$ exhibits quadratic run-time behaviour. Remedy this defect. □

Exercise 15 Instead of using a tree-like structure for format directives, we can alternatively employ a list-like structure.

$$\begin{array}{lcll}
\textbf{data}\ \mathit{Dir}\ x & = & \mathit{End} & \textbf{with}\ x = \mathit{String} \\
& \mid & \mathit{Lit}\ \mathit{String}\ (\mathit{Dir}\ y) & \textbf{with}\ x = y \\
& \mid & \mathit{Int}\ (\mathit{Dir}\ y) & \textbf{with}\ x = \mathit{Int} \to y \\
& \mid & \mathit{String}\ (\mathit{Dir}\ y) & \textbf{with}\ x = \mathit{String} \to y
\end{array}$$

Implement $\mathit{format} :: \forall x\,.\,\mathit{Dir}\ x \to x$ using this type. *Hint:* define an auxiliary function of type $\mathit{format}' :: \forall x\,.\,\mathit{Dir}\ x \to \mathit{String} \to x$. □

7 A type equality type

We have seen in the previous sections that **with** clauses add considerably to the expressiveness of Haskell. Rather surprisingly, **with** clauses need not be a primitive concept, they can be simulated using polymorphic types. The resulting programs are more verbose—this is why we have used **with** clauses in the first place—but they can be readily evaluated using a Haskell 98 implementation that additionally supports existential types.

The principle idea is to represent type equations by a *type equality type*: the **data** declaration

$$\textbf{data}\ T\ t \quad = \quad \cdots \mid C\ t_1\ \ldots\ t_n\ \textbf{with}\ t = u \mid \cdots$$

becomes

$$\textbf{data}\ T\ t \quad = \quad \cdots \mid C\ (u :=: t)\ t_1\ \ldots\ t_n \mid \cdots,$$

where ':=:' is a binary type constructor, the type equality type. This type has the intriguing property that it is non-empty if and only if its argument types are equal.[1] Even more intriguing, its definition goes back to Leibniz. According to Leibniz, two terms are equal if one may be substituted for the other. Adapting this principle to types, we define

$$\textbf{newtype}\ a :=: b \quad = \quad \mathit{Proof}\{\, \mathit{apply} :: \forall f\,.\,f\ a \to f\ b\,\}.$$

Note that the universally quantified type variable f ranges over *type constructors* of kind $* \to *$. Thus, an element of $a :=: b$ is a function that converts an element of type $f\ a$ into an element of $f\ b$ for *any* type constructor f. This function can be seen as constituting a *proof* of the type equality $a = b$. The identity function, for instance, serves as the proof of reflexivity.

$$\begin{array}{lcl}
\mathit{refl} & :: & \forall a\,.\,a :=: a \\
\mathit{refl} & = & \mathit{Proof}\ \mathit{id}
\end{array}$$

[1] We ignore the fact, that in Haskell every type contains the bottom element.

Since we have extended the value constructor C by an additional argument, we also have to adapt programs that use C. Every occurrence of the constructor C on the right-hand side of an equation is replaced by $C\ \mathit{refl}$. It is not hard to convince oneself that $C\ \mathit{refl}$ has indeed the right type. Occurrences on the left-hand side are treated as follows: the equation

$$f\ (C\ p_1\ \ldots\ p_n) \quad = \quad e$$

becomes

$$f\ (C\ p\ p_1\ \ldots\ p_n) \quad = \quad \mathit{apply}\ p\ e.$$

Assume that f has type $\forall t\,.\,T\ t \to F\ t$ where $F\ t$ is some type expression possibly involving t. The **with** clause associated with C dictates that e has type $F\ u$. The right-hand side of the transformed program, however, must have the type $F\ t$. The proof p of type $u :=: t$ allows us to turn e into a value of the desired type. Note that the universally quantified type variable f of the type equality type is instantiated to F.

In some cases it is necessary to guide the Haskell type inferencer so that it indeed instantiates f to F. The problem is that Haskell employs a *kinded first-order unification*. For instance, the types $\mathit{Int} \to [\mathit{Bit}]$ and $f\ \mathit{Int}$ are not unifiable, since the type checker reduces the type equation $((\to)\ \mathit{Int})\ [\mathit{Bit}] = f\ \mathit{Int}$ to $(\to)\ \mathit{Int} = f$ and $[\mathit{Bit}] = \mathit{Int}$. The standard trick to circumvent this problem is to introduce a new type F' that is isomorphic to F.

$$\textbf{newtype}\ F'\ a \quad = \quad \mathit{In}\{\,\mathit{out} :: F\ a\}$$

The equation then becomes

$$f\ (C\ p\ p_1\ \ldots\ p_n) \quad = \quad (\mathit{out} \cdot \mathit{apply}\ p \cdot \mathit{In})\ e.$$

Turning back to the type equality type it is interesting to note that it has all the properties of an *congruence relation*. We have already seen that it is reflexive. It is furthermore symmetric, transitive, and congruent. Here are programs that implement the respective proofs.

$$
\begin{array}{lcl}
\mathbf{newtype}\ \mathit{Flip}\ f\ a\ b & = & \mathit{Flip}\{\mathit{unFlip} :: f\ b\ a\} \\
\mathit{symm} & :: & \forall a\ b\,.\,(a :=: b) \to (b :=: a) \\
\mathit{symm}\ p & = & \mathit{unFlip}\ (\mathit{apply}\ p\ (\mathit{Flip}\ \mathit{refl})) \\
\mathit{trans} & :: & \forall a\ b\ c\,.\,(a :=: b) \to (b :=: c) \to (a :=: c) \\
\mathit{trans}\ p\ q & = & \mathit{Proof}\ (\mathit{apply}\ q \cdot \mathit{apply}\ p) \\
\mathbf{newtype}\ \mathit{List}\ f\ a & = & \mathit{List}\{\mathit{unList} :: f\ [a]\} \\
\mathit{list} & :: & \forall a\ b\,.\,(a :=: b) \to ([a] :=: [b]) \\
\mathit{list}\ p & = & \mathit{Proof}\ (\mathit{unList} \cdot \mathit{apply}\ p \cdot \mathit{List}) \\
\mathbf{newtype}\ \mathit{Pair}_1\ f\ b\ a & = & \mathit{Pair}_1\{\mathit{unPair}_1 :: f\ (a,b)\} \\
\mathbf{newtype}\ \mathit{Pair}_2\ f\ a\ b & = & \mathit{Pair}_2\{\mathit{unPair}_2 :: f\ (a,b)\} \\
\mathit{pair} & :: & \forall a\ b\ c\ d\,.\,(a :=: c) \to (b :=: d) \to ((a,b) :=: (c,d)) \\
\mathit{pair}\ p_1\ p_2 & = & \mathit{Proof}\ (\mathit{unPair}_2 \cdot \mathit{apply}\ p_2 \cdot \mathit{Pair}_2 \\
 & & \qquad \cdot\ \mathit{unPair}_1 \cdot \mathit{apply}\ p_1 \cdot \mathit{Pair}_1)
\end{array}
$$

Again, we have to introduce auxiliary data types to direct Haskell's type inferencer. As an example, the proof of symmetry works as follows. We first specialize the given proof of $(a :=: b) = (\forall f\,.\,f\ a \to f\ b)$ setting f to $(:=: a)$. We obtain a function of type $(a :=: a) \to (b :=: a)$, which is then passed *refl* to yield the desired proof of $b :=: a$.

Before we conclude, let us briefly revise the type equality check *tequal* of Section 3. Recall that *tequal* returns a conversion function of type $t \to u$ that allows us to transform dynamic values into static values. A far more flexible approach is to replace $t \to u$ by $t :=: u$, so that we can transform a t to a u in *any* context.

$$\mathit{tequal} \ :: \ \forall t\ u\,.\,\mathit{Type}\ t \to \mathit{Type}\ u \to \mathit{Maybe}\ (t :=: u)$$

The changes to the definition of *tequal* are simple: we have to replace *id* by *refl*, and the mapping functions *pair* and *list* by the congruence proofs of the same name.

Exercise 16 Extend the above transformation to cover multiple type arguments and multiple type equations. □

Exercise 17 Define conversion functions $\mathit{from} :: \forall a\ b\,.\,(a :=: b) \to (a \to b)$ and $\mathit{to} :: \forall a\ b\,.\,(a :=: b) \to (b \to a)$. Try to implement them from scratch. □

Exercise 18 We have defined congruence proofs for the list and the pair type constructor. Generalize the construction to an arbitrary n-ary data type not necessarily being a functor. □

8 Chapter notes

This chapter is based on a paper by Cheney and Hinze [2], which shows how to combine generics and dynamics in a type-safe manner. The term *phantom type* was coined by Leijen and Meijer [8] to denote parameterized types that do not use their type argument.

There is an abundance of work on generic programming, see, for instance, [6, 5]. For a gentle introduction to the topic the interested reader is referred to [1].

Section 4 draws from a paper by Lämmel and Peyton Jones [7]. Sections 5 and 6 adopt two pearls by Danvy, Rhiger and Rose [4] and by Danvy [3], respectively. An alternative approach to unparsing is described by Hinze [**?**].

Acknowledgement

I would like to thank Andres Löh for his helpful and immediate feedback on a draft version of this chapter.

References

[1] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic Programming — An Introduction —. In S. Doaitse Swierstra, Pedro R. Henriques, and Jose N. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming, Braga, Portugal*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115. Springer-Verlag, Berlin, 1999.

[2] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In Manuel M.T. Chakravarty, editor, *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop*, pages 90–104. ACM Press, October 2002.

[3] Olivier Danvy. Functional unparsing. *J. Functional Programming*, 8(6):621–625, November 1998.

[4] Olivier Danvy, Morten Rhiger, and Kristoffer H. Rose. Normalization by evaluation with typed abstract synatx. *J. Functional Programming*, 11(6):673–680, November 2001.

[5] Ralf Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00), Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.

[6] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 470–482. ACM Press, January 1997.

[7] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. Available from `http://research.microsoft.com/~simonpj/papers/hmap/`, 2002.

[8] Daan Leijen and Erik Meijer. Domain-specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 109–122, Berkeley, CA, October 1999. USENIX Association.