Unfolding Abstract Datatypes

Jeremy Gibbons

Computing Laboratory, University of Oxford
http://www.comlab.ox.ac.uk/jeremy.gibbons/

Abstract. We argue that *abstract datatypes* — with public interfaces hiding private implementations — represent a form of *codata* rather than ordinary data, and hence that proof methods for *corecursive* programs are the appropriate techniques to use for reasoning with them. In particular, we show that the universal properties of unfold operators are perfectly suited for the task. We illustrate with the solution to a problem in the recent literature.

1 Introduction

Dijkstra [10] argued that the single most important contribution computing science has made to the world is the emphasis on designing abstractions in order to manage complexity. *Abstract datatypes* [30] — with public interfaces hiding private implementations — have a pivotal role to play in that contribution. Nevertheless, the use of abstract datatypes is not as common among functional programmers (particularly those using languages like Haskell, which does not have first-class modules) as one might expect from history's lesson. One reason for this phenomenon might be the seductive attractions of pattern matching over algebraic datatypes [55], which seem to rely on making visible the representation of data and hence breaking the encapsulation; we return to this point in Section 5. But another reason for the underuse of abstract datatypes, we feel, is that they are not subject to the familiar proof methods of equational reasoning and induction to which functional programming so readily lends itself [2].

The essential reason why standard proof techniques are inapplicable to abstract datatypes is that they are a form of *codata* rather than a form of *data*, with an emphasis on observation rather than construction, process rather than value, and the indefinite rather than the finite. In this paper, we argue that the appropriate proof methods for reasoning about abstract datatypes are those associated with *corecursive programs* [15]. In particular, building on established work on final coalgebra semantics for object-oriented programs, we show that the universal properties of unfold operators [16] fit the bill very nicely.

Our use here of unfold operators, and hence of possibly-infinite data structures, pushes us towards lazy rather than eager functional programming. That works out nicely, because it is the lazy functional programmer who tends to place greater emphasis on equational reasoning. On the other hand, aficionados of ML at least have a powerful module facility at their disposal, and so might be expected to make greater use of abstract datatypes than do adherents of Haskell.

We illustrate our case with the solution to a problem of reasoning with abstract datatypes from the recent literature, concerning the elimination of redundant conversions to and from lists in a stream-based reimplementation of the Haskell standard list library [9].

The remainder of this paper is structured as follows. Section 2 explains the modelling of abstract datatypes using existential type quantification, and Section 3 discusses the corecursive proof methods appropriate for reasoning about such constructions. Section 4 presents the case study on Coutts *et al.*'s stream fusion. Section 5 concludes and discusses related work.

2 Abstract types have existential type

Abstract datatypes are "a kind of data abstraction where a type's internal form is hidden behind a set of access functions; values of the type are created and inspected only by calls to the access functions" [22]. Hiding of the internal form is achieved by existential quantification over the representation type: an abstract data structure consists of operations operating on an internal state, whose type is hidden from everything except those operations. Mitchell and Plotkin [32] expressed this view in the slogan "abstract types have existential type".

2.1 An example: complex numbers

For example, consider (a simplification of) the Haskell datatype *Complex*:

$$\textbf{data}\ \mathit{Complex} = \mathit{MkComplex}\ \mathit{Double}\ \mathit{Double}$$

This introduces a constructor $\mathit{MkComplex} :: \mathit{Double} \to \mathit{Double} \to \mathit{Complex}$. The outcome is not an abstract datatype, because the representation as a pair of *Double*s (as Cartesian coordinates, as it happens) is visible. Instead, it is a *concrete datatype*. This provides other advantages — such as pattern matching — but loses the benefit of information hiding. For example, to determine whether a complex number is real, we can use the following function:

$$\begin{aligned}&\mathit{isReal} :: \mathit{Complex} \to \mathit{Bool}\\ &\mathit{isReal}\ (\mathit{MkComplex}\ x\ y) = (y == 0.0)\end{aligned}$$

But if we decide to change the representation to polar coordinates, all such definitions will need modification.

An abstract datatype of complex numbers should hide the representation, as follows [43].

$$\begin{aligned}\textbf{data}\ \mathit{Complex} = \exists s.\ C\ &(s \to (\mathit{Double}, \mathit{Double}) \to s) && \text{-- create}\\ &(s \to \mathit{Complex} \to s) && \text{-- add}\\ &(s \to \mathit{Double}) && \text{-- real}\\ &(s \to \mathit{Double}) && \text{-- imaginary}\\ &s && \text{-- self}\end{aligned}$$

As above, this introduces a new constructor C; this takes four functions and an internal representation or 'self', and yields a *Complex*. Here, the self is of type s, for some s; the four functions each take an argument of type s. Note that the type variable s does not appear on the left-hand side of the datatype declaration, so it has to be quantified somehow. A universal quantification would be inappropriate, because the representation is of *some* type, not *any* type; existential quantification is what is required. (In common extensions to Standard Haskell supporting existential quantification, somewhat perversely, it is written with the keyword `forall` [39] — the justification being that a datatype declaration such as

$$\textbf{data}\ D = \exists s.\ MkD\ (s, s \to Integer)$$

introduces a constructor $MkD :: (\exists s.\ (s, s \to Integer)) \to D$, and this type is isomorphic to $\forall s.\ ((s, s \to Integer) \to D)$ because D is independent of s — but in this paper we will pretty-print that keyword as '$\exists$'.)

Packaged up with the internal representation of a complex number are four functions: for creating a new complex number, adding on a second complex number (obtaining an updated representation), and extracting the real and imaginary components. These can be given more user-friendly names:

$$\begin{aligned}
&new :: Complex \to Double \to Double \to Complex\\
&new\ (C\ n\ a\ r\ i\ s)\ x\ y = C\ n\ a\ r\ i\ (n\ s\ (x, y))\\
&add :: Complex \to Complex \to Complex\\
&add\ (C\ n\ a\ r\ i\ s)\ c\ \ \ = C\ n\ a\ r\ i\ (a\ s\ c)\\
&rea, ima :: Complex \to Double\\
&rea\ \ (C\ n\ a\ r\ i\ s)\ \ \ \ = r\ s\\
&ima\ (C\ n\ a\ r\ i\ s)\ \ \ \ = i\ s
\end{aligned}$$

Crucially, nothing other than these four functions can access the internal representation; that is guaranteed by the quantification over the type variable s. In particular, there is no way to extract the representation itself. So of course, the set of operations made available has to be considered carefully; in contrast to concrete datatypes, which support pattern matching and hence easy extension with new functions, adding a new operation to an abstract datatype inexpressible in terms of existing functions requires a change to the datatype definition [8].

The abstract datatype specifies a signature, but not an implementation. Here is one implementation, in the expected Cartesian coordinates:

$$\begin{aligned}
&zeroC :: Complex\\
&zeroC = C\ (\lambda(x, y) \to \lambda z \to z)\\
&\qquad\quad (\lambda(x, y) \to \lambda c \to (x + rea\ c, y + ima\ c))\\
&\qquad\quad (\lambda(x, y) \to x)\\
&\qquad\quad (\lambda(x, y) \to y)\\
&\qquad\quad (0.0, 0.0)
\end{aligned}$$

Notice the type $s \to Complex \to s$ for the addition function, allowing complex numbers of different representations to be added. Consequently, the

implementation of *add* has privileged access to the representation of the first argument (through the fields x and y), but only public access to the second argument (through the operations *rea* and *ima*). This inefficiency is a well-known problem with binary methods in object orientation [4]. Mitchell and Plotkin's approach [32] differs, providing privileged access to the representations of both arguments of *add* but therefore requiring both arguments to have the same representation; their alternative is more efficient, but less flexible. The difference is essentially a matter of whether the scope of the existential quantification is narrowed down to specific objects, or widened out to the whole program.

Note also the rather odd type $Complex \to Double \to Double \to Complex$ for *new*, requiring an existing complex number before a new one can be created. Of course, the implementation of an abstract data structure has to come from somewhere; the function *new* creates a new structure using the operations and data representation of an existing structure, simply assigning a new state. This is more analogous to cloning in prototype-based languages [54] than it is to construction *de novo* in more traditional object-oriented programming.

2.2 An alternative implementation

Here is an alternative implementation of complex numbers, using polar coordinates.

$$
\begin{aligned}
&\mathbf{data}\ Polar = P\{\, mag :: Double, phase :: Double \,\}\\
&zeroP :: Complex\\
&zeroP = C\ (\lambda p \to \lambda z \to c2p\ z)\\
&\qquad\qquad (\lambda p \to \lambda c \to \mathbf{let}\ (x, y) = p2c\ p\ \mathbf{in}\ c2p\ (x + rea\ c, y + ima\ c))\\
&\qquad\qquad (\lambda p \to fst\ (p2c\ p))\\
&\qquad\qquad (\lambda p \to snd\ (p2c\ p))\\
&\qquad\qquad (P\{\, mag = 0.0, phase = 0.0 \,\})\\
&p2c\ p = (mag\ p \times cos\ (phase\ p), mag\ p \times sin\ (phase\ p))\\
&c2p\ (x, y) = P\{\, mag = sqrt\ (x{\times}x + y{\times}y), phase = atan2\ y\ x \,\}
\end{aligned}
$$

(A wiser definition of *c2p* would scale the two coordinates before multiplying, to avoid overflows; but the naive version above is clearer.) Note that although *zeroC* and *zeroP* have different representations, the existential quantification allows them to be of the same type.

Now, the reality check for complex numbers becomes:

$$
\begin{aligned}
&isReal :: Complex \to Bool\\
&isReal\ z = (ima\ z == 0)
\end{aligned}
$$

We can no longer use pattern matching on the representation; but this definition works just as well for a polar — or indeed, any other — representation as for Cartesian.

2.3 An explicit signature for complex numbers

The type declaration *Complex* is rather complicated, on account of the number of operations provided. Moreover, those operations all have a common domain, the hidden representation type; so they can be coalesced into one function returning a tuple. We will adopt a convention of separating out the description of the signature (that is, the number and types of the operations) from the existential quantification, writing the following mutually recursive definitions instead.

$$\begin{aligned}&\textbf{data}\ \mathit{ComplexF}\ s = \mathit{CF}\{\ \_\mathit{new} :: (\mathit{Double}, \mathit{Double}) \to s,\\ &\qquad\qquad\qquad\quad \_\mathit{add}\ :: \mathit{Complex} \to s,\\ &\qquad\qquad\qquad\quad \_\mathit{rea}\ \ :: \mathit{Double},\\ &\qquad\qquad\qquad\quad \_\mathit{ima} :: \mathit{Double}\,\}\\ &\textbf{data}\ \mathit{Complex} = \exists s.\ C\ (s \to \mathit{ComplexF}\ s)\ s\end{aligned}$$

Here is the Cartesian implementation of zero:

$$\begin{aligned}&\mathit{zeroC} :: \mathit{Complex}\\ &\mathit{zeroC} = C\ \mathit{fc}\ (0.0, 0.0)\ \textbf{where}\\ &\quad \mathit{fc} :: (\mathit{Double}, \mathit{Double}) \to \mathit{ComplexF}\ (\mathit{Double}, \mathit{Double})\\ &\quad \mathit{fc} = (\lambda(x, y) \to \mathit{CF}\{\ \_\mathit{new} = \lambda z \to z,\\ &\qquad\qquad\qquad\qquad \_\mathit{add} = \lambda c\ \to (x + \mathit{rea}\ c, y + \mathit{ima}\ c),\\ &\qquad\qquad\qquad\qquad \_\mathit{rea} = x,\\ &\qquad\qquad\qquad\qquad \_\mathit{ima} = y\,\})\end{aligned}$$

Note that the function *fc* is analogous to a class in object-oriented terms: it determines the data representation, and provides implementations of the methods on that representation. We can provide user-friendly wrappers $\mathit{rea}, \mathit{ima}, \mathit{add}, \mathit{new}$ as before. Similarly for the polar implementation *zeroP*:

$$\begin{aligned}&\mathit{zeroP} :: \mathit{Complex}\\ &\mathit{zeroP} = C\ \mathit{fp}\ (P\{\,\mathit{mag} = 0.0, \mathit{phase} = 0.0\,\})\ \textbf{where}\\ &\quad \mathit{fp} :: \mathit{Polar} \to \mathit{ComplexF}\ \mathit{Polar}\\ &\quad \mathit{fp} = (\lambda p \to \mathit{CF}\{\ \_\mathit{new} = \lambda z \to \mathit{c2p}\ z,\\ &\qquad\qquad\qquad\quad \_\mathit{add} = \lambda c\ \to \textbf{let}\ (x, y) = \mathit{p2c}\ p\ \textbf{in}\\ &\qquad\qquad\qquad\qquad\qquad\quad \mathit{c2p}\ (x + \mathit{rea}\ c, y + \mathit{ima}\ c),\\ &\qquad\qquad\qquad\quad \_\mathit{rea} = \mathit{fst}\ (\mathit{p2c}\ p),\\ &\qquad\qquad\qquad\quad \_\mathit{ima} = \mathit{snd}\ (\mathit{p2c}\ p)\,\})\end{aligned}$$

2.4 Abstract datatype genericity

Of course, there is nothing special about the particular datatype of complex numbers; the approach generalises very nicely. This leads to *datatype-generic abstract datatypes*, parametrised by the signature [14]. The signature should be a *strictly positive functor*: it should be functorial in the state type (in order to allow the definition of the unfold for the final coalgebra semantics), and no

occurrences of the type parameter may appear to the left of an arrow (to maintain the encapsulation of the hidden state). The abstract datatype itself packages up some hidden state with the operations, of type specified by the signature.

$$\textbf{data}\ \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f = \exists s.\ D\ (s \to f\ s)\ s$$

(The Haskell type class context '*Functor f*' entails an operation *fmap* of type $(a \to b) \to (f\ a \to f\ b)$.) Instantiating the signature parameter to *ComplexF* yields complex numbers:

$$\begin{aligned}&\textbf{type}\ \mathit{Complex} = \mathit{ADT}\ \mathit{ComplexF}\\ &\mathit{zeroCG}, \mathit{zeroPG} :: \mathit{Complex}\\ &\mathit{zeroCG} = D\ \mathit{fc}\ (0.0, 0.0)\\ &\mathit{zeroPG} = D\ \mathit{fp}\ (P\{\mathit{mag} = 0.0, \mathit{phase} = 0.0\})\end{aligned}$$

(Note that *ComplexF* and *Complex* are mutually recursive, so this redefinition of *Complex* entails also a redefinition of *ComplexF*.)

3 Data and codata

Abstract datatypes are inhabited by *codata*, as opposed to the ordinary data inhabiting the more familiar algebraic datatypes. In general, codata is manipulated through destructors instead of constructors. Kieburtz [27] identifies some fundamental respects in which codata differs from data:

- Codata structures have hidden representations, accessible only via operations specifically provided for this purpose; whereas the representations of data structures are visible, for example through pattern matching.
- Consequently, standard datatype-generic operations such as pretty-printing and comparison, automatically defined or easily derived from the structure of an arbitrary datatype, are generally not applicable to codatatypes.
- Codata is typically infinite, since it may provide operations that yield other instances of the same type, as with the *new* and *add* operations in the complex number example in Section 2 (but records with field extractors are an exception to this rule). Data is usually finite (although with lazy evaluation, infinite recursive algebraic data structures can be constructed; we see an example shortly).

As it happens, codatatypes are generally *greatest fixpoints* of recursive type equations, whereas datatypes are *least fixpoints*. In some settings (such as that of continuous functions between complete partial orders, as embodied in Haskell for example), least and greatest type fixpoints coincide; but in many settings (such as total functions between sets, as used in Cockett's Charity [7] and Turner's Total Functional Programming [52,53]) the two are distinguished.

3.1 Greatest fixpoint types as codata

Given a suitable operation F on types (technically, a covariant functor; but for simplicity, think of some combination of sums and products), the least fixpoint $\mu(F)$ of F is the smallest type X such that $F\,(X) \approx X$. This corresponds to the algebraic datatype declaration

$$\textbf{data}\ \mathit{Mu}\,f = \mathit{In}\{\,\mathit{out} :: f\,(\mathit{Mu}\,f)\,\}$$

when read in terms of total functions between sets (rather than continuous functions between complete partial orders), capturing just the total and finite data structures of a given shape. The constructor $\mathit{In} :: f\,(\mathit{Mu}\,f) \to \mathit{Mu}\,f$ and destructor $\mathit{out} :: \mathit{Mu}\,f \to f\,(\mathit{Mu}\,f)$ are the witnesses to the isomorphism.

There is a well-known technique called *Church encoding* [3,19] for representing such least fixpoint or *initial* recursive datatypes in the polymorphic lambda calculus, without having to introduce new language constructs like **data** and pattern matching. The encoding is as a higher-order functional type:

$$\mu(F) = \forall X.\ (F\,(X) \to X) \to X$$

(Technically, *strong initiality*, conferring also a corresponding proof principle, requires additional assumptions, such as parametricity or "theorems for free" in the underlying category [46,57].)

For example, integer lists have shape determined by $L\,(X) = 1 + \mathit{Integer} \times X$, where 1 denotes the unit type with a single element, *Integer* the type of integers, $+$ disjoint union, and $\times$ Cartesian product. Integer lists therefore have the Church encoding $\mu(L) = \forall X.\ (L\,(X) \to X) \to X$. Note that, using standard type isomorphisms, we have $L\,(X) \to X \approx X \times (\mathit{Integer} \times X \to X)$, so an equivalent definition is $\mu(L) = \forall X.\ (X \times (\mathit{Integer} \times X \to X)) \to X$. Moreover, similar type isomorphisms yield a type $\forall a.\ [a] \to \forall b.\ (b, (a, b) \to b) \to b$ for the function *foldr* from the Haskell standard library [40]. In other words, when specialised to $a = \mathit{Integer}$, *foldr* computes the Church encoding of a list of integers.

What is rather less well known is that this encoding dualises, allowing the representation also of greatest fixpoint or *final* recursive types [62,58]:

$$\nu(F) = \exists X.\ (X \to F\,(X)) \times X$$

(Again, parametricity is required in order to deduce *strong finality*, conferring the corresponding proof principle). For example, the type of finite and infinite integer lists, the greatest fixpoint $\nu(L)$ of the functor L, is encoded as $\exists X.\ (X \to 1 + \mathit{Integer} \times X) \times X$. Moreover, standard type isomorphisms yield a type $\forall a.\ (\exists b.\ (b \to \mathit{Maybe}\,(a, b), b)) \to [a]$ for the function *unfoldr* from the Haskell standard library [40]. In other words, when specialised to $a = \mathit{Integer}$, *unfoldr* computes a co-list of integers from its co-Church encoding.

In summary, whereas least fixpoint types correspond to universal type quantifications, greatest fixpoint types correspond to existential quantifications.

3.2 Proof methods for codata

So, *zeroCG* and *zeroPG* are both elements of a greatest fixpoint type. We might expect them to be 'equal' in some sense, since they both represent 'the same' complex number. But in what sense could they be equal? They have different representations, so straightforward structural comparisons are inappropriate.

The generally accepted approach to take to equality on codata, such as between instances of abstract datatypes, is *observational equivalence*, or "equality as far as we can see" [24]. Two instances of an abstract datatype are clearly different if there is an experiment — that is, a sequence of operations provided by the signature — yielding distinguishable *concrete* outputs (which might without loss of generality be as primitive as bits, but we take here to include types like *Integer* and *Double*); and conversely, if no such experiment exists, we consider the two instances to be equal.

That informal characterisation of observational equivalence can be formalised in two ways, which turn out to be equivalent: via *bisimulation and coinduction* [36,21,25] or via *universal properties of final coalgebras* [59,26,45,23]. In this context, bisimulation amounts to the same thing as logical relations and relational parametricity [31,44]. Bisimulation and universal properties are compared in a recent survey paper [15]. That survey applies the techniques to proving equality of concrete datatypes, specifically streams, for which structural comparisons are also available. It therefore takes the structural comparison as the definition of equality, and proves that the other notions coincide with it. Since the present paper concentrates on abstract datatypes, structural comparison is unavailable; but these two notions of observational equivalence still agree with each other.

3.3 Final coalgebras

Here, we take the final coalgebra approach. The single experimental steps available on an abstract data structure of type $ADT\ f$ are captured, with their required inputs and specified outputs, by the signature functor f. The tree of all possible experiments is obtained by repeatedly applying these operations.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ f = T\{\mathit{unT} :: f\ (\mathit{Tree}\ f)\}\\
&\mathit{tree} :: \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f \to \mathit{Tree}\ f\\
&\mathit{tree}\ (D\ h\ s) = \mathit{unfold}\ h\ s\ \textbf{where}\\
&\quad \mathit{unfold} :: \mathit{Functor}\ f \Rightarrow (a \to f\ a) \to a \to \mathit{Tree}\ f\\
&\quad \mathit{unfold}\ f\ x = T\ (\mathit{fmap}\ (\mathit{unfold}\ f)\ (f\ x))
\end{aligned}
$$

As we noted above, because some experimental steps may yield new abstract data structures, observation trees will typically be infinite; accordingly, $\mathit{Tree}\ f$ is the greatest fixpoint of f. Nevertheless, we consider *Tree* to be a type of data rather than of codata: it is amenable to pattern matching and to structural datatype-generic operations such as pretty-printing and comparison. Although in Haskell $\mathit{Mu}\ f$ and $\mathit{Tree}\ f$ coincide semantically, we use different datatypes to reinforce the distinction between least and greatest fixpoints.

Now, the claim that the abstract data structures *zeroCG* and *zeroPG* are *observationally equivalent* reduces to a more amenable statement that the concrete data structures *tree zeroCG* and *tree zeroPG* are *structurally equal*: an experiment distinguishing *zeroCG* and *zeroPG* corresponds to a difference between the two trees, and the absence of such an experiment implies the equality of those trees. The claim is still not effectively decidable, because the trees are both infinitely deep and infinitely wide; but at least it is now open to proof via familiar equational reasoning at the meta-level.

3.4 Proving equivalence

Observational equivalence of the complex numbers *zeroCG* and *zeroPG* follows from structural equality of their observation trees *tree zeroCG* and *tree zeroPG*, which can be demonstrated using the universal property of *unfold*:

$$h = \mathit{unfold}\ f \iff \mathit{unT} \cdot h = \mathit{fmap}\ h \cdot f$$

We have

$$\mathit{tree}\ \mathit{zeroCG} = \mathit{tree}\ \mathit{zeroPG} \iff \mathit{unfold}\ \mathit{fc}\ (0.0, 0.0) = \mathit{unfold}\ \mathit{fp}\ (P\ 0.0\ 0.0)$$

But it isn't immediately obvious how to apply the universal property here: this is not an equation between two functions of the form *unfold h*, but rather between two trees of the form *unfold h s*. How can we move forward?

Fortunately, this is a somewhat special case, because there is a simulation relationship between the two instances. Specifically, we can abstract from the initial state $(0.0, 0.0)$ of the Cartesian implementation, since $(0.0, 0.0) = \mathit{p2c}\ (P\ 0.0\ 0.0)$, obtaining the proof obligation $\mathit{unfold}\ \mathit{fc} \cdot \mathit{p2c} = \mathit{unfold}\ \mathit{fp}$. This equation can be proved using the fusion law of *unfold*, a simple corollary of the universal property:

$$\mathit{unfold}\ f \cdot g = \mathit{unfold}\ f' \impliedby f \cdot g = \mathit{fmap}\ g \cdot f'$$

All that remains is to establish the premise, $\mathit{fc} \cdot \mathit{p2c} = \mathit{fmap}\ \mathit{p2c} \cdot \mathit{fp}$ — that is, that *p2c* is the abstraction function relating *fc* and *fp*. The only property of complex numbers required in the proof is that $\mathit{p2c} \cdot \mathit{c2p} = \mathit{id}$. The calculation can be found in an appendix (Section 6).

We chose here to abstract from the initial state of the Cartesian implementation of the abstract datatype, effectively expressing that implementation in terms of the polar representation. This particular proof of equivalence is doubly special, because the simulation also works the other way around: we could have abstracted the initial state $P\ 0.0\ 0.0$ of the polar implementation instead. (The only complication in doing so is that $\mathit{c2p} \cdot \mathit{p2c}$ is not quite the identity function; however, it is the identity on the reachable states — those with non-negative magnitude, phase between 0 and 2π, and zero phase if zero magnitude.)

In general, given two implementations of an abstract datatype, neither will simulate the other; instead, each introduces extensions inexpressible by the other. In that case, each can be shown observationally equivalent to a third implementation that can simulate both. We will see an example in Section 4.8.

4 Stream fusion

Coutts *et al.* [9] present an elegant technique for obtaining better fusion of list functions, by reimplementing the Standard Haskell list library [40] to use internally an abstract datatype (of 'streams') rather than the familiar algebraic datatype of lists; we summarise work in Sections 4.1–4.4 and 4.7 below. However, they don't prove that their reimplementation is sound; we present such a proof in the remainder of this section.

4.1 An abstract datatype of streams

A simplistic version of Coutts *et al.*'s approach uses a curried version of Haskell's *Maybe* datatype on pairs as the signature of the stream abstract datatype:

$$\begin{aligned}&\textbf{data}\ \mathit{Maybe}_2\ a\ b = \mathit{Nothing}_2 \mid \mathit{Just}_2\ a\ b\\&\textbf{type}\ \mathit{Stream}\ a = \mathit{ADT}\ (\mathit{Maybe}_2\ a)\end{aligned}$$

Thus, a stream has two components, qualified by some existentially bound state type s: an internal state of type s, and a body that when applied to such a state yields either a head and a new state, or nothing. (In other words, the greatest-fixpoint or co-Church encoding is being used.) For example, here is one implementation of the string `"abc"`:

$$\begin{aligned}&\mathit{abc} = D\ h\ 0\ \textbf{where}\\&\quad h\ i = \textbf{case}\ i\ \textbf{of}\\&\qquad 0 \to \mathit{Just}_2\ \texttt{'a'}\ 1\\&\qquad 1 \to \mathit{Just}_2\ \texttt{'b'}\ 2\\&\qquad 2 \to \mathit{Just}_2\ \texttt{'c'}\ 3\\&\qquad 3 \to \mathit{Nothing}_2\end{aligned}$$

The approach can be seen as an implementation of the ITERATOR design pattern from object-oriented programming [13]. The full story of the approach permits a third outcome of the stream body, to deal with nested recursions; we return to this point in Section 4.7 below.

4.2 Stream operations

The list library is redefined in terms of streams. For example, the map function on lists is reimplemented as follows:

$$\begin{aligned}&\mathit{mapS}\ f\ (D\ h\ s) = D\ h'\ s\ \textbf{where}\\&\quad h'\ s = \textbf{case}\ h\ s\ \textbf{of}\ \mathit{Nothing}_2 \to \mathit{Nothing}_2\\&\qquad\qquad\qquad\qquad \mathit{Just}_2\ x\ s' \to \mathit{Just}_2\ (f\ x)\ s'\end{aligned}$$

This function is a rather special case, because the internal representation of the stream $\mathit{mapS}\ f\ \mathit{xs}$ is the same as that of xs. In general, internal representations change; for example, the zip function is:

$$
\begin{aligned}
&\mathit{zipS} :: \mathit{Stream}\ a \to \mathit{Stream}\ b \to \mathit{Stream}\ (a, b)\\
&\mathit{zipS}\ (D\ h\ s)\ (D\ j\ t) = D\ k\ (s, t)\ \mathbf{where}\\
&\quad k\ (s, t) = \mathbf{case}\ (h\ s, j\ t)\ \mathbf{of}\\
&\qquad (\mathit{Just}_2\ x\ s', \mathit{Just}_2\ y\ t') \to \mathit{Just}_2\ (x, y)\ (s', t')\\
&\qquad \_ \qquad\qquad\qquad\quad \to \mathit{Nothing}_2
\end{aligned}
$$

The internal representation (s, t) of the result is of a different type to the representations s and t of the two arguments.

4.3 A list interface

The standard library is reimplemented using streams, but the interface presented to the programmer still uses the familiar algebraic datatype of lists; therefore, conversion functions $\mathit{stream} :: [a] \to \mathit{Stream}\ a$ and $\mathit{unstream} :: \mathit{Stream}\ a \to [a]$ are needed.

$$
\begin{aligned}
&\mathit{stream} :: [a] \to \mathit{Stream}\ a\\
&\mathit{stream}\ \mathit{xs} = D\ \mathit{uncons}\ \mathit{xs}\ \mathbf{where}\\
&\quad \mathit{uncons} :: [a] \to \mathit{Maybe}_2\ a\ [a]\\
&\quad \mathit{uncons}\ \mathit{xs} = \mathbf{if}\ \mathit{null}\ \mathit{xs}\ \mathbf{then}\ \mathit{Nothing}_2\ \mathbf{else}\ \mathit{Just}_2\ (\mathit{head}\ \mathit{xs})\ (\mathit{tail}\ \mathit{xs})\\
&\mathit{unstream} :: \mathit{Stream}\ a \to [a]\\
&\mathit{unstream}\ (D\ h\ s) = \mathit{unfoldr}\ h\ s\ \mathbf{where}\\
&\quad \mathit{unfoldr} :: (b \to \mathit{Maybe}_2\ a\ b) \to b \to [a]\\
&\quad \mathit{unfoldr}\ f\ y = \mathbf{case}\ f\ y\ \mathbf{of}\ \mathit{Nothing}_2 \to [\,];\ \mathit{Just}_2\ x\ y' \to x : \mathit{unfoldr}\ f\ y'
\end{aligned}
$$

For example, the familiar *map* on lists is retrieved by

$$\mathit{map}\ f = \mathit{unstream} \cdot \mathit{mapS}\ f \cdot \mathit{stream}$$

(In fact, *unstream* is essentially a specialisation of *tree*.)

4.4 Eliminating conversions

The crucial point in Coutts *et al.*'s work is the elimination of redundant conversions in the composition of list operations, such as in the composition of two maps:

$$\mathit{unstream} \cdot \mathit{mapS}\ f \cdot \mathit{stream} \cdot \mathit{unstream} \cdot \mathit{mapS}\ g \cdot \mathit{stream}$$

Here, the double conversion $\mathit{stream} \cdot \mathit{unstream}$ from streams to lists and back again is redundant. If it could be eliminated, the two occurrences of *mapS* would become adjacent; and because the definition of *mapS* is non-recursive, standard compiler optimisations — specifically, a case-of-case optimisation — can relatively easily fuse their bodies. The actual elimination of $\mathit{stream} \cdot \mathit{unstream}$ itself is easy, using the Glasgow Haskell Compiler's programmer-definable rewrite rules [42].

In fact, $\mathit{stream} \cdot \mathit{unstream}$ is not quite the identity: $\mathit{stream}\ (\mathit{unstream}\ \bot)$ equals $D\ \mathit{uncons}\ \bot$ rather than $\bot$. In practice, this difference does not cause a problem

if (a) the *Stream* datatype is not exported from the library, and (b) the library itself does not construct bottom values of type *Stream*; so their implementation is carefully arranged to satisfy these conditions.

Coutts *et al.* do claim (implicitly) that $stream\ (unstream\ (D\ h\ s)) = D\ h\ s$, but provide no proof of this claim; they say that "it is not entirely trivial to define a useful equivalence relation on streams [...] due to the fact that a single list can be modeled by infinitely many streams" [9, p320].

4.5 Destroying streams

In fact, there is a simple proof of the $stream \cdot unstream$ identity: it is an instance of the *destroy*/*unfoldr* rule [48], the dual of the better-known *foldr*/*build* rule [18]. The function *destroy* is defined as follows:

$$\begin{aligned} &destroy :: (\forall b.\ (b \to Maybe_2\ a\ b) \to b \to c) \to [a] \to c \\ &destroy\ g = g\ uncons \end{aligned}$$

so that $stream\ xs = destroy\ D\ xs$. Then the *destroy*/*unfoldr* rule states that

$$destroy\ g\ (unfoldr\ f\ s) = g\ f\ s$$

The proof of this rule is a straightforward application of Reynolds' parametricity [47]: the free theorem [57] of the type of the argument g of *destroy* is

$$\psi \cdot f = fmap\ f \cdot \phi \implies g\ \psi \cdot f = g\ \phi$$

Letting $\psi = uncons$ and $f = unfoldr\ \phi$ gives the *destroy*/*unfoldr* rule.

We note in passing that whereas *foldr*/*build* fusion has turned out to be a little disappointing in its applicability [33], *destroy*/*unfoldr* fusion seems to have a much wider scope. For example, it is straightforward to apply the latter technique to zip-like functions and functions with accumulating parameters [48], avoiding the need for *augment*-like generalisations [17]. This additional promise lends weight to our advocacy for greater appreciation of corecursive programming [16].

It is quite natural, so to speak, that equivalence proofs for abstract datatypes boil down to applications of parametricity. Intuitively, parametricity results capture "the only thing you can do, for type reasons". For example, for the abstract data structure $D\ h\ s$, the type of the representation s is hidden, and so "the only thing you can do, for type reasons", is to apply h to s. Indeed, Reynolds [46] originally called his result the "representation theorem", and motivated it by appeal to independence from a choice of representation: "We expect that the meaning of [...] a program will remain unchanged if the [definition of an abstract datatype] is altered by changing the representation of the type and redefining its primitive operations in a consistent manner" [46].

4.6 Unfolding observations

Another way of looking at Coutts *et al.*'s problem is to remember that streams are an abstract datatype, and so structural equivalence is the wrong tool to use; observational equivalence is what is needed.

In this case, since we have by definition that

$$stream\ (unstream\ (D\ h\ s)) = D\ uncons\ (unfoldr\ h\ s)$$

it suffices to show observational equivalence of $D\ h\ s$ and $D\ uncons\ (unfoldr\ h\ s)$. (Notice that these two streams will generally have different representations. The latter necessarily uses a list for the state, whereas the former may have an arbitrary state type.)

As we argued in Section 3.3, observational equivalence of abstract data structures is just structural equivalence of their unfoldings to the final coalgebra. For datatype $Stream\ a$, the signature is the functor $Maybe_2\ a$, whose final coalgebra is possibly infinite lists of as, and the operation to build such a list is the familiar but underappreciated *unfoldr* [16]. So we have to prove

$$unfoldr\ uncons\ (unfoldr\ h\ s) = unfoldr\ h\ s$$

This follows easily from the universal property

$$h = unfoldr\ f \iff uncons \cdot h = fmap\ h \cdot f$$

of *unfoldr*. This alternative proof using the universal property of unfold is important: as we shall see in Section 4.8, it seems to generalise better than the parametricity-based proof underlying *destroy*/*unfoldr*.

4.7 Streams that skip

The simple version above of Coutts *et al.*'s story uses a representation of streams providing a single observation, yielding either no information (for an empty stream) or a head and the state for a tail (for a non-empty stream). In fact, the complete story is more sophisticated, allowing a third outcome: a new state, but no head.

$$\textbf{data}\ Step\ a\ s = Done \mid Yield\ a\ s \mid Skip\ s$$
$$\textbf{type}\ SStream\ a = ADT\ (Step\ a)$$

The extra outcome is needed to support operations such as filtering, which do not produce an element at every step — when the filter discards an element from an underlying stream, or that stream skips itself, then the outer stream skips instead of yielding:

$$filterS :: (a \to Bool) \to SStream\ a \to SStream\ a$$
$$filterS\ p\ (D\ h\ s) = D\ (try\ h\ p)\ s\ \textbf{where}$$

```
try h p s = case h s of
  Done       → Done
  Skip s'    → Skip s'
  Yield x s' → if p x then Yield x s' else Skip s'
```

Without the possibility of skipping, the body of the stream would have to be recursive, thereby complicating fusion optimisations.

The conversions from lists is very similar to the simple case:

```
sstream :: [a] → SStream a
sstream xs = D unconsS xs where
  unconsS :: [a] → Step a [a]
  unconsS []       = Done
  unconsS (x : xs) = Yield x xs
```

The conversion back to lists is more involved, because of the need to handle skips:

```
unsstream :: SStream a → [a]
unsstream (D h s) = unfoldr (force h) s where
  force :: (s → Step a s) → (s → Maybe_2 a s)
  force h s = case h s of
    Done       → Nothing_2
    Yield x s' → Just_2 x s'
    Skip s'    → force h s'
```

Note that the body *force* of the unfold here is recursive, so it would be difficult for standard compiler optimisations to fuse a following function. That is not a big problem, because *unsstream* is intended to be used only when leaving the improved implementation of the list library, when fusion is not expected anyway. Moreover, note that *unsstream* may be unproductive, although for example even *filterS* (*const False*) is always productive: that particular lump in the carpet has been shuffled under the furniture, but no library reimplementation can eliminate it altogether.

4.8 Reasoning with skips

The presence of skips has interesting consequences for proofs. We should no longer take pure observational equivalence as the appropriate notion of equality on skipping streams, because we ought to treat some observationally distinguishable skipping streams as effectively equivalent. Coutts *et al.* say that "equivalence on streams should be defined modulo *Skip* values [...] semantics should not be affected by the presence or absence of *Skip* values" [9, p320].

For example, consider the stream version of the standard list function *concat*:

```
concatS :: SStream (SStream a) → SStream a
concatS (D hs ss) = D hc (Nothing, ss) where
```

$$
\begin{array}{l}
hc\ (Nothing, ss) = \mathbf{case}\ hs\ ss\ \mathbf{of} \\
\quad Done \qquad \rightarrow Done \\
\quad Skip\ ss' \quad \rightarrow Skip\ (Nothing, ss') \\
\quad Yield\ s\ ss' \rightarrow Skip\ (Just\ s, ss') \\
hc\ (Just\ (D\ ha\ sa), ss) = \mathbf{case}\ ha\ sa\ \mathbf{of} \\
\quad Done \qquad \rightarrow Skip\ (Nothing, ss) \\
\quad Skip\ sa' \quad \rightarrow Skip\ (Just\ (D\ ha\ sa'), ss) \\
\quad Yield\ y\ sa' \rightarrow Yield\ y\ (Just\ (D\ ha\ sa'), ss)
\end{array}
$$

In order to maintain a non-recursive body, this uses a rather complex internal state consisting of an optional *SStream a* and the internal state of a remaining *SStream* (*SStream a*); only if the former is present and yielding does the whole yield. We might expect the following property — one of the monad laws for streams — to hold:

$$concatS \cdot wrapS = id$$

where *wrapS* wraps an element up as a singleton stream:

$$
\begin{array}{l}
wrapS :: a \rightarrow SStream\ a \\
wrapS\ x = D\ fetch\ (Just\ x)\ \mathbf{where} \\
\quad fetch :: Maybe\ a \rightarrow Step\ a\ (Maybe\ a) \\
\quad fetch\ (Just\ x) = Yield\ x\ Nothing \\
\quad fetch\ Nothing = Done
\end{array}
$$

The two sides of the property are not even observationally equivalent, because the left-hand side $concatS \cdot wrapS$ introduces quite a few extra *Skip*s.

In fact, the appropriate notion of "equivalence modulo *Skip*s" is obtained precisely by taking structural equality on their unfoldings to lists:

$$unsstream \cdot concatS \cdot wrapS = unsstream \cdot id$$

The proof of this latter property is a fairly straightforward (albeit somewhat tedious) application of the universal property of *unfoldr*; it is relegated to an appendix (Section 7).

The alternative proof technique in terms of the universal property of *unfoldr* is important, because the *destroy*/*unfoldr* rule used in Section 4.5 does not seem to generalise nicely to skipping streams. The analogous development would be to introduce a function

$$
\begin{array}{l}
destroyS :: (\forall b.\ (b \rightarrow Step\ a\ b) \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow c \\
destroyS\ g = g\ unconsS
\end{array}
$$

so that $sstream\ xs = destroyS\ D\ xs$. The free theorem of the type of the argument g of *destroyS* is

$$\psi \cdot f = fmap\ f \cdot \phi \implies g\ \psi \cdot f = g\ \phi$$

but it isn't clear how to instantiate this equation to obtain the desired result; indeed, 'equivalence modulo *Skip*s' feels more ad hoc than parametric.

5 Conclusions

5.1 Related work

Data abstraction has long been recognised as a crucial tool in managing the complexity of software systems [37,30]. Pattern matching on algebraic datatypes is also widely appreciated as an extremely convenient technique [55]. But as Wadler's proposal [56] noted twenty years ago, it is difficult to marry the two together: data abstraction depends on hiding a data representation that pattern matching relies on revealing. There have been numerous other proposals for combining data abstraction with pattern matching over the years [5,35,6,34,51], and indeed a recent flurry of activity in the area [49,11,41,61].

One could look at *final coalgebra semantics* as a disciplined way of thinking about pattern matching over abstract datatypes. Rather than trying to force these two somewhat conflicting ideas together, one could instead define a *view* of codata (supporting abstraction) as data (supporting pattern matching), using the function *tree* from Section 3.3. In case a full transformation from 'completely codata' to 'completely data' is inappropriate, simply apply the body of the abstract datatype once:

$$\begin{aligned} &\mathit{unpack} :: \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f \to f\ (\mathit{ADT}\ f) \\ &\mathit{unpack}\ (D\ h\ s) = \mathit{fmap}\ (D\ h)\ (h\ s) \end{aligned}$$

This yields a piece of data (the outermost type constructor f) with codata as components (the inner occurrences of $\mathit{ADT}\ f$). This construction justifies a number of earlier attempts to treat algebraic datatypes abstractly [50,12,60,38].

The idea of using final coalgebras as the semantics of abstract datatypes has a long history. Wand [59] writes that "an abstract data type is a final object in the category of its representations", and Kamin [26] that "only externally observable behavior matters [...] the final data type is the most abstract realization of any given data abstraction." Considering how close the relationship between abstract datatypes and object-oriented classes is, it is surprising that it seems to have taken over a decade for the idea to arise that final coalgebras provide a semantics for classes too [45,23]. For a good historical review of coinduction for behavioural satisfaction, see [20].

5.2 Summary

We have presented an approach to reasoning about abstract datatypes in a functional language, based on the (well-known) model of abstraction through existential quantification over the hidden representation type [32,28,29], and the (somewhat less well-known and appreciated) reasoning principles for codata through universal properties of final coalgebras [16,15]. We have illustrated this approach by considering a problem arising from Coutts *et al.*'s work [9] on stream fusion. In a nutshell, we advocate the following steps for reasoning about abstract datatypes:

– express the signature as a (strictly positive) functor f;
– enforce the abstraction via existential quantification:

$$\textbf{data}\ \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f = \exists s.\ D\ (s \to f\ s)\ s$$

– capture observations as concrete data:

$$\textbf{data}\ \mathit{Tree}\ f = T\{\mathit{unT} :: f\ (\mathit{Tree}\ f)\}$$

– transform abstract data to concrete data:

$$\begin{aligned}&\mathit{tree} :: \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f \to \mathit{Tree}\ f\\&\mathit{tree}\ (D\ h\ s) = \mathit{unfold}\ h\ s\ \textbf{where}\\&\quad \mathit{unfold} :: \mathit{Functor}\ f \Rightarrow (a \to f\ a) \to a \to \mathit{Tree}\ f\\&\quad \mathit{unfold}\ f\ x = T\ (\mathit{fmap}\ (\mathit{unfold}\ f)\ (f\ x))\end{aligned}$$

– exploit the universal property of *unfold* for reasoning:

$$h = \mathit{unfold}\ f \iff \mathit{unT} \cdot h = \mathit{fmap}\ h \cdot f$$

– view data as a mixture of concrete and abstract:

$$\mathit{unpack} :: \mathit{Functor}\ f \Rightarrow \mathit{ADT}\ f \to f\ (\mathit{ADT}\ f)$$

5.3 Acknowledgements

Particular thanks are due to Duncan Coutts, whose talk about his paper [9] with Roman Leshchinskiy and Don Stewart posed the question that inspired this work. Richard Bird's paper [1] was also an influence, not least on the title. I would also like to thank the Algebra of Programming group at Oxford and Pablo Nogueira, for helpful contributions and discussions, and the anonymous reviewers, whose comments have led to significant improvements.

References

1. Richard Bird. Unfolding pointer algorithms. *Journal of Functional Programming*, 11(3):347–358, 2001.
2. Richard S. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.
3. Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
4. Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.
5. F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, 1993.

6. Warren Burton, Erik Meijer, Patrick Sansom, Simon Thompson, and Phil Wadler. Views: An extension to Haskell pattern matching. http://www.haskell.org/development/views.html, October 1996.
7. Robin Cockett and Tom Fukushima. About Charity. Department of Computer Science, University of Calgary, May 1992.
8. William R. Cook. Object-oriented programming versus abstract data types. In *REX Workshop/School on the Foundations of Object-Oriented Languages (FOOL)*, volume 489 of *Lecture Notes in Computer Science*, pages 151–178, 1990.
9. Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *International Conference on Functional Programming*, pages 315–326, 2007.
10. Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, October 1972. See http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html.
11. Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming*, volume 4609 of *LNCS*, pages 273–298, 2007.
12. Martin Erwig. Categorical programming with abstract data types. In *Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 406–421, 1998.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
14. Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
15. Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, April/May 2005.
16. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.
17. Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow, 1998.
18. Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, 1993.
19. Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 1989. Available online at http://www.monad.me.uk/stable/Proofs+Types.html.
20. Joseph Goguen and Grant Malcolm. Hidden coinduction: Behavioural correctness proofs for objects. *Mathematical Structures in Computer Science*, 9:287–319, 1999.
21. Andrew D. Gordon. A tutorial on co-induction and functional programming. In *Glasgow Workshop on Functional Programming*, 1994.
22. Denis Howe. Free on-line dictionary of computing. foldoc.org, 1993.
23. Bart Jacobs. Objects and classes, coalgebraically. In B. Freitag, C. B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer, 1996.
24. Bart Jacobs. Coalgebras in specification and verification for object-oriented languages. *Newsletter of the Dutch Association for Theoretical Computer Science (NVTI)*, (3):15–27, 1999.

25. Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, (62):222–259, 1997.
26. Samuel Kamin. Final data types and their specification. *ACM Transactions on Programming Languages and Systems*, 5(1):97–123, 1983.
27. Richard B. Kieburtz. Codata and comonads in Haskell. Unpublished manuscript, Oregon Graduate Institute, 1999.
28. Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *SIGPLAN Workshop on ML and its Applications*, June 1992.
29. Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.
30. Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM Press.
31. John C. Mitchell. On the equivalence of data representations. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–329. Academic Press Professional, Inc., San Diego, CA, USA, 1991.
32. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.
33. Laszlo Németh. *Catamorphism Based Program Transformations for Non-Strict Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, 2000.
34. Chris Okasaki. Views for Standard ML. In *SIGPLAN Workshop in ML*, pages 14–23, 1998.
35. Pedro Palao Gostanza, Ricardo Peña, and Manual Núñez. A new look at pattern matching in abstract data types. In *International Conference on Functional Programming*, pages 110–121. ACM, 1996.
36. David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.
37. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
38. Ross Paterson. Haskell hierarchical libraries: Data.Sequence. http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Sequence.html, Accessed 2007.
39. Simon Peyton Jones. Explicit quantification in Haskell. http://research.microsoft.com/~simonpj/Haskell/quantification.html, 1998.
40. Simon Peyton Jones. *The Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
41. Simon Peyton Jones. View patterns: Lightweight views for Haskell. http://hackage.haskell.org/trac/ghc/wiki/ViewPatternsArchive, January 2007.
42. Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *Haskell Workshop*, 2001.
43. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
44. Gordon Plotkin and Martin Abadi. A logic for parametric polymorphism. In *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, 1993.
45. Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 1995.

46. John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.
47. John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier, 1983.
48. Josef Svenningsson. Shortcut fusion for accumulating parameters and zip-like functions. In *International Conference on Functional Programming*, 2002.
49. Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *International Conference on Functional Programming*, pages 29–40, 2007.
50. Simon Thompson. Higher-order + polymorphic = reusable. Technical Report 224, University of Kent at Canterbury, May 1997. http://www.cs.kent.ac.uk/pubs/1997/224/.
51. Mark Tullsen. First class patterns. In *Practical Aspects of Declarative Languages*, volume 1753 of *Lecture Notes in Computer Science*, pages 1–15, 2000.
52. David A. Turner. Elementary strong functional programming. In Pieter H. Hartel and Rinus Plasmeijer, editors, *Functional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.
53. David A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.
54. David Ungar and Randall B. Smith. Self: The power of simplicity. In *Object-Oriented Programming: Systems, Languages and Applications*, pages 227–242, 1987.
55. Philip Wadler. A critique of Abelson and Sussman: Why calculating is better than scheming. *SIGPLAN Notices*, 22(3):8, 1987.
56. Philip Wadler. Views: A way for mattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, pages 307–313. ACM, 1987.
57. Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.
58. Philip Wadler. Recursive types for free! Unpublished manuscript, http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt, July 1990.
59. Mitchell Wand. Final algebra semantics and data type extensions. *Journal of Computer and System Sciences*, 19:27–44, 1979.
60. Daniel C. Wang and Tom Murphy, VII. Programming with recursion schemes. Unpublished manuscript, Agere Systems/Carnegie Mellon University, 2002.
61. Meng Wang and Jeremy Gibbons. Translucent abstraction: Algebraic datatypes with safe views. Submitted for publication, April 2008.
62. Gavin Wraith. A note on categorical datatypes. In D. H. Pitt, D. E. Rydeheard, P. Dyjber, A. M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

6 Appendix: Equivalence of complex numbers

Section 3.4 shows how to prove observational equivalence of different implementations of complex numbers, using the fusion property of *unfold*. Here, we discharge the proof obligation

$$fc \cdot p2c = fmap\ p2c \cdot fp$$

using the (idealised, since not quite valid in approximate floating point numbers) equation relating polar and Cartesian representations

$$p2c \cdot c2p = id$$

and the definitions

$$\begin{aligned}
fc\,(x,y) = CF\{&_new = \lambda z \to z,\\
&_add = \lambda c \to (x + re\ c, y + im\ c),\\
&_rea = x,\\
&_ima = y\})\\
fp\ p \quad = CF\{&_new = \lambda z \to c2p\ z,\\
&_add = \lambda c \to \mathbf{let}\ (x,y) = p2c\ p\ \mathbf{in}\\
&\qquad\qquad c2p\ (x + re\ c, y + im\ c),\\
&_rea = fst\ (p2c\ p),\\
&_ima = snd\ (p2c\ p)\})
\end{aligned}$$

Note also that the appropriate definition of *fmap* on *ComplexF* is

$$\begin{aligned}
fmap\ f\ c = CF\{&_new = \lambda z \to f\ (_new\ c\ z),\\
&_add = \lambda c' \to f\ (_add\ c\ c'),\\
&_rea = _rea\ c,\\
&_ima = _ima\ c\}
\end{aligned}$$

We calculate:

$$\begin{aligned}
&fmap\ p2c\ (fp\ p)\\
=\ &\{\ \ fp\ \ \}\\
&fmap\ p2c\ (CF\{_new = \lambda z \to c2p\ z,\\
&\qquad\qquad _add = \lambda c\ \to \mathbf{let}\ (x,y) = p2c\ p\ \mathbf{in}\\
&\qquad\qquad\qquad\qquad c2p\ (x + re\ c, y + im\ c),\\
&\qquad\qquad _rea = fst\ (p2c\ p),\\
&\qquad\qquad _ima = snd\ (p2c\ p)\})\\
=\ &\{\ \ fmap \text{ on } ComplexF\ \ \}\\
&CF\{_new = \lambda z \to p2c\ (c2p\ z),\\
&\quad _add = \lambda c\ \to \mathbf{let}\ (x,y) = p2c\ p\ \mathbf{in}\\
&\qquad\qquad p2c\ (c2p\ (x + re\ c, y + im\ c)),\\
&\quad _rea = fst\ (p2c\ p),\\
&\quad _ima = snd\ (p2c\ p)\}\\
=\ &\{\ \ p2c \cdot c2p = id\ \ \}\\
&CF\{_new = \lambda z \to z,\\
&\quad _add = \lambda c\ \to \mathbf{let}\ (x,y) = p2c\ p\ \mathbf{in}\\
&\qquad\qquad (x + re\ c, y + im\ c),\\
&\quad _rea = fst\ (p2c\ p),\\
&\quad _ima = snd\ (p2c\ p)\}\\
=\ &\{\ \ \text{lift out the } \mathbf{let} \text{ binding}\ \ \}\\
&\mathbf{let}\ (x,y) = p2c\ p\ \mathbf{in}
\end{aligned}$$

$$
\begin{array}{l}
\quad CF\{\,\_new = \lambda z \to z, \\
\qquad\; \_add = \lambda c \to (x + re\ c, y + im\ c), \\
\qquad\; \_rea = x, \\
\qquad\; \_ima = y\,\} \\
= \quad \{\quad fc \quad\} \\
\quad \textbf{let}\ (x, y) = p2c\ p\ \textbf{in}\ fc\ (x, y) \\
= \quad \{\quad \text{application} \quad\} \\
\quad fc\ (p2c\ p)
\end{array}
$$

which completes the proof.

7 Appendix: Equivalence of skipping streams

Section 4.8 makes a claim about the observational equivalence modulo *Skip*s of the skipping streams $concatS\ (wrapS\ s)$ and s, where

$$
\begin{array}{l}
concatS :: SStream\ (SStream\ a) \to SStream\ a \\
concatS\ (D\ hs\ ss) = D\ hc\ (Nothing, ss)\ \textbf{where} \\
\quad hc\ (Nothing, ss) = \textbf{case}\ hs\ ss\ \textbf{of} \\
\qquad Done \qquad \to Done \\
\qquad Skip\ ss' \qquad \to Skip\ (Nothing, ss') \\
\qquad Yield\ s\ ss' \to Skip\ (Just\ s, ss') \\
\quad hc\ (Just\ (D\ ha\ sa), ss) = \textbf{case}\ ha\ sa\ \textbf{of} \\
\qquad Done \qquad \to Skip\ (Nothing, ss) \\
\qquad Skip\ sa' \qquad \to Skip\ (Just\ (D\ ha\ sa'), ss) \\
\qquad Yield\ y\ sa' \to Yield\ y\ (Just\ (D\ ha\ sa'), ss) \\
wrapS :: a \to SStream\ a \\
wrapS\ x = D\ fetch\ (Just\ x)\ \textbf{where} \\
\quad fetch\ (Just\ x) = Yield\ x\ Nothing \\
\quad fetch\ Nothing = Done
\end{array}
$$

The claim boils down to the following equation between functions on lists,

$$unsstream \cdot concatS \cdot wrapS = unsstream$$

where

$$
\begin{array}{l}
unsstream :: SStream\ a \to [a] \\
unsstream\ (D\ h\ s) = unfoldr\ (force\ h)\ s\ \textbf{where} \\
\quad force\ h\ s = \textbf{case}\ h\ s\ \textbf{of}\ Done \qquad \to Nothing_2 \\
\qquad\qquad\qquad\qquad\quad Yield\ x\ s' \to Just_2\ x\ s' \\
\qquad\qquad\qquad\qquad\quad Skip\ s' \qquad \to force\ h\ s'
\end{array}
$$

Consider for example the skipping stream $s = D\ h\ 0$ where

$$h\ n = [Skip\ 1, Yield\ \texttt{'a'}\ 2, Skip\ 3, Yield\ \texttt{'b'}\ 4, Skip\ 5, Done] \mathbin{!!} n$$

(The operation '!!' denotes list indexing.) Unwinding this stream proceeds through each of the above states in turn, yielding in total the list of characters `['a','b']`. The stream $concatS\ (wrapS\ s)$, on the other hand, exhibits the following behaviour:

state	output
$(Nothing, Just\ (D\ h\ 0))$	$Skip$
$(Just\ (D\ h\ 0), Nothing)$	$Skip$
$(Just\ (D\ h\ 1), Nothing)$	$Yield$ `'a'`
$(Just\ (D\ h\ 2), Nothing)$	$Skip$
$(Just\ (D\ h\ 3), Nothing)$	$Yield$ `'b'`
$(Just\ (D\ h\ 4), Nothing)$	$Skip$
$(Just\ (D\ h\ 5), Nothing)$	$Skip$
$(Nothing, Nothing)$	$Done$

Each row of the table presents a state and the output from that state, omitting the successor state if present. For example,

$$hc\ (Just\ (D\ h\ 1), Nothing) = Yield\ \texttt{'a'}\ (Just\ (D\ h\ 2), Nothing)$$

Evidently the closest match between these states and those of the original stream s are those whose first component is a $Just$. We therefore proceed to show that

$$\begin{array}{ll} & unsstream\ (concatS\ (wrapS\ s)) \\ = & \{\!\{\ \text{definitions}\ \}\!\} \\ & unfoldr\ (force\ (hc\ fetch))\ (Nothing, Just\ (D\ h\ n)) \\ = & \{\!\{\ \text{expanding: } f\ x = f\ y \Longrightarrow unfoldr\ f\ x = unfoldr\ f\ y\ \}\!\} \\ & unfoldr\ (force\ (hc\ fetch))\ (Just\ (S\ h\ n), Nothing) \\ = & \{\!\{\ unfoldr\ \text{fusion}\ \}\!\} \\ & unfoldr\ (force\ h)\ n \\ = & \{\!\{\ \text{definitions}\ \}\!\} \\ & unsstream\ (D\ h\ n) \end{array}$$

For the 'expansion' step, it is easy to verify that

$$fetch\ (Just\ (D\ h\ n)) = Yield\ (D\ h\ n)\ Nothing$$

and so

$$hc\ fetch\ (Nothing, Just\ (D\ h\ n)) = Skip\ (Just\ (D\ h\ n), Nothing)$$

and so

$$\begin{array}{ll} & force\ (hc\ fetch)\ (Nothing, Just\ (D\ h\ n)) \\ = & \\ & force\ (hc\ fetch)\ (Just\ (D\ h\ n), Nothing) \end{array}$$

as required.

For the 'fusion' step, we define

$$inject\ n = (Just\ (D\ h\ n), Nothing)$$

so that the obligation is to prove

$$unfoldr\ (force\ (fc\ fetch)) \cdot inject = unfoldr\ (force\ h)$$

This can be done using the fusion rule for *unfoldr*:

$$unfoldr\ f \cdot g = unfoldr\ f' \iff f \cdot g = fmap\ (prod\ id\ g) \cdot f'$$

which reduces the obligation to showing that

$$force\ (hc\ fetch) \cdot inject = fmap\ (prod\ id\ inject) \cdot force\ h$$

This last step has to be done using fixpoint induction, because *force* is not defined using a structured form of recursion. We simplify both sides to the point at which they make a recursive call to *force*; the surrounding contexts turn out to be equal, and so fixpoint induction shows that the least fixpoints are equal. On the left-hand side, we have:

$$\begin{array}{ll}
 & force\ (hc\ fetch)\ (inject\ n) \\
= & \quad \{\!\!\{\ inject\ \}\!\!\} \\
 & force\ (hc\ fetch)\ (Just\ (D\ h\ n), Nothing) \\
= & \quad \{\!\!\{\ force, hc, fetch\ \}\!\!\} \\
 & \textbf{case}\ h\ n\ \textbf{of} \\
 & \quad Done \rightarrow Nothing_2 \\
 & \quad Skip\ m \rightarrow force\ (hc\ fetch)\ (Just\ (D\ h\ m), Nothing) \\
 & \quad Yield\ x\ m \rightarrow Just_2\ x\ (Just\ (D\ h\ m), Nothing) \\
= & \quad \{\!\!\{\ inject\ \}\!\!\} \\
 & \textbf{case}\ h\ n\ \textbf{of} \\
 & \quad Done \rightarrow Nothing_2 \\
 & \quad Skip\ m \rightarrow force\ (hc\ fetch)\ (inject\ m) \\
 & \quad Yield\ x\ m \rightarrow Just_2\ x\ (inject\ m)
\end{array}$$

On the right, we have:

$$\begin{array}{ll}
 & fmap\ (prod\ id\ inject)\ (force\ h\ n) \\
= & \quad \{\!\!\{\ force\ \}\!\!\} \\
 & fmap\ (prod\ id\ inject)\ (\textbf{case}\ h\ n\ \textbf{of} \\
 & \quad Done \rightarrow Nothing_2 \\
 & \quad Skip\ m \rightarrow force\ h\ m \\
 & \quad Yield\ x\ m \rightarrow Just_2\ x\ m) \\
= & \quad \{\!\!\{\ fmap, prod\ \}\!\!\} \\
 & \textbf{case}\ h\ n\ \textbf{of} \\
 & \quad Done \rightarrow Nothing_2 \\
 & \quad Skip\ m \rightarrow fmap\ (prod\ id\ inject)\ (force\ h\ m) \\
 & \quad Yield\ x\ m \rightarrow Just_2\ x\ (inject\ m)
\end{array}$$

This completes the proof.