Slicing It

Indexed Containers in Haskell (Programming Pearl)

Conor McBride
University of Strathclyde
conor@cis.strath.ac.uk

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

This paper is about datatypes in Haskell. In particular, it concerns container-like data—structures parametrized by a notion of *element*, like lists, labelled trees, expressions (where 'elements' play the role of variables) and many others. In Haskell, such structures are often manifested as *type constructors* of *kind* $* \to *$, and made instances of the Functor type class:

```
class Functor (φ :: * → *) where
  fmap :: (α → β) → (φ α → φ β)
```

where the fmap method supports the systematic transformation of elements without disturbing the structure in which they sit.[1]

Here, however, I investigate what happens if we index the container structures (lists with length, trees with height, terms with type) but also the notions of element (typed variables in typed terms, the different sorts of substructure in a labelled rose tree). If we want to gather these creatures and equip them with their common structure, we shall need a new type class,

```
class IFunctor (φ :: ({ι} → *) → ({o} → *)) where ...
```

for type constructors taking indexed sets to indexed sets. But I rush ahead too quickly: we shall need to think afresh about indexed sets, their kinds and their uses, if we are to develop good tools. This paper contributes

- an extension of Haskell's kind system with *lifted types*, e.g. {Either α β}, suitable for describing types indexed by data, and the corresponding extension of types with *lifted terms*, e.g. {L *a*}, {R *b*}, and just the corresponding *polymorphism*;
- a type class IFunctor, slicing Functor into its polymorphic collection of indexed counterparts;
- a repertoire of IFunctor instances, closed not only under identity, composition, sum, and product, but also under *fixpoints*, suitable for developing generic operations on mutually defined collections of GADTs;
- a peek at the corresponding notion of IMonad, and its relationship with the "parametrized monads" popularised by Atkey and others [**?**];
- a translation from the language extensions in this paper back to GHC Haskell *now*, forgetting discipline but preserving functionality.

The code in this paper may look like fantasy Haskell, but it all compiles with help from the *Strathclyde Haskell Enhancement* [**?**], a preprocessor for the Glasgow Haskell Compiler. SHE comes bundled with this paper's examples, and a variety other experiments.

1.1 Some Related Work

More detailed discussion of related work must necessarily await its technical context, but I feel I should acknowledge my primary debts on the front page.

The programs in this paper builds on the theoretical study of *indexed containers* by Altenkirch, Morris, and colleagues [**?**]. They show that Haskell, viewed through a suitable lens, already has enough dependent types to exploit many of the insights of that rich seam of research.

As an exercise in modelling collections of datatypes, this work owes much to the 'Oxford origami' style of generic programming, based on folds and unfolds for fixpoints at functors of various arities, and so on [**?**, **?**]. Indexing allows us to abstract uniformly over the arities and simplify the general treatment: one map, one fold, one unfold.

Indeed, Rodriguez, Holdermans, Jeuring, and Löh have shown how to to model bunches of mutually defined simple types by taking fixpoints at kind $(* \to *) \to (* \to *)$ [**?**]. This paper can be seen as a refinement of the basic idea behind their work, providing a tighter analysis of ths kinds involved, a local fixpoint operator rather than top-level recursion, and a generalization to GADTs.

1.2 Overview

I begin with revision of essential background material: laws of exponentiation (i.e., index manipulation), Haskell's kind system, the fixpoints-of-functors presentation of recursive datatypes and the fixpoints-of-bifunctors presentation of recursive functors. I then explore the connection between mutual recursion and GADTs. Together, these motivate the generalization to indexed functors:

[1] There are instances of Functor which do not have 'elements in positions' and are thus not containers — continuations are the classic example. Grasping the notion of position is easy with dependent types [Abbott et al.(2005)Abbott, Altenkirch, and Ghani], but ugly in Haskell, so I defer its treatment until Haskell improves.

I explore some specific examples, and some generic technology. With a look ahead to the technology which seems likely to emerge as a consequence of shifting to the indexed world, I conclude.

2. Watch Out for Exponentials!

Let me remind you of some laws of exponentials from high school algebra:

$$\begin{aligned} 1 &= X^0 \\ X &= X^1 \\ X^\alpha \times X^\beta &= X^{\alpha+\beta} \\ (X^\alpha)^\beta &= X^{\alpha\times\beta} \end{aligned}$$

These laws (viewed left-to-right) show how to express various formulae as exponentials. At various points in our journey, we shall need to do just that, shifting structure into the exponent. I encourage you to watch out for opportunities to exponentiate. In Haskell, however, the exponential X^α is more likely to be a function space $\alpha \to X$, and the exponents types like Void (the type with no defined values), $()$, Either α β, (α, β). It is nothing new to do high school algebra with types, but the twist here will be to do algebra with *kinds* — the X in this paper will sometimes be the kind $*$ which types inhabit. How can a type be the exponent of a kind? Haskell's kinds are given by the simple grammar

$$\kappa = * \mid \kappa \to \kappa$$

giving kinds for types and type constructors. Perhaps we may need to increase the power of the kind system.

3. Functor, Fix, fmap, fold

Let us recall the construction of recursive datatypes as fixpoints of polynomial functors. For example, consider the simple datatype of expressions with integers and addition:

```
data Exp = Val Int | Add Exp Exp
```

We can equip Exp with a fold (or iterator, or catamorphism) explaining how to compute a value in τ from an Exp, given an alternative τ-based implementation of Val and Add.

```
foldExp :: (Int → τ) → (τ → τ → τ) → Exp → τ
foldExp v a (Val i)   = v i
foldExp v a (Add s t) = a (foldExp v a s) (foldExp v a t)
```

The recipe for fold is 'for each constructor, apply fold recursively to the substructures, then process the results with the alternative implementation'. We can express this recipe directly, if we can find a way to address the 'substructures' uniformly. One way to do this is to express the definition of Exp as a container of its substructures, yielding a type constructor which is readily made an instance of the Functor class, so that fmap means 'apply to substructures'.

```
data ExpF α = ValF Int | AddF α α
instance Functor ExpF where
  fmap f (ValF i)   = ValF i
  fmap f (AddF s t) = AddF (f s) (f t)
```

We can recover our Exp type as the ExpF instance of a *general* fixpoint construction

```
newtype Fix ϕ = In (ϕ (Fix ϕ))

type    Exp     = Fix ExpF
pattern Val i   = In (ValF i)
pattern Add s t = In (AddF s t)
```

I have added some *pattern synonyms* to show we code the original constructors, now that containment and recursion have been separated. Pattern synonyms, first proposed by Aitken and Reppy [Aitken and Reppy(1992)], are non-recursive definitions whose right-hand sides are restricted to be patterns, linear in the variables bound on the left. SHE supports pattern synonyms, fully applied in patterns, and partially applied in expressions.

Now that we have rebuilt Exp, we can equip it with specific functionality just as before. However, we get $\mathsf{fold}_{\mathsf{Exp}}$ for free, by instantiating the general fold — an 'alternative τ-based implementation' becomes a single function in $\phi\ \tau \to \tau$, sometimes called a ϕ-*algebra with carrier* τ.

```
fold :: Functor ϕ ⇒ (ϕ τ → τ) → Fix ϕ → τ
fold f (In rs) = f (fmap (fold f) rs)
```

Pattern synonyms are not official Haskell, but the notational convenience they lend this compositional approach to data structures is too good to miss. We can even go further, rebuilding ExpF from the *polynomial functor kit*:

```
newtype I       α = I α
newtype K τ     α = K τ
newtype (:+:) ϕ θ α = Sum (Either (ϕ α) (θ α))
newtype (:×:) ϕ θ α = Prod (ϕ α, θ α)

type    ExpF     = K Int :+: I :×: I
pattern ValF i   = Sum (L (K i))
pattern AddF s t = Sum (R (Prod (I s, I t)))
```

This kit generates any one-parameter type constructor given by a polynomial in one variable. By declaring a fixed set of instances, we can show that all of these polynomials have lots of useful structure: they all preserve decidability of equality, they are all differentiable, they are Functors and Traversable to boot!

Exercise. Use **newtype** isomorphism and the laws of exponentiation to show that our two versions of $\mathsf{fold}_{\mathsf{Exp}}$ have isomorphic types, i.e., that

```
  (Int → τ) → (τ → τ → τ) → Exp → τ
≡ (ExpF τ → τ) → Exp → τ
```

4. Refunctor Your Bifactor

Fixpoints of polynomials in one variable are good for constructing recursive types, but what if we want to construct recursive parametrized types, making clear that they are functors? We can already define lists as follows:

```
type    ListF α   = K () :+: K α :×: I
type    List α   = Fix (ListF α)
pattern Nil      = Sum (L (K ()))
pattern Cons x xs = Sum (R (Prod (K x, I xs)))
```

This entitles us to fold for lists, but it does not give us an fmap to lift operations on *elements*. Rather, ListF α is seen as a container only of *sublists*. The fix for this problem is standard [**?**, **?**]: switch to working with containers for *two* sorts of data, 'elements' and 'sublists'. We shall need the Bifunctor kit. The Bifunctor class

```
class Bifunctor (ϕ :: * → * → *) where
  bimap :: (α → α') → (β → β') → ϕ α β → ϕ α' β'
```

is closed under the following constructions:

```
newtype Fst       α β = Fst α
newtype Snd       α β = Snd β
newtype K2 τ      α β = K2 τ
newtype (:+:) ϕ θ α β = Sum2 (Either (ϕ α β) (θ α β))
newtype (:×:) ϕ θ α β = Prod2 (ϕ α β, θ α β)
```

Now we can expose the container structure of lists:

```
type ListB = K2 () :+: Fst :×: Snd
```

yielding

$$\mathsf{ListB}\ \alpha\ \alpha\sigma\ \cong\ \mathsf{Either}\ ()\ (\alpha, \alpha\sigma)$$

To get recursive parametrized datatypes, we take the fixpoint of the bifunctor in its second argument, retaining the first argument as a parameter:[2]

$$\textbf{newtype}\ \mathsf{PFix}\ \phi\ \alpha = \mathsf{PIn}\ (\phi\ \alpha\ (\mathsf{PFix}\ \phi\ \alpha))$$

Now we can finish our construction of lists:

$$\begin{array}{ll}\textbf{type}\ [\,] & = \mathsf{PFix}\ \mathsf{ListB}\\ pat\ \ [\,] & = \mathsf{PIn}\ (\mathsf{Sum}_2\ (\mathsf{L}\ (\mathsf{K}_2\ ())))\\ pat\ \ x : xs & = \mathsf{PIn}\ (\mathsf{Sum}_2\ (\mathsf{R}\ (\mathsf{Prod}_2\ (\mathsf{Fst}\ x, \mathsf{Snd}\ xs))))\end{array}$$

The payoff from our bifunctorial construction is that we get both 'map' and 'fold', once for all.

$$\begin{array}{l}\mathsf{pfold} :: \mathsf{Bifunctor}\ \phi \Rightarrow (\phi\ \alpha\ \tau \to \tau) \to \mathsf{PFix}\ \phi\ \alpha \to \tau\\ \mathsf{pfold}\ f\ (\mathsf{PIn}\ xrs) = f\ (\mathsf{bimap}\ id\ (\mathsf{pfold}\ f)\ xrs)\end{array}$$

$$\begin{array}{l}\textbf{instance}\ \mathsf{Bifunctor}\ \phi \Rightarrow \mathsf{Functor}\ (\mathsf{PFix}\ \phi)\ \textbf{where}\\ \quad \mathsf{fmap}\ f\ (\mathsf{PIn}\ xrs) = \mathsf{PIn}\ (\mathsf{bimap}\ f\ (\mathsf{fmap}\ f)\ xrs)\end{array}$$

We shall revisit this construction later, in generalized form. The key point is to distinguish the 'recursive stuff' bound by the fixpoint from the 'parametric stuff' over which we should still expect to be functorial.

Except occasionally and by accident, I am not a category theorist, but if I were, I should be a little disturbed by this rigmarole. I might point out that a bifunctor is *just*[3] a functor from a product category. It is perhaps a little hasty of Haskellers to spend the name Functor on only the endofunctors of the category $*$, when there surely are other Haskell categories, and functors between them.

But perhaps we might recover the functorial nature of Bifunctor with a bit of algebra. The laws of exponentiation suggest that we might recover the treatment via product categories if only we had product kinds. However, we have only exponentials. We could replace $* \times *$ by an exponential

$$* \to * \to *\ \cong\ (*, *) \to *\ \cong\ (\{\mathsf{Bool}\} \to *) \to *$$

if only we had a kind $\{\mathsf{Bool}\}$ with two inhabitants $\{\mathsf{T}\}$ and $\{\mathsf{F}\}$. Let's just pretend, for now, that $\{\mathsf{Bool}\}$ exists. To rebuild the Bifunctor machinery, we shall need to pack up any pair of types as a $\{\mathsf{Bool}\}$-indexed type. We can do this with a GADT playing the role of **if** ... **then** ... **else**..., but with the condition coming last.

$$\begin{array}{l}\textbf{data}\ (:??:) :: * \to * \to \{\mathsf{Bool}\} \to *\ \textbf{where}\\ \quad \mathsf{InT} :: t \to (t :??: f)\ \{\mathsf{T}\}\\ \quad \mathsf{InF} :: f \to (t :??: f)\ \{\mathsf{F}\}\end{array}$$

Correspondingly, we can pack up a pair of functions as an *index-preserving* function between the corresponding TorFs.

$$\begin{array}{l}(<??>) :: (\alpha \to \alpha') \to (\beta \to \beta')\\ \qquad \to (\alpha :??: \beta)\ b \to (\alpha' :??: \beta')\ b\\ (tr <??> fa)\ (\mathsf{InT}\ t) = \mathsf{InT}\ (tr\ t)\\ (tr <??> fa)\ (\mathsf{InF}\ f) = \mathsf{InF}\ (fa\ f)\end{array}$$

These index-preserving functions provide us with an appropriate notion of morphism between indexed types. Let us name them:

$$\textbf{type}\ \sigma \rightarrowtail \tau = \forall\ i.\ \sigma\ i \to \tau\ i$$

Conveniently, we have that $\mathsf{id} :: \sigma \rightarrowtail \sigma$, and that if $f :: \sigma \rightarrowtail \tau$ and $g :: \rho \rightarrowtail \sigma$, then $f.\ g :: \rho \rightarrowtail \tau$. This composition is, of course, associative and absorbs id on either side. We may consider $\{\mathsf{Bool}\} \to *$ with $\rightarrowtail$ to be a category! Let us be tidy, and observe that

$$(<??>) :: (\alpha \to \alpha') \to (\beta \to \beta') \to \alpha :??: \beta \rightarrowtail \alpha' :??: \beta'$$

We are now in a position to see Bifunctors as functors between $\{\mathsf{Bool}\} \to *$ and $* -$ we must explain how to take $\rightarrowtail$ morphisms and return ordinary functions.

$$\begin{array}{l}\textbf{class}\ \mathsf{BooFunctor}\ (\phi :: (\{\mathsf{Bool}\} \to *) \to *)\ \textbf{where}\\ \quad \mathsf{boomap} :: (\sigma \rightarrowtail \tau) \to (\phi\ \sigma \to \phi\ \tau)\end{array}$$

This looks just like the declaration for Functor, except that we have different source and target categories, and we use the relevant morphisms for each. When we build BooFunctors, we use the choice of index to select which sorts of substructure to put where. Here, ListBoo does the same job as ListB, with $\{\mathsf{T}\}$ indicating list element positions and $\{\mathsf{F}\}$ indicating positions for sublists.

$$\begin{array}{l}\textbf{data}\ \mathsf{ListBoo} :: (\{\mathsf{Bool}\} \to *) \to *\ \textbf{where}\\ \quad \mathsf{NilBoo}\ \ :: \qquad\qquad\qquad\ \ \mathsf{ListBoo}\ \tau\\ \quad \mathsf{ConsBoo} :: \tau\ \{\mathsf{T}\} \to \tau\ \{\mathsf{F}\} \to \mathsf{ListBoo}\ \tau\\ \textbf{instance}\ \mathsf{BooFunctor}\ \mathsf{ListBoo}\ \textbf{where}\\ \quad \mathsf{boomap}\ tf\ \mathsf{NilBoo} \qquad\ = \mathsf{NilBoo}\\ \quad \mathsf{boomap}\ tf\ (\mathsf{ConsBoo}\ x\ xs) = \mathsf{ConsBoo}\ (tf\ x)\ (tf\ xs)\end{array}$$

Note that tf must be polynorphic enough to be used on substructures of either kind — that is, it must actually represent a pair of functions, just as in bimap.

If you suspect that BooFunctor looks too specific to be the basis of a good library, you would be right. BooFunctor is just a refactoring of Bifunctor, using indexing to collect the two parameters. We could readily replace $\{\mathsf{Bool}\}$ with $\{\mathsf{Maybe\ Bool}\}$ if we wanted a third parameter, but it might be preferable not to have a further **class** MayBooFunctor to capture the appropriate structure. We shall clearly need to look for a suitable generalization. Before we do so, however, let us look at another opportunity for indexing.

5. Mutual Datatypes as Indexed Structures

We have just seen how indexing the parameter to a functor models multiple sorts of substructure. We may equally well index the 'output' of a functor to model multiple sorts of superstructure, in a single definition. Mutually defined datatypes, and multi-sorted syntaxes in particular, can be seen this way. My particular favourite example is the definition of locally nameless λ-terms in β-normal form, with de Bruijn indices for bound variables (Var), but no commitment to a particular representation of free variables (Par).

$$\begin{array}{l}\textbf{data}\ \mathsf{Norm} :: * \to *\ \textbf{where}\\ \quad \mathsf{Lam} :: \mathsf{Norm}\ x \to \mathsf{Norm}\ x\\ \quad \mathsf{Neu} :: \mathsf{Neut}\ x \to\ \mathsf{Norm}\ x\\ \textbf{data}\ \mathsf{Neut} :: * \to *\ \textbf{where}\\ \quad \mathsf{Par}\ \ :: x \to \qquad\qquad\ \ \mathsf{Neut}\ x\\ \quad \mathsf{Var}\ \ :: \mathsf{Int} \to \qquad\qquad\ \ \mathsf{Neut}\ x\\ \quad \mathsf{App} :: \mathsf{Neut}\ x \to \mathsf{Norm}\ x \to \mathsf{Neut}\ x\end{array}$$

We have two type constructors of kind $* \to *$, but we can use a little algebra to express the pair as a single exponential.

$$(* \to *, * \to *)\ \cong\ * \to (*, *)\ \cong\ * \to (\{\mathsf{Bool}\} \to *)$$

This suggests that we might combine the two branches of our definition into one, sharing the 'variables' parameter and using a $\{\mathsf{Bool}\}$ index to distinguish Norm ($\{\mathsf{T}\}$, say) from Neut.

$$\begin{array}{l}\textbf{data}\ \mathsf{NorN} :: * \to (\{\mathsf{Bool}\} \to *)\ \textbf{where}\\ \quad \mathsf{Lam} :: \mathsf{NorN}\ x\ \{\mathsf{T}\} \to \qquad\qquad \mathsf{NorN}\ x\ \{\mathsf{T}\}\\ \quad \mathsf{Neu} :: \mathsf{NorN}\ x\ \{\mathsf{F}\} \to \qquad\qquad \mathsf{NorN}\ x\ \{\mathsf{T}\}\\ \quad \mathsf{Par}\ \ :: x \to \qquad\qquad\qquad\ \ \mathsf{NorN}\ x\ \{\mathsf{F}\}\end{array}$$

[2] The P in PFix is silent.

[3] This is the 'categorical' just, which acts like cyanide in lepidoptery.

$$\begin{array}{l}\mathsf{Var}\ :: \mathsf{Int} \to \qquad\qquad\qquad\qquad\qquad \mathsf{NorN}\ x\ \{\mathsf{F}\} \\ \mathsf{App} :: \mathsf{NorN}\ x\ \{\mathsf{F}\} \to \mathsf{NorN}\ x\ \{\mathsf{T}\} \to \mathsf{NorN}\ x\ \{\mathsf{F}\}\end{array}$$

We can see NorN as a functor from $*$ to $\{\mathsf{Bool}\} \to *$ whose action is simultaneous *renaming*, lifting functions on free variables to functions on terms which respect syntactic structure and preserving their status as normal or neutral. Bound variables are, of course, unaffected.

$$\begin{array}{l}\mathsf{rename} :: (\alpha \to \beta) \to (\mathsf{NorN}\ \alpha \twoheadrightarrow \mathsf{NorN}\ \beta) \\ \mathsf{rename}\ r\ (\mathsf{Lam}\ t) \quad = \mathsf{Lam}\ (\mathsf{rename}\ r\ t) \\ \mathsf{rename}\ r\ (\mathsf{Neu}\ t) \quad = \mathsf{Neu}\ (\mathsf{rename}\ r\ t) \\ \mathsf{rename}\ r\ (\mathsf{Par}\ x) \quad = \mathsf{Par}\ (r\ x) \\ \mathsf{rename}\ r\ (\mathsf{Var}\ i) \quad = \mathsf{Var}\ i \\ \mathsf{rename}\ r\ (\mathsf{App}\ f\ s) = \mathsf{App}\ (\mathsf{rename}\ r\ f)\ (\mathsf{rename}\ r\ s)\end{array}$$

Of course, the above representation fails to enforce the safe scoping of bound variables. The traditional remedy is also an instance of indexing. Given a type Nat of natural numbers with constructors Z and S, we may imagine defining finite sets of bounded numbers according to venerable tradition, in which $\mathsf{Fin}\ \{\mathsf{Z}\}$ is empty and $\mathsf{Fin}\ \{\mathsf{S}\ n\}$ has one element more than $\mathsf{Fin}\ \{n\}$.

$$\begin{array}{l}\mathbf{data}\ \mathsf{Fin} :: \{\mathsf{Nat}\} \to *\ \mathbf{where} \\ \quad \mathsf{FZ} :: \qquad\quad \mathsf{Fin}\ \{\mathsf{S}\ n\} \\ \quad \mathsf{FS} :: \mathsf{Fin}\ \{n\} \to \mathsf{Fin}\ \{\mathsf{S}\ n\}\end{array}$$

Now we can make our definition guarantee scope:

$$\begin{array}{l}\mathbf{data}\ \mathsf{NN} :: * \to (\{\mathsf{Nat}, \mathsf{Bool}\} \to *)\ \mathbf{where} \\ \quad \mathsf{Lam} :: \mathsf{NN}\ x\ \{\mathsf{S}\ n, \mathsf{T}\} \to \qquad\qquad \mathsf{NN}\ x\ \{n, \mathsf{T}\} \\ \quad \mathsf{Neu} :: \mathsf{NN}\ x\ \{n, \mathsf{F}\} \to \qquad\qquad \mathsf{NN}\ x\ \{n, \mathsf{T}\} \\ \quad \mathsf{Par}\ :: x \to \qquad\qquad\qquad\quad \mathsf{NN}\ x\ \{n, \mathsf{F}\} \\ \quad \mathsf{Var}\ :: \mathsf{Fin}\ \{n\} \to \qquad\qquad\quad \mathsf{NN}\ x\ \{n, \mathsf{F}\} \\ \quad \mathsf{App} :: \mathsf{NN}\ x\ \{n, \mathsf{F}\} \to \mathsf{NN}\ x\ \{n, \mathsf{T}\} \to \mathsf{NN}\ x\ \{n, \mathsf{F}\}\end{array}$$

We have stepped beyond the realm of mutually defined simple types — NN cannot be 'demutualized' into finitely many unindexed definitions. This should give us reason to hoipe that a principled approach to indexed data structures might deliver more than just a model of simple mutual definitions, as found in [**?**]. But so far, I have been rather casual in adding type-like kinds to this paper-local dialect of Haskell. It is high time I was at least precise in my wishful thinking.

6. The Braces of Upward Mobility

The Glasgow Haskell Compiler currently extends the Haskell 98 standard significantly, supporting in particular the declaration of Generalized Algebraic DataTypes (GADTs), a form of what dependent type theorists still call 'inductive families' when they are talking amongst themselves. However, in order to maintain the separation of terms from types, GHC requires that GADTs are indexed by type-level stuff and have kinds built from $*$ and $\to$.

Let me propose a little Strathclyde Haskell Enhancement. I extend the syntax of kinds as follows

$$\begin{array}{rl}\kappa = & * \\ \mid & \{\tau\} \\ \mid & \kappa \to \kappa \\ \mid & \forall\ \alpha :: \kappa.\ \kappa\end{array}$$

where τ denotes the syntactic category of types. The $\{\tau\}$ construct lifts a type τ to the kind level, allowing it to be used as an index. The kind $\{\mathsf{Nat}, \mathsf{Bool}\}$ I wrote above is just a little sugar for the lifted pair type $\{(\mathsf{Nat}, \mathsf{Bool})\}$. However, with this increased diversity at the kind level comes the need for polymorphism. Correspondingly, let us permit quantification over inhabitants of kinds. Note that the latter falls far short of *kind polymorphism*, abstracting kinds themselves. The variables α bound by the quantifier live at the type level, and can thus only be used inside $\{\ldots\}$.

Lifted types are inhabited by lifted expressions, but before anyone starts panicking about dependent types in Haskell, or boasting about not needing them, let me be quite clear that this is a much more modest proposal. In order to retain essentially the same type inference technology we use today, let us be sure that the expressions we lift

1. are guaranteed to terminate;
2. admit first-order unification.

One fragment of the expressions which clearly satisfies these criteria consists of just those which can be built from variables and fully applied value constructors. Let us extend the type language

$$\begin{array}{rl}\tau = & \ldots \\ \mid & \{\upsilon\} \\ \upsilon = & x \\ \mid & \mathsf{C}\ \upsilon^*\end{array}$$

These υ-forms are the very stuff of first-order unification, and they do not compute at all, let alone for ever. Greater ambition is to be encouraged in the long term, but I shall choose to be patient in this paper. Let me also assure you that the variables permitted in υ-forms are *not* the program variables in scope, but rather the $\forall$-bound type-level variables which happen to inhabit $\{\tau\}$ kinds. Correspondingly, we retain the separation of run-time and type-level values for the time being, even if the re-use of notation suggests otherwise, and nods towards stronger possibilities in future.

With this extension in place, it becomes harder to impress your friends by faking type-level copies of value-level data. We can just use plain old data and get along. So let us not be distracted by coding tricks any longer, and get to work.

7. Slice Categories, Indexed Functors

I can only learn category theory by accident, discovering that some abstruse phrase of the categorical taxonomy connects to some computational intuition with which I am already at home: the dual of the categorical just is the programmer's 'is *that* what they're on about?'. In that spirit, let me try to equip the working Haskell programmer with some helpful categorical kit in tandem with its computational payoff.

For a given category $\mathbb{C}$ — think of $*$ in the first instance — let us write

$$\alpha :: \mathbb{C} \qquad f :: \sigma \to \tau$$

to indicate that α is an *object* of $\mathbb{C}$ and that f is an *arrow* from object σ to object τ. For $\mathbb{C}$ to be a category, we require the existence of identities and composites

$$\frac{\alpha :: \mathbb{C}}{\mathsf{id} :: \alpha \to \alpha} \qquad \frac{f :: \sigma \to \tau \quad g :: \rho \to \sigma}{f \circ g :: \rho \to \tau}$$

satisfying the usual trio of laws:

$$\mathsf{id} \circ g = g \qquad f \circ \mathsf{id} = f \qquad (f \circ g) \circ h = f \circ (g \circ h)$$

Now pick an 'index' object $\iota :: \mathbb{C}$. The *slice category* $\mathbb{C}\ /\ \iota$ has 'objects with indexing', and 'index-preserving arrows'. That is, each object of $\mathbb{C}\ /\ \iota$ is an object of $\mathbb{C}$ equipped with a $\mathbb{C}$-arrow to ι, and the arrows between two such indexed objects are exactly the arrows on the underlying objects which happen to preserve indexing.

$$\frac{\alpha :: \mathbb{C} \quad a :: \alpha \to \iota}{(\alpha, a) :: \mathbb{C}\ /\ \iota} \qquad \frac{f :: \sigma \to \tau \quad t \circ f = s}{f :: (\sigma, s) \to (\tau, t)}$$

It is not hard to check that $\mathsf{id} :: (\alpha, a) \to (\alpha, a)$, and that $f \circ g$ preserves indexing if both f and g do.

In a language with dependent pairs and equality types, it is straightforward to code up slices for the category of types and functions: an object is the dependent pair of a type α and an indexing function from α; an arrow is the dependent pair of a function f and the proof that f preserves index.

Our category of indexed type constructors $\sigma, \tau :: \{\iota\} \to *$ and index-preserving functions $f :: \sigma \dashrightarrow \tau$ performs the role of $*/\iota$ without the need

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[Abbott et al.(2005)Abbott, Altenkirch, and Ghani] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[Aitken and Reppy(1992)] William Aitken and John Reppy. Abstract value constructors. Technical Report TR 92-1290, Cornell University, 1992.