Hasochism

The Pleasure and Pain of Dependently Typed Haskell Programming

Sam Lindley
University of Strathclyde
Sam.Lindley@ed.ac.uk

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

Abstract

Haskell's type system has outgrown its Hindley-Milner roots to the extent that it now stretches to the basics of dependently typed programming. In this paper, we collate and classify techniques for programming with dependent types in Haskell, and contribute some new ones. In particular, through extended examples—merge-sort and rectangular tilings—we show how to exploit Haskell's constraint solver as a theorem prover, delivering code which, as Agda programmers, we envy. We explore the compromises involved in simulating variations on the theme of the dependent function space in an attempt to help programmers put dependent types to work, and to inform the evolving language design both of Haskell and of dependently typed languages more broadly.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]

Keywords dependent types; singletons; data type promotion; proof search; invariants

1. Introduction

In the design of Standard ML, Milner and his colleagues achieved a remarkable alignment of distinctions: [16, 17]

syntactic category	**terms**	**types**
phase distinction	**dynamic**	**static**
inference	**explicit**	**implicit**
abstraction	**simple**	**dependent**

The things you write are the things you run, namely terms, for which abstraction (with an explicit λ) is simply typed—the bound variable does not occur in the return type of the function. The things which you leave to be inferred, namely polymorphic type schemes, exist only at compile time and allow (outermost) dependent abstraction over types, with implicit application at usage sites instantiating the bound variables.

An unintended consequence of Milner's achievement is that we sometimes blur the distinctions between these distinctions. We find it hard to push them out of alignment because we lose sight of the very possibility of doing so. For example, some have voiced objections to the prospect of terms in types on the grounds that efficient compilation relies on erasure to the dynamic fragment of the language. However, renegotiating the term/type distinction need not destroy the dynamic/static distinction, as shown by Coq's venerable program extraction algorithm [21], erasing types and proofs from dependently typed constructions.

Meanwhile, Haskell's type classes [25] demonstrate the value of dynamic components which are none the less implicit—instance dictionaries. Indeed, type inference seems a timid virtue once you glimpse the prospect of *program* inference, yet some are made nervous by the prospect of unwritten programs being run. Similarly, Haskell's combination of higher kinds and constraints means that sometimes static types must be given explicitly, in order not only to check them, but also to drive the generation of invisible boilerplate.

Milner's aligned distinctions have shifted apart, but Haskell persists with one dependent quantifier for implicit abstraction over static types. What counts as a 'type' has begun to stretch. Our *Strathclyde Haskell Enhancement* (SHE) preprocessor [13] systematized and sugared common constructions for building the type level analogues of run time data, together with run time witnesses to type level values. SHE then provided something which resembles a dependent quantifier for explicit abstraction over dynamic terms—the Π-type of dependent type theory—in domains simple enough to admit the singleton construction. Before long, Glasgow Haskell headquarters responded with a proper kind system for 'promoted' data types [27], making possible the singletons library [7]. The arrival of data types at the kind level necessitated polymorphism in kinds: Haskell is now a dependently *kinded* language, and although it is a nuisance that the kind-level $\forall$ is compulsorily implicit, the fresh abstractions it offers have yielded considerable simplification, e.g., in support of generic programming [10].

So we decided to have some fun, motivated by the reliability benefits of programming at a higher level of static precision, and the experience of doing so in prototype dependently typed languages—in our case, Epigram [14] and Agda [20]. There is a real sense of comfort which comes from achieving a high level of hygiene, and it is something which we want to bring with us into practical programming in industrial strength languages like Haskell. Of course, reality intervenes at this point: some desirable methods are harder to express than one might hope, but we can also report some pleasant surprises. We hope our experiments inform both programming practice with current tools and the direction of travel for Haskell's evolution.

Specifically, this paper contributes

- an analysis of how to achieve dependent quantification in Haskell, framed by the distinctions drawn above—we note that Agda and Haskell both have room for improvement;

Haskell '13, September 23–24, Boston, MA, USA.

ACM 978-1-4503-2383-3/13/09…$15.00.
http://dx.doi.org/10.1145/2503778.2503786

- practical techniques for dependently typed programming in Haskell, with a view to minimizing explicit proof in program texts;
- an implementation of merge-sort guaranteeing the ordering invariant for its output, in which the proofs are *silent*;
- an algebra for tiling size-indexed boxes, fitting with precision, leading to an implementation of a screen editor.

Overview Section 2 explores variations on the theme of dependent quantification, through paradigmatic examples involving natural numbers and vectors. Section 3 focuses on the implicit/explicit distinction, whilst developing standard library functionality for vectors, identifying areas of concern. Section 4 delivers merge-sort, using instance inference for proof search. Section 5 explores the use of data types to represent effective evidence. Section 6 introduces an algebra of size-indexed boxes, which is used to build a text editor in Section 7. Section 8 concludes.

Online code All of the Haskell source code for the developments in this paper are available at `https://github.com/slindley/dependent-haskell/tree/master/Hasochism`.

Acknowledgements This work was supported by EPSRC project *Haskell Types with Added Value*, grant EP/J014591/1. We are grateful to be part of the long running conversation about Haskell's type system, and in particular to Simon Peyton Jones, Stephanie Weirich, Richard Eisenberg, Iavor Diatchki, Dimitrios Vytiniotis, José Pedro Magalhães and our colleague Adam Gundry.

2. A Variety of Quantifiers

Haskell's DataKinds extension [27] has the impact of duplicating an ordinary data type, such as

```
data Nat = Z | S Nat deriving (Show, Eq, Ord)
```

at the *kind* level. That is, for the price of the above *type* declaration, GHC silently generates a new *kind*, also Nat, with inhabitants formed by type level data constructors 'Z and 'S, where the prefixed quote may be dropped for names which do not clash with declared types. It is pleasant to think that the *same* Nat is both a type and a kind, but sadly, the current conceptual separation of types and kinds requires the construction of a separate kind-level replica.

The Nat kind is now available as a domain for various forms of universal quantification, classified on the one hand by whether the quantified values are available only statically or also dynamically, and on the other hand by whether the associated abstraction and application are implicit or explicit in the program text. Picking apart Milner's alignment of distinctions, we acquire a matrix of four dependent quantifiers for term-like things. In this section and the next, we explore the Haskell encodings and the typical usage of these quantifiers, tabulated here for the paradigmatic example of natural numbers:

	implicit	**explicit**
static	$\forall(n :: \mathsf{Nat}).$	$\forall(n :: \mathsf{Nat}).\mathsf{Proxy}\ n \rightarrow$
dynamic	$\forall n.\mathsf{NATTY}\ n \Rightarrow$	$\forall n.\mathsf{Natty}\ n \rightarrow$

To get to work, we must find types which involve numbers. Generalized algebraic data types, now bearing an even stronger resemblance to the inductive families of dependent type theories, provide one source. The family of *vectors* is the traditional first example of such a creature, and we shall resist the contrarian urge to choose another because we shall need vectors later in the paper.

```
data Vec :: Nat → ⋆ → ⋆ where
  V0   ::                   Vec Z x
  (:>) :: x → Vec n x → Vec (S n) x
```

In Haskell, one must choose a type's order of arguments with care, as partial application is permitted but λ-abstraction is not. Here we depart a little from the dependently typed tradition by giving Vec its length *index* to the left of its payload type *parameter*, x, because we plan to develop the functorial structure of each Vec n in the next section.

We note that the correspondence with the inductive families of Agda, Coq and Idris is not exact. The n in the Haskell type of (:>) is given a *static* implicit quantifier and erased at run time, whereas its type theoretic counterpart is *dynamic* and implicit. Idris, at least, is clever enough to erase the run time copy of n, through Brady's 'forcing' optimization [5].

Meanwhile, type level data are useful for more than just indexing data types. We may indeed compute with them, making use of Haskell's 'type family' extension, which allows us to define 'families' (meaning merely 'functions') of 'types' in the sloppy sense of 'things at the type level', not just the pedantic sense of 'things of kind $\star$'.

```
type family (m :: Nat) :+ (n :: Nat) :: Nat
type instance Z    :+ n = n
type instance S m :+ n = S (m :+ n)
```

In an intensional dependent type theory, such a definition extends the normalization algorithm by which the type checker decides type equality up to the partial evaluation of open terms. If syntactically distinct types share a normal form, then they share the same terms. For example, in type theory, terms inhabiting Vec (S (S Z) :+ n) x also inhabit Vec (S (S n)) x without further ado. Of course, functions often satisfy laws, e.g. associativity and commutativity, which are not directly computational: terms of type Vec (n :+ S (S Z)) x do not inhabit Vec (S (S n)) x, even though the two coincide for all concrete values of n. Fortunately, one can formulate 'propositional equality' types, whose inhabitants constitute evidence for equations. Values can be transported between provably equal types by explicit appeal to such evidence.

In Haskell's kernel, type equality is entirely syntactic [24], so that kernel terms in Vec (S (S Z) :+ n) x do not also inhabit Vec (S (S n)) x. The above 'definition' *axiomatizes* (:+) for Haskell's propositional equality, and every program which relies on computing sums must be elaborated with explicit appeal to evidence derived from those axioms. The translation from the surface language to the kernel attempts to generate this evidence by a powerful but inscrutable constraint solving heuristic. Experience suggests that the solver computes aggressively, regardless of whether type level programs are totally recursive, so we may confidently type vector concatenation in terms of addition.

```
vappend :: Vec m x → Vec n x → Vec (m :+ n) x
vappend V0        ys = ys
vappend (x :> xs) ys = x :> vappend xs ys
```

Note that the numbers here play an entirely static role: the flow of control can be determined entirely from the constructors of the first vector. Suppose, however, that we wish to invert concatenation, chopping a vector in two.

```
vchop :: Vec (m :+ n) x → (Vec m x, Vec n x)
```

Unlike with vappend, we shall certainly need m at run time, and we shall need to refer to it explicitly in order to judge where to chop. However, Haskell's dependent $\forall$·quantifier is for implicit and exclusively static things. The standard solution is to define the run time replica of some static data as a *singleton* GADT.

```
data Natty :: Nat → ⋆ where
  Zy :: Natty Z
  Sy :: Natty n → Natty (S n)
```

Each type level value n in the Nat kind has a unique representative in the type Natty n, so analysing the latter will reveal useful facts about the former. The 'Π-types', often written $(x : S) \rightarrow T$, of dependent type theory abstract dependently over explicit dynamic things. In Haskell, we can simulate this behaviour by abstracting dependently at the type level and non-dependently over the singleton representative. We translate (from Agda notation to Haskell):

$$(n{:}\mathsf{Nat}) \to T \qquad \rightsquigarrow \qquad \forall n :: \mathsf{Nat}.\mathsf{Natty}\ n \to T$$

Thus equipped, we may write

```
vchop :: Natty m → Vec (m :+ n) x → (Vec m x, Vec n x)
vchop Zy        xs        = (V0,       xs)
vchop (Sy m) (x :> xs) = (x :> ys, zs)
  where (ys, zs) = vchop m xs
```

There may be an argument from implementation inertia in favour of this means of dependent quantification, but it proliferates representations of cognate notions, which is an eccentric way to keep things simple.

Moreover, we can only construct Π-types with domains admitting the singleton construction. Whilst Monnier and Haguenauer [19] have given a generic treatment of the singleton construction, their result is not reproducible in current GHC because GADTs are not promotable as kinds. We cannot form a Haskell analogue of

$$(n{:}\mathsf{Nat}) \to (xs{:}\mathsf{Vec}\ n\ x) \to T[xs]$$

but we expect this gap to be plugged in the near future. Promoting Vec n x to a kind perforce involves using numbers not only in terms and types, but in kinds as well. In the new, more flexible world, the type/kind distinction is increasingly inconvenient, and a clear candidate for abolition, as Weirich, Hsu, and Eisenberg propose [26].

Meanwhile, a further disturbance is in store if we choose to compute only the first component returned by vchop. Cutting out the suffix gives us

```
vtake :: Natty m → Vec (m :+ n) x → Vec m x   -- (×)
vtake Zy        xs        = V0
vtake (Sy m) (x :> xs) = x :> vtake m xs
```

but the resulting type error

```
NatVec.lhs:120:44:
    Could not deduce (n2 ~ (n1 :+ n0))
    from the context (m ~ 'S n1)
      bound by a pattern with constructor
          Sy :: forall (n :: Nat). Natty n -> Natty ('S n),
        in an equation for ‘vtake’
      at NatVec.lhs:120:10-13
    or from ((m :+ n) ~ 'S n2)
      bound by a pattern with constructor
          :> :: forall x (n :: Nat).
                    x -> Vec n x -> Vec ('S n) x,
        in an equation for ‘vtake’
      at NatVec.lhs:120:18-24
      ‘n2’ is a rigid type variable bound by
           a pattern with constructor
             :> :: forall x (n :: Nat).
                       x -> Vec n x -> Vec ('S n) x,
           in an equation for ‘vtake’
           at NatVec.lhs:120:18
    Expected type: Vec (n1 :+ n0) x
      Actual type: Vec n2 x
    In the second argument of ‘vtake’, namely ‘xs’
    In the second argument of ‘(:>)’, namely ‘vtake m xs’
    In the expression: x :> vtake m xs
```

amounts to the fact that it is not clear how to instantiate n in the recursive call. It takes sophisticated reasoning about addition to realise that $(m{:}+)$ is injective. To GHC, it is just an unknown axiomatised function. The problem did not arise for vchop, because relaying the suffix, zs, from the recursive output to the result makes clear that the same n is needed in both places. This n is not needed at run time, but without it there is no way to see that the program makes sense.

The upshot is that there are data which, despite being static, must be made explicit. One way to manifest them is via ‘proxy types’, e.g.,

```
data Proxy :: κ → ⋆ where
  Proxy :: Proxy i
```

As you can see, the only dynamic information in Proxy i is definedness, which there is never the need to check. Kind polymorphism allows us to declare the proxy type once and for all. The only point of a proxy is to point out that it has the same type at its binding and its usage sites. Although it is compulsory to instantiate quantifiers by inference, proxies let us rig the guessing game so that GHC can win it. We repair the definition of vtake thus:

```
vtake :: Natty m → Proxy n → Vec (m :+ n) x → Vec m x
vtake Zy        n xs          = V0
vtake (Sy m) n (x :> xs) = x :> vtake m n xs
```

Of course, when calling vtake, we need to get a proxy from somewhere. If we do not already have one, we can write (Proxy :: Proxy t) for the relevant type level expression t. The ScopedTypeVariables extension allows us to write open types. If we already have some other value with the same index, e.g. a singleton value, we can erase it to a proxy with

```
proxy :: f i → Proxy i
proxy _ = Proxy
```

The vtake example shows that Haskell’s ∀·quantifier supports abstraction over data which play a relevant and computational role in static types but have no impact on run time execution and are thus erasable. Most dependently typed languages, with ATS [6] being a notable exception, do not offer such a quantifier, which seems to us something of an oversight. Coq’s program extraction [21] and Brady’s compilation method [4] both erase components whose types show that they cannot be needed in computation, but they do not allow us to make the promise that ordinary data in types like Nat will not be needed at run time.

Meanwhile, Agda has an ‘irrelevant’ quantifier [1], abstracting over data which will even be ignored by the definitional equality of the type system. In effect, the erasure induced by ‘irrelevance’ is static as well as dynamic, and is thus more powerful but less applicable. The Agda translation of vtake cannot make n an irrelevant argument, because it is needed to compute the length of the input, which most certainly is statically relevant. In contemporary Agda, it seems that this n must be present at run time.

A further example, showing implicit quantification over data used statically to compute a type but erased at run time, applies an n-ary operator to an n-vector of arguments.

```
type family Arity (n :: Nat) (x :: ⋆) :: ⋆
type instance Arity Z      x = x
type instance Arity (S n) x = x → Arity n x

varity :: Arity n x → Vec n x → x
varity x V0          = x
varity f (x :> xs) = varity (f x) xs
```

Here, pattern matching on the vector delivers sufficient information about its length to unfold the Arity computation. Once again, Agda would allow n to remain implicit in source code, but insist on retaining n at run time. Meanwhile, Brady’s ‘detagging’ optimization [5] would retain n but remove the constructor tag from the representation of vectors, compiling the above match on the vector to match instead on n then project from the vector.

Miquel’s implicit calculus of constructions (ICC) [18] extends type theory with a static implicit quantifier, the “implicit product”, which erases like a System F ∀·quantifier. Barras and Bernado’s ICC* [3] adds a static explicit quantifier to restore decidable type checking. Adding something like the static explicit quantifier (and a

Pollack-style implicit synthesis mechanism) to Agda would restore to Agda the missing static half of our quantifier matrix.

To sum up, we have distinguished Haskell's dependent static implicit $\forall\cdot$quantifier from the dependent dynamic explicit Π-types of dependent type theory. We have seen how to make $\forall\cdot$ static and explicit with a Proxy, and how to make it dynamic and explicit whenever the singleton construction is possible. However, we have noted that whilst Haskell struggles to simulate Π with $\forall\cdot$, the reverse is the case in type theory. What is needed, on both sides, is a more systematic treatment of the varieties of quantification.

3. Explicit and Implicit Π-Types

We have already seen that singletons like Natty simulate a dependent dynamic explicit quantifier, corresponding to the explicit Π-type of type theory: Agda's $(x : S) \to T$. Implementations of type theory, following Pollack's lead [22], often support a dependent dynamic *implicit* quantifier, the $\{x : S\} \to T$ of Agda, allowing type constraints to induce the synthesis of useful information. The method is Milner's—substitution arising from unification problems generated by the typechecker—but the direction of inference runs from types to programs, rather than the other way around.

The Haskell analogue of the implicit Π is constructed with singleton *classes*. For example, the following NATTY type class defines a single method natty, delivering the Natty singleton corresponding to each promoted Nat. A NATTY number is known at run time, despite not being given explicitly.

```
class NATTY (n :: Nat) where
  natty :: Natty n
instance NATTY Z where
  natty = Zy
instance NATTY n ⇒ NATTY (S n) where
  natty = Sy natty
```

For example, we may write a more implicit version of vtake:

```
vtrunc :: NATTY m ⇒ Proxy n → Vec (m :+ n) x → Vec m x
vtrunc = vtake natty
```

The return type determines the required length, so we can leave the business of singleton construction to instance inference.

```
> vtrunc Proxy (1 :> 2 :> 3 :> 4 :> V0) :: Vec (S (S Z)) Int
1 :> 2 :> V0
```

3.1 Instances for Indexed Types

It is convenient to omit singleton arguments when the machine can figure them out, but we are entitled to ask whether the additional cost of defining singleton classes as well as singleton types is worth the benefit. However, there is a situation where we have no choice but to work implicitly: we cannot abstract an **instance** over a singleton type, but we can constrain it. For example, the Applicative instance [15] for vectors requires a NATTY constraint.

```
instance NATTY n ⇒ Applicative (Vec n) where
  pure   = vcopies natty
  (<⋆>) = vapp
```

where vcopies needs to inspect a run time length to make the right number of copies—we are obliged to define a helper function:

```
vcopies :: ∀n x.Natty n → x → Vec n x
vcopies Zy      x = V0
vcopies (Sy n) x = x :> vcopies n x
```

Meanwhile, vapp is pointwise application, requiring only static knowledge of the length.

```
vapp :: ∀n s t.Vec n (s → t) → Vec n s → Vec n t
vapp V0        V0        = V0
vapp (f :> fs) (s :> ss) = f s :> vapp fs ss
```

We note that simply defining (<⋆>) by pattern matching in place

```
instance NATTY n ⇒ Applicative (Vec n) where   -- (×)
  pure = vcopies natty
  V0        <⋆> V0        = V0
  (f :> fs) <⋆> (s :> ss) = f s :> (fs <⋆> ss)
```

yields an error in the step case, where $n \sim$ S m but NATTY m cannot be deduced. *We* know that the NATTY n instance must be a NATTY (S m) instance which can arise only via an instance declaration which presupposes NATTY m. However, such an argument via 'inversion' does not explain how to construct the method dictionary for NATTY m from that of NATTY (S m). When we work with Natty explicitly, the corresponding inversion is just what we get from pattern matching. The irony here is that (<⋆>) does not need the singleton at all!

Although we are obliged to define the helper functions, vcopies and vapp, we could keep them local to their usage sites inside the instance declaration. We choose instead to expose them: it can be convenient to call vcopies rather than pure when a Natty n value is to hand but a NATTY n dictionary is not; vapp needs neither.

To finish the Applicative instance, we must ensure that Vec n is a Functor. In fact, vectors are Traversable, hence also Foldable Functors in the default way, without need for a NATTY constraint.

```
instance Traversable (Vec n) where
  traverse f V0 = pure V0
  traverse f (x :> xs) = (:>) <$> f x <⋆> traverse f xs
instance Foldable (Vec n) where
  foldMap = foldMapDefault
instance Functor (Vec n) where
  fmap = fmapDefault
```

3.2 Matrices and a Monad

It is quite handy that Vec n is both Applicative and Traversable. If we define a Matrix as a vertical vector of height h containing horizontal vectors of width w, thus (arranging Matrix's arguments conveniently for the tiling library later in the paper),

```
data Matrix :: ⋆ → (Nat, Nat) → ⋆ where
  Mat :: {unMat :: Vec h (Vec w a)} → Matrix a '(w, h)
```

we get transposition cheaply, provided we know the width.

```
transpose :: NATTY w ⇒ Matrix a '(w, h) → Matrix a '(h, w)
transpose = Mat ◦ sequenceA ◦ unMat
```

The width information really is used at run time, and is otherwise unobtainable in the degenerate case when the height is Z: transpose must know how many V0s to deliver.

Completists may also be interested to define the Monad instance for vectors whose join is given by the diagonal of a matrix. This fits the Applicative instance, whose (<⋆>) method more directly captures the notion of 'corresponding positions'.

```
vtail :: Vec (S n) x → Vec n x
vtail (_ :> xs) = xs
diag :: Matrix x '(n, n) → Vec n x
diag (Mat V0)                = V0
diag (Mat ((x :> _) :> xss)) = x :> diag (Mat (fmap vtail xss))
instance NATTY n ⇒ Monad (Vec n) where
  return = pure
  xs ≫= f = diag (Mat (fmap f xs))
```

Gibbons (in communication with McBride and Paterson [15]) notes that the diag construction for unsized lists does not yield a monad, because the associativity law fails in the case of 'ragged' lists of lists. By using sized vectors, we square away the problem cases.

3.3 Exchanging Explicit and Implicit

Some interplay between the explicit and implicit Π-types is inevitable. Pollack wisely anticipated situations where argument synthesis fails because the constraints are too difficult or too few, and provides a way to override the default implicit behaviour manually. In Agda, if $f : \{x : S\} \to T$, then one may write $f\ \{s\}$ to give the argument.

The Hindley-Milner type system faces the same issue: even though unification is more tractable, we still encounter terms like const True ⊥ :: Bool where we do not know which type to give ⊥—parametric polymorphism ensures that we don't need to know. As soon as we lose parametricity, e.g. in show ∘ read, the ambiguity of the underconstrained type is a problem and rightly yields a type error. The 'manual override' takes the form of a type annotation, which may need to refer to type variables in scope.

As we have already seen, the natty method allows us to extract an explicit singleton whenever we have implicit run time knowledge of a value. Occasionally, however, we must work the other way around. Suppose we have an explicit Natty n to hand, but would like to use it in a context with an implicit NATTY n type class constraint. We can cajole GHC into building us a NATTY n dictionary as follows:

```
natter :: Natty n → (NATTY n ⇒ t) → t
natter Zy      t = t
natter (Sy n) t = natter n t
```

This may look like an obfuscated identity function, but its type tells us otherwise. The t being passed along recursively is successively but silently precomposed with the dictionary transformer generated from the **instance** NATTY n ⇒ NATTY (S n) declaration. Particularly galling, however, is the fact that the dictionary thus constructed contains just an exact replica of the Natty n value which natter has traversed.

We have completed our matrix of dependent quantifiers involving the kind Nat and two ways (neither of which is the type Nat) to give its inhabitants run time representation, NATTY and Natty, which are only clumsily interchangeable despite the former wrapping the latter. We could (and indeed SHE does) provide a more pleasing notation to make the dynamic quantifiers look like Π-types and their explicit instantiators look like ordinary data, but the awkwardness is more than skin deep.

3.4 The NATTY-in-Natty Question

Recall that we defined the singleton representation of natural numbers as follows.

```
data Natty :: Nat → ⋆ where
  Zy :: Natty Z
  Sy :: Natty n → Natty (S n)
```

Another possible design choice is to insert a NATTY constraint in the successor case, effectively storing two copies of the predecessor. This is the choice taken by Eisenberg and Weirich in the Singletons library [7].

```
data Natty :: Nat → ⋆ where
  Zy :: Natty Z
  Sy :: NATTY n ⇒ Natty n → Natty (S n)
```

Each choice has advantages and disadvantages. The unconstrained version clearly makes for easier construction of singletons, whilst the constrained version makes for more powerful elimination.

Without the NATTY constraint on Sy, we can write a function to compute the length of a vector as follows:

```
vlength :: Vec n x → Natty n
vlength V0          = Zy
vlength (x :> xs) = Sy (vlength xs)
```

However, with the NATTY constraint on Sy, the construction becomes more complex, and we must write:

```
vlength :: Vec n x → Natty n
vlength V0          = Zy
vlength (x :> xs) = natter n (Sy n)  where n = vlength xs
```

in order to bring the appropriate NATTY constraint into scope for the inductive case.

Let us write a function to construct an identity matrix of size n. Here, we are eliminating a singleton. Without the NATTY constraint on Sy, we must use natter to enable the use of the relevant Applicative structure.

```
idMatrix :: Natty n → Matrix Int '(n, n)
idMatrix (Sy n) = natter n $
  Mat ((1 :> pure 0) :> ((0:>) <$> unMat (idMatrix n)))
idMatrix Zy       = Mat V0
```

However, with the NATTY constraint on Sy, we can omit natter, because the required constraint is brought into scope by pattern matching.

```
idMatrix :: Natty n → Matrix Int '(n, n)
idMatrix (Sy n) =
  Mat ((1 :> pure 0) :> ((0:>) <$> unMat (idMatrix n)))
idMatrix Zy       = Mat V0
```

For constructions like vlength it is most convenient to omit the NATTY constraint from the successor constructor. For eliminations like idMatrix, it is most convenient to attach the NATTY constraint to the successor constructor. It is hard to predict which polarity is more likely to dominate, but the issue with elimination happens only when we have the explicit witness but need the implicit one.

There is also a time/space trade-off, as including the constraint effectively requires storing the same information twice at each node, but allows for an implementation of natter by one step of case analysis, rather than a full recursion.

```
natter :: Natty n → (NATTY n ⇒ t) → t
natter Zy      t = t
natter (Sy n) t = t
```

SHE has vacillated between the two: the first implementation did not add the constraint; a tricky example provoked us to add it, but it broke too much code, so we reverted the change. Our experience suggests that omitting the constraint is more convenient more of the time. We should, however, prefer to omit the entire construction.

4. An Ordered Silence

We turn now to a slightly larger example—a development of merge-sort which guarantees by type alone to produce outputs in order. The significant thing about this construction is what is missing from it: explicit proofs. By coding the necessary logic using type classes, we harness instance inference as an implicit proof search mechanism and find it quite adequate to the task.

Let us start by defining ⩽ as a 'type' class, seen as a relation.

```
class LeN (m :: Nat) (n :: Nat)  where
instance                   LeN Z n  where
instance LeN m n ⇒ LeN (S m) (S n)  where
```

If we wanted to *close* this class, we could use the module abstraction method of Kiselyov and Shan [9] which uses a non-exported superclass. We leave this elaboration to the interested reader. The LeN class has no methods, but it might make sense to deliver at least the explicit evidence of ordering in the corresponding GADT, just as the NATTY class method delivers Natty evidence.

In order to sort numbers, we need to know that any two numbers can be ordered *one way or the other* (OWOTO). Let us say what it means for two numbers to be so orderable.

```
data OWOTO :: Nat → Nat → ⋆ where
  LE :: LeN x y ⇒ OWOTO x y
  GE :: LeN y x ⇒ OWOTO x y
```

Testing which way round the numbers are is quite a lot like the usual Boolean version, except with evidence. The step case requires unpacking and repacking because the constructors are used at different types (OWOTO m n versus OWOTO (S m) (S n)). However, instance inference is sufficient to deduce the logical goals from the information revealed by testing.

```
owoto :: ∀m n.Natty m → Natty n → OWOTO m n
owoto Zy       n       = LE
owoto (Sy m) Zy        = GE
owoto (Sy m) (Sy n) = case owoto m n of
  LE → LE
  GE → GE
```

Now we know how to put numbers in order, let us see how to make ordered lists. The plan is to describe what it is to be in order between *loose bounds* [11]. Of course, we do not want to exclude any elements from being sortable, so the type of bounds extends the element type with bottom and top elements.

```
data Bound x = Bot | Val x | Top deriving (Show, Eq, Ord)
```

We extend the notion of $\leqslant$ accordingly, so that instance inference can manage bound checking.

```
class LeB (a :: Bound Nat) (b :: Bound Nat)  where
instance              LeB Bot      b        where
instance LeN x y ⇒ LeB (Val x) (Val y)  where
instance              LeB (Val x) Top      where
instance              LeB Top      Top      where
```

And here are ordered lists of numbers: an OList l u is a sequence x_1 :< x_2 :< ... :< x_n :< ONil such that $l \leqslant x_1 \leqslant x_2 \leqslant \ldots \leqslant x_n \leqslant u$. The x:< checks that x is above the lower bound, then imposes x as the lower bound on the tail.

```
data OList :: Bound Nat → Bound Nat → ⋆ where
  ONil :: LeB l u ⇒ OList l u
  (:<) :: ∀l x u.LeB l (Val x) ⇒
          Natty x → OList (Val x) u → OList l u
```

We can write merge for ordered lists just the same way we would if they were ordinary. The key invariant is that if both lists share the same bounds, so does their merge.

```
merge :: OList l u → OList l u → OList l u
merge ONil        lu          = lu
merge lu          ONil        = lu
merge (x :< xu) (y :< yu) = case owoto x y of
  LE → x :< merge xu (y :< yu)
  GE → y :< merge (x :< xu) yu
```

The branches of the case analysis extend what is already known from the inputs with just enough ordering information to satisfy the requirements for the results. Instance inference acts as a basic theorem prover: fortunately (or rather, with a bit of practice) the proof obligations are easy enough.

Now that we can combine ordered lists of singleton numbers, we shall need to construct singletons for the numbers we intend to sort. We do so via a general data type for existential quantification.

```
data Ex (p :: κ → ⋆)  where
  Ex :: p i → Ex p
```

A 'wrapped Nat' is then a Natty singleton for any type-level number.

```
type WNat = Ex Natty
```

We can translate a Nat to its wrapped version by writing what is, morally, another obfuscated identity function between our two types of term level natural numbers.

```
wrapNat :: Nat → WNat
wrapNat Z       = Ex Zy
wrapNat (S m) = case wrapNat m of Ex n → Ex (Sy n)
```

You can see that wrapNat delivers the WNat corresponding to the Nat it receives, but that property is sadly not enforced by type—an inevitable consequence of separating Nat from its singletons. However, once we have WNats, we can build merge-sort in the usual divide-and-conquer way.

```
deal :: [x] → ([x], [x])
deal []         = ([], [])
deal (x : xs) = (x : zs, ys)  where (ys, zs) = deal xs

sort :: [Nat] → OList Bot Top
sort []   = ONil
sort [n] = case wrapNat n of Ex n → n :< ONil
sort xs   = merge (sort ys) (sort zs)  where (ys, zs) = deal xs
```

The need to work with WNat is a little clunky compared to what one might do in Agda where a single Nat type serves for Nat and its promotion, Natty, NATTY and WNat, but Agda does not have the proof search capacity of Haskell's constraint solver, and so requires the theorem proving to be more explicit. There is certainly room for improvement in both settings.

5. What are the Data that Go with Proofs?

In the previous section we gave ordering proofs as instances of the OWOTO data type. In this section, and even more so in the next, we will be concerned not only with the fact of ordering, but also the degree of it.

Let us consider the operation of comparing two singleton natural numbers. We refine the standard Haskell Ordering type to be indexed by the natural numbers under comparison.

As a naïve first attempt, we might copy the following definition from McBride and McKinna [14]:

```
data Cmp :: Nat → Nat → ⋆ where
  LTNat  :: Cmp m (m :+ S z)
  EQNat :: Cmp m m
  GTNat :: Cmp (n :+ S z) n
```

If $m < n$, then there exists some z such that $n = m + (z + 1)$. Similarly if $m > n$ then there exists some z such that $m = n + (z + 1)$.

Following a comparison, it can be useful to be able to inspect the difference between two numbers. In the EQNat case, this is simply 0. In the other two cases it is $z + 1$, thus in each case we store a singleton representation of z as a witness.

```
data Cmp :: Nat → Nat → ⋆ where
  LTNat  :: Natty z → Cmp m (m :+ S z)
  EQNat ::               Cmp m m
  GTNat :: Natty z → Cmp (n :+ S z) n
```

Note that in more conventional dependently typed programming languages, such as Agda, it is not possible to write an equivalent of our naive definition of Cmp—the value of z must be provided as an argument to the LTNat and GTNat constructors.

We can now write a comparison function that constructs a suitable proof object:

```
cmp :: Natty m → Natty n → Cmp m n
cmp Zy       Zy       = EQNat
cmp Zy       (Sy n) = LTNat n
cmp (Sy m) Zy       = GTNat m
cmp (Sy m) (Sy n) = case cmp m n of
  LTNat z → LTNat z
  EQNat    → EQNat
  GTNat z → GTNat z
```

The procrustes function fits a vector of length m into a vector of length n, by padding or trimming as necessary. (Procrustes was a mythical Greek brigand who would make his unfortunate guests fit into an iron bed either by stretching their limbs or by chopping them off.)

```
procrustes :: a → Natty m → Natty n → Vec m a → Vec n a
procrustes p m n xs = case cmp m n of
  LTNat z  → vappend xs (vcopies (Sy z) p)
  EQNat    → xs
  GTNat z  → vtake n (proxy (Sy z)) xs
```

In both the less-than and greater-than cases, we need the evidence z provided by the Cmp data type; in the former, we even compute with it.

Dependently typed programming often combines testing with the acquisition of new data that is justified by the test—the difference, in this case—and the refinement of the data being tested—the discovery that one number is the other plus the difference. We make sure that every computation which analyses data has a type which characterizes what we expect to learn.

6. Boxes

Here we introduce our main example, an algebra for building size-indexed rectangular tilings, which we call simply *boxes*.

6.1 Two Flavours of Conjunction

In order to define size indexes we introduce some kit which turns out to be more generally useful. The type of sizes is given by the *separated conjunction* [23] of Natty with Natty.

```
type Size = Natty :**: Natty

data (p :: ι → ⋆) :**: (q :: κ → ⋆) :: (ι, κ) → ⋆ where
  (:&&:) :: p ι → q κ → (p :**: q) '(ι, κ)
```

In general, the separating conjunction (:**:) of two indexed type constructors is an indexed product whose index is also a product, in which each component of the indexed product is indexed by the corresponding component of the index.

We also define a *non-separating conjunction*.

```
data (p :: κ → ⋆) :*: (q :: κ → ⋆) :: κ → ⋆ where
  (:&:) :: p κ → q κ → (p :*: q) k
```

The non-separating conjunction (:*:) is an indexed product in which the index is shared across both components of the product.

We will use both separating and non-separating conjunction extensively in Section 7.2.

6.2 The Box Data Type

We now introduce the type of boxes.

```
data Box :: ((Nat, Nat) → ⋆) → (Nat, Nat) → ⋆ where
  Stuff :: p wh → Box p wh
  Clear :: Box p wh
  Hor   :: Natty w1 → Box p '(w1, h) →
           Natty w2 → Box p '(w2, h) → Box p '(w1 :+ w2, h)
  Ver   :: Natty h1 → Box p '(w, h1) →
           Natty h2 → Box p '(w, h2) → Box p '(w, h1 :+ h2)
```

A box b with content of size-indexed type p and size wh has type Box p wh. Boxes are constructed from content (Stuff), clear boxes (Clear), and horizontal (Hor) and vertical (Ver) composition. Given suitable instantiations for the content, boxes can be used as the building blocks for arbitrary graphical user interfaces. In Section 7 we instantiate content to the type of character matrices, which we use to implement a text editor.

Though Box clearly does not have the right type to be an instance of the Monad type class, it is worth noting that it is a perfectly ordinary monad over a slightly richer base category than the category of Haskell types used by the Monad type class. The objects in this category are indexed. The morphisms are inhabitants of the following :→ type.

```
type s :→ t = ∀i.s i → t i
```

Let us define a type class of monads over indexed types.

```
class MonadIx (m :: (κ → ⋆) → (κ → ⋆)) where
  returnIx :: a :→ m a
  extendIx :: (a :→ m b) → (m a :→ m b)
```

The returnIx method is the unit, and extendIx is the Kleisli extension of a monad over indexed types. It is straightforward to provide an instance for boxes.

```
instance MonadIx Box where
  returnIx                     = Stuff
  extendIx f (Stuff c)         = f c
  extendIx f Clear             = Clear
  extendIx f (Hor w1 b1 w2 b2) =
    Hor w1 (extendIx f b1) w2 (extendIx f b2)
  extendIx f (Ver h1 b1 h2 b2) =
    Ver h1 (extendIx f b1) h2 (extendIx f b2)
```

The extendIx operation performs substitution at Stuff constructors, by applying its first argument to the content.

Monads over indexed sets, in general, are explored in depth in the second author's previous work [12].

6.3 Juxtaposition

A natural operation to define is the one that juxtaposes two boxes together, horizontally or vertically, adding appropriate padding if the sizes do not match up. Let us consider the horizontal version juxH. Its type signature is:

```
juxH :: Size '(w1, h1) → Size '(w2, h2) →
          Box p '(w1, h1) → Box p '(w2, h2) →
            Box p '(w1 :+ w2, Max h1 h2)
```

where Max computes the maximum of two promoted Nats:

```
type family Max (m :: Nat) (n :: Nat) :: Nat
type instance Max Z     n     = n
type instance Max (S m) Z     = S m
type instance Max (S m) (S n) = S (Max m n)
```

As well as the two boxes it takes singleton representations of their sizes, as it must compute on these.

We might try to write a definition for juxH as follows:

```
juxH (w1 :&&: h1) (w2 :&&: h2) b1 b2 =
  case cmp h1 h2 of
    LTNat n →
      Hor w1 (Ver h1 b1 (Sy n) Clear) w2 b2   -- (×)
    EQNat   →
      Hor w1 b1 w2 b2                         -- (×)
    GTNat n →
      Hor w1 b1 w2 (Ver h2 b2 (Sy n) Clear)   -- (×)
```

Unfortunately, this code does not type check, because GHC has no way of knowing that the height of the resulting box is the maximum of the heights of the component boxes.

6.4 Pain

One approach to resolving this issue is to encode lemmas, given by parameterised equations, as Haskell functions. In general, such lemmas may be encoded as functions of type:

$$\forall x_1 \ldots x_n.\mathsf{Natty}\ x_1 \to \ldots \to \mathsf{Natty}\ x_n \to ((l \sim r) \Rightarrow t) \to t$$

where l and r are the left- and right-hand-side of the equation, and $x_1, \ldots, x_n$ are natural number variables that may appear free in the equation. The first n arguments are singleton natural numbers. The last argument represents a context that expects the equation to hold.

For juxH, we need one lemma for each case of the comparison:

```
juxH (w1 :&&: h1) (w2 :&&: h2) b1 b2 =
  case cmp h1 h2 of
    LTNat z  → maxLT h1 z $
      Hor w1 (Ver h1 b1 (Sy z) Clear) w2 b2
    EQNat    → maxEQ h1 $
      Hor w1 b1 w2 b2
    GTNat z → maxGT h2 z $
      Hor w1 b1 w2 (Ver h2 b2 (Sy z) Clear)
```

Each lemma is defined by a straightforward induction:

```
maxLT :: ∀m z t.Natty m → Natty z →
            ((Max m (m :+ S z) ∼ (m :+ S z)) ⇒ t) → t
maxLT Zy      z t = t
maxLT (Sy m) z t = maxLT m z t

maxEQ :: ∀m t.Natty m → ((Max m m ∼ m) ⇒ t) → t
maxEQ Zy      t = t
maxEQ (Sy m) t = maxEQ m t

maxGT :: ∀n z t.Natty n → Natty z →
            ((Max (n :+ S z) n ∼ (n :+ S z)) ⇒ t) → t
maxGT Zy      z t = t
maxGT (Sy n) z t = maxGT n z t
```

Using this pattern, it is now possible to use GHC as a theorem prover. As GHC does not provide anything in the way of direct support for theorem proving (along the lines of tactics in Coq, say), we would like to avoid the pain of explicit theorem proving as much as possible, so we now change tack.

6.5 Pleasure

In order to avoid explicit calls to lemmas we would like to obtain the type equations we need for free as part of the proof object. As a first step, we observe that this is essentially what we are already doing in the proof object to encode the necessary equations concerning addition. One can always rephrase a GADT as an existential algebraic data type with suitable type equalities. For our basic Cmp data type, this yields:

```
data Cmp :: Nat → Nat → ⋆ where
  LTNat :: ((m :+ S z) ∼ n)  ⇒ Natty z → Cmp m n
  EQNat :: (m ∼ n)           ⇒            Cmp m n
  GTNat :: (m ∼ (n :+ S z)) ⇒ Natty z → Cmp m n
```

Now the fun starts. As well as the equations that define the proof object, we can incorporate other equations that encapsulate further knowledge implied by the result of the comparison. For now, we add equations for computing the maximum of m and n in each case.

```
data Cmp :: Nat → Nat → ⋆ where
  LTNat :: ((m :+ S z) ∼ n, Max m n ∼ n)  ⇒
    Natty z → Cmp m n
  EQNat :: (m ∼ n,            Max m n ∼ m) ⇒
              Cmp m n
  GTNat :: (m ∼ (n :+ S z), Max m n ∼ m) ⇒
    Natty z → Cmp m n
```

Having added these straightforward equalities, our definition of juxH now type checks without the need to explicitly invoke any lemmas.

```
juxH :: Size '(w1, h1) → Size '(w2, h2) →
          Box p '(w1, h1) → Box p '(w2, h2) →
            Box p '(w1 :+ w2, Max h1 h2)
juxH (w1 :&&: h1) (w2 :&&: h2) b1 b2 =
  case cmp h1 h2 of
    LTNat z  →
      Hor w1 (Ver h1 b1 (Sy z) Clear) w2 b2
    EQNat    →
      Hor w1 b1 w2 b2
    GTNat z →
      Hor w1 b1 w2 (Ver h2 b2 (Sy z) Clear)
```

The juxV function is defined similarly.

As we shall see in Section 6.6, it can be useful to attach further equational constraints to the Cmp constructors. A limitation of our current formulation is that we have to go back and modify the Cmp data type each time we wish to extract new evidence from the cmp function. The code of cmp remains the same, and typechecks without explicit proof provided the induction which establishes the evidence fits with the recursion pattern. Ideally we would have some way to abstract Cmp and cmp over properties, but it seems hard to deliver the same implicit checking of ‘fitting the pattern’ without higher-order constraints, which are currently unsupported in Haskell. We leave a proper investigation to future work.

6.6 Cutting

For cutting up boxes, and two-dimensional entities in general, we introduce a type class Cut.

```
class Cut (p :: (Nat, Nat) → ⋆)  where
  horCut :: Natty m → Natty n →
              p '(m :+ n, h) → (p '(m, h), p '(n, h))
  verCut :: Natty m → Natty n →
              p '(w, m :+ n) → (p '(w, m), p '(w, n))
```

We can cut horizontally or vertically by supplying the width or height of the two smaller boxes we wish to cut a box into. Thus horCut takes natural numbers m and n, an indexed thing of width $m + n$ and height h, and cuts it into two indexed things of height h, one of width m, and the other of width n. The verCut function is similar.

In order to handle the case in which we horizontally cut the horizontal composition of two boxes, we need to perform a special kind of comparison. In general, we wish to compare natural numbers a and c given the equation $a + b = c + d$, and capture the constraints on a, b, c, and d implied by the result of the comparison. For instance, if $a < c$ then there must exist some number z, such that $b = (z + 1) + d$ and $c = a + (z + 1)$.

We encode proof objects for cut comparisons using the following data type.

```
data CmpCuts :: Nat → Nat → Nat → Nat → ⋆ where
  LTCuts :: (b ∼ (S z :+ d), c ∼ (a :+ S z)) ⇒
    Natty z → CmpCuts a b c d
  EQCuts :: (a ∼ c, b ∼ d) ⇒
              CmpCuts a b c d
  GTCuts :: (a ∼ (c :+ S z), d ∼ (S z :+ b)) ⇒
    Natty z → CmpCuts a b c d
```

We can straightforwardly define a cut comparison function.

```
cmpCuts :: ((a :+ b) ∼ (c :+ d)) ⇒
  Natty a → Natty b →
  Natty c → Natty d →
    CmpCuts a b c d
cmpCuts Zy      b Zy      d = EQCuts
cmpCuts Zy      b (Sy c) d = LTCuts c
cmpCuts (Sy a) b Zy      d = GTCuts a
cmpCuts (Sy a) b (Sy c) d = case cmpCuts a b c d of
  LTCuts z  → LTCuts z
  EQCuts    → EQCuts
  GTCuts z → GTCuts z
```

Now we define cuts for boxes.

```
instance Cut p ⇒ Cut (Box p)  where
  horCut m n (Stuff p) = (Stuff p1, Stuff p2)
    where (p1, p2) = horCut m n p
  horCut m n Clear = (Clear, Clear)
horCut m n (Hor w1 b1 w2 b2) =
  case cmpCuts m n w1 w2 of
    LTCuts z → let (b11, b12) = horCut m (Sy z) b1
               in (b11, Hor (Sy z) b12 w2 b2)
    EQCuts   → (b1, b2)
    GTCuts z → let (b21, b22) = horCut (Sy z) n b2
               in (Hor w1 b1 (Sy z) b21, b22)
horCut m n (Ver h1 b1 h2 b2) =
  (Ver h1 b11 h2 b21, Ver h1 b12 h2 b22)
   where (b11, b12) = horCut m n b1
     (b21, b22) = horCut m n b2

verCut m n b = ...
```

The interesting case occurs when horizontally cutting the horizontal composition of two sub-boxes. We must identify which sub-box the cut occurs in, and recurse appropriately. Note that we rely on being able to cut content. The definition of vertical box cutting is similar.

6.7 Boxes as Monoids

As well as monadic structure, boxes also have monoidal structure.

```
instance Cut p ⇒ Monoid (Box p wh)  where
  mempty = Clear
  mappend b Clear             = b
  mappend Clear b′            = b′
  mappend b@(Stuff _) _       = b
  mappend (Hor w1 b1 w2 b2) b′ =
    Hor w1 (mappend b1 b1′) w2 (mappend b2 b2′)
      where (b1′, b2′) = horCut w1 w2 b′
  mappend (Ver h1 b1 h2 b2) b′  =
    Ver h1 (mappend b1 b1′) h2 (mappend b2 b2′)
      where (b1′, b2′) = verCut h1 h2 b′
```

The multiplication operation b‘mappend‘b' overlays b on top of b'. It makes essential use of cutting to handle the Hor and Ver cases.

6.8 Cropping = Clipping + Fitting

We can *crop* a box to a region. First we need to specify a suitably indexed type of regions. A point identifies a position inside a box, where (Zy, Zy) represents the top-left corner, counting top-to-bottom, left-to-right.

```
type Point = Natty :**: Natty
```

A region identifies a rectangular area inside a box by a pair of the point representing the top-left corner of the region, and the size of the region.

```
type Region = Point :**: Size
```

We decompose cropping into two parts, *clipping* and *fitting*.

Clipping discards everything to the left and above the specified point. The type signature of clip is:

```
clip :: Cut p ⇒ Size ′(w, h) → Point ′(x, y) →
  Box p ′(w, h) → Box p ′(w :− x, h :− y)
```

where :− is type level subtraction:

```
type family (m :: Nat) :− (n :: Nat) :: Nat
type instance Z    :− n    = Z
type instance S m :− Z     = S m
type instance S m :− S n = (m :− n)
```

In order to account for the subtraction in the result, we need to augment the Cmp data type to include the necessary equations.

```
data Cmp :: Nat → Nat → ⋆ where
  LTNat :: ((m :+ S z) ∼ n, Max m n ∼ n, (m :− n) ∼ Z)   ⇒
    Natty z → Cmp m n
  EQNat :: (m ∼ n,          Max m n ∼ m, (m :− n) ∼ Z)   ⇒
            Cmp m n
  GTNat :: (m ∼ (n :+ S z), Max m n ∼ m, (m :− n) ∼ S z) ⇒
    Natty z → Cmp m n
```

To clip in both dimensions, we first clip horizontally, and then clip vertically.

In order to define clipping we first lift subtraction on types :− to subtract on singleton naturals /−/.

```
(/−/) :: Natty m → Natty n → Natty (m :− n)
Zy    /−/n     = Zy
Sy m/−/Zy     = Sy m
Sy m/−/Sy n = m/−/n
```

In general one needs to define each operation on naturals three times: once for Nat values, once for Nat types, and once for Natty values. The pain can be somewhat alleviated using the singletons library [7], which provides a Template Haskell extension to automatically generate all three versions from a single definition.

Let us now define clipping.

```
clip (w :&&: h) (x :&&: y) b =
  clipV (w/−/x :&&: h) y (clipH (w :&&: h) x b)

clipH :: Cut p ⇒ Size ′(w, h) → Natty x →
  Box p ′(w, h) → Box p ′(w :− x, h)
clipH (w :&&: h) x b = case cmp w x of
  GTNat d → snd (horCut x (Sy d) b)
  _           → Clear

clipV :: Cut p ⇒ Size ′(w, h) → Natty y →
  Box p ′(w, h) → Box p ′(w, h :− y)
clipV (w :&&: h) y b = case cmp h y of
  GTNat d → snd (verCut y (Sy d) b)
  _           → Clear
```

Fitting pads or cuts a box to the given size. To fit in both dimensions, we first fit horizontally, and then fit veritcally.

```
fit :: Cut p ⇒ Size ′(w1, h1) → Size ′(w2, h2) →
  Box p ′(w1, h1) → Box p ′(w2, h2)
fit (w1 :&&: h1) (w2 :&&: h2) b = fitV h1 h2 (fitH w1 w2 b)

fitH :: Cut p ⇒ Natty w1 → Natty w2 →
  Box p ′(w1, h) → Box p ′(w2, h)
fitH w1 w2 b = case cmp w1 w2 of
  LTNat d  → Hor w1 b (Sy d) Clear
  EQNat    → b
  GTNat d → fst (horCut w2 (Sy d) b)

fitV :: Cut p ⇒ Natty h1 → Natty h2 →
  Box p ′(w, h1) → Box p ′(w, h2)
fitV h1 h2 b = case cmp h1 h2 of
  LTNat d  → Ver h1 b (Sy d) Clear
  EQNat    → b
  GTNat d → fst (verCut h2 (Sy d) b)
```

Observe that fitH and fitV do essentially the same thing as the procrustes function, but on boxes rather than vectors, and always using Clear boxes for padding.

To crop a box to a region, we simply clip then fit.

```
crop :: Cut p ⇒ Region ′(′(x, y), ′(w, h)) → Size ′(s, t) →
  Box p ′(s, t) → Box p ′(w, h)
crop ((x :&&: y) :&&: (w :&&: h)) (s :&&: t) b =
  fit ((s/−/x) :&&: (t/−/y)) (w :&&: h)
    (clip (s :&&: t) (x :&&: y) b)
```

A convenient feature of our cropping code is that type-level subtraction is confined to the clip function. This works because in the type of fit the output box is independent of the size of the input box.

In an earlier version of the code we experimented with a more refined cropping function of type:

```
Cut p ⇒ Region ′(′(x, y), ′(w, h)) → Size ′(s, t) →
  Box p ′(s, t) → Box p ′(Min w (s :− x), Min h (t :− y))
```

where Min is minimum on promoted Nats. This proved considerably more difficult to use as we had to reason about interactions between subtraction, addition, and minimum. Moreover, the less-refined version is often what we want in practice.

7. An Editor

We outline the design of a basic text editor, which represents the text buffer as a size-indexed box. Using this representation guarantees that manipulations such as cropping the buffer to generate screen output only generate well-formed boxes of a given size. We will also need to handle dynamic values coming from the outside world. We convert these to equivalent size-indexed values using existentials, building on the Ex data type of Section 4 and the separating and non-separating conjunction operators of Section 6.1.

7.1 Character Boxes

A character box is a box whose content is given by character matrices.

```
type CharMatrix = Matrix Char
type CharBox = Box CharMatrix
```

Concretely, we will use a character box to represent a text buffer. We can fill an entire matrix with the same character.

```
matrixChar :: Char → Size wh → CharMatrix wh
matrixChar c (w :&&: h) = Mat (vcopies h (vcopies w c))
```

We can render a character box as a character matrix.

```
renderCharBox ::
  Size wh → CharBox wh → CharMatrix wh
renderCharBox _ (Stuff css) = css
renderCharBox wh Clear =
  matrixChar ' ' wh
renderCharBox (w :&&: _) (Ver h1 b1 h2 b2) =
  Mat (unMat (renderCharBox (w :&&: h1) b1)
    `vappend` unMat (renderCharBox (w :&&: h2) b2))
renderCharBox (_ :&&: h) (Hor w1 b1 w2 b2) =
  Mat (vcopies h vappend
         `vapp`   unMat (renderCharBox (w1 :&&: h) b1)
         `vapp`   unMat (renderCharBox (w2 :&&: h) b2))
```

Ideally, we would prefer to use the standard Applicative interface, but here we use vcopies h for pure and vapp for $(<\star>)$ to avoid the overhead of appealing to natter h.

We can display a character matrix as a list of strings.

```
stringsOfCharMatrix :: CharMatrix wh → [String]
stringsOfCharMatrix (Mat vs) =
  foldMap ((:[]) ∘ foldMap (:[])) vs
```

In order to be able to cut (and hence crop) boxes with matrix content we instantiate the Cut type class for matrices.

```
instance Cut (Matrix e) where
  horCut m _ (Mat ess) =
    (Mat (fst <$> ps), Mat (snd <$> ps)) where
      ps = vchop m <$> ess
  verCut m _ (Mat ess) = (Mat tess, Mat bess) where
    (tess, bess) = vchop m ess
```

7.2 Existentials

In Section 4 we introduced existentially quantified singletons as a means for taking dynamic values and converting them into equivalent singletons.

We now present combinators for constructing existentials over composite indexes. For the editor, we will need to generate a region, that is, a pair of pairs of singleton naturals from a pair of pairs of natural numbers.

```
wrapPair :: (a → Ex p) → (b → Ex q) → (a, b) → Ex (p :**: q)
wrapPair w1 w2 (x1, x2) =
  case (w1 x1, w2 x2) of
    (Ex v1, Ex v2) → Ex (v1 :&&: v2)
```

The wrapPair function wraps a pair of dynamic objects in a suitable existential package using a separating conjunction.

```
type WPoint = Ex Point
type WSize = Ex Size
type WRegion = Ex Region

intToNat :: Int → Nat
intToNat 0 = Z
intToNat n = S (intToNat (n − 1))

wrapInt    = wrapNat ∘ intToNat
wrapPoint  = wrapPair wrapInt wrapInt
wrapSize   = wrapPair wrapInt wrapInt
wrapRegion = wrapPair wrapPoint wrapSize
```

We might wish to wrap vectors, but the Vec type takes the length index first, so we cannot use it as is with Ex. Thus we can define and use a Flip combinator, which reverses the arguments of a two argument type-operator.

```
newtype Flip f a b = Flip {unFlip :: f b a}

type WVec a = Ex (Flip Vec a)

wrapVec :: [a] → WVec a
wrapVec []       = Ex (Flip V0)
wrapVec (x : xs) = case wrapVec xs of
  Ex (Flip v) → Ex (Flip (x :> v))
```

In fact, we wish to wrap a vector up together with its length. This is where the non-separating conjunction comes into play. The Natty representing the length of the vector and the Flip Vec a representing the vector itself should share the same index.

```
type WLenVec a = Ex (Natty :*: Flip Vec a)

wrapLenVec :: [a] → WLenVec a
wrapLenVec []       = Ex (Zy :&: Flip V0)
wrapLenVec (x : xs) = case wrapLenVec xs of
  Ex (n :&: Flip v) → Ex (Sy n :&: Flip (x :> v))
```

Similarly, we use non-separating conjunction to wrap a box with its size.

```
type WSizeCharBox = Ex (Size :*: CharBox)
```

Given a string of length w, we can wrap it as a character box of size $(w, 1)$.

```
wrapString :: String → WSizeCharBox
wrapString s = case wrapLenVec s of
  Ex (n :&: Flip v) →
    Ex ((n :&&: Sy Zy) :&: Stuff (Mat (pure v)))
```

Given a list of h strings of maximum length w, we can wrap it as a character box of size (w, h)

```
wrapStrings :: [String] → WSizeCharBox
wrapStrings []       = Ex ((Zy :&&: Zy) :&: Clear)
wrapStrings (s : ss) =
  case (wrapString s, wrapStrings ss) of
    (Ex ((w1 :&&: h1) :&: b1),
     Ex ((w2 :&&: h2) :&: b2)) →
       Ex (((w1 `maxn` w2) :&&: (h1/+/h2)) :&:
            juxV (w1 :&&: h1) (w2 :&&: h2) b1 b2)
```

where maxn is maximum and $(/+/)$ is addition on singleton natural numbers:

```
maxn :: Natty m → Natty n → Natty (Max m n)
maxn Zy      n      = n
maxn (Sy m) Zy      = Sy m
maxn (Sy m) (Sy n) = Sy (maxn m n)

(/+/) :: Natty m → Natty n → Natty (m :+ n)
Zy    /+/n    = n
Sy m /+/n    = Sy (m/+/n)
```

Curiously, the singletons library does not appear to provide any special support for existential quantification over singletons. It should be possible to automatically generate the code for wrapping dynamic objects in existentials.

We note also that the tendency to use stock data type components, e.g., Ex, Flip, :∗: and :∗∗:, causes extra layering of wrapping constructors in *patterns* and expressions. We could use a bespoke GADT for each type we build in this way, but that would make it harder to develop library functionality. Ordinary 'let' allows us to hide the extra layers in expressions, but is no help for patterns, which are currently peculiar in that they admit no form of definitional abstraction [2]. This basic oversight would be readily remedied by *pattern synonyms*—linear, constructor-form definitions which expand like macros either side of the = sign.

7.3 Cursors

We use a zipper structure [8] to represent a cursor into a text buffer. We make no attempt to statically track the size of the buffer as a cursor, but do so when we wish to manipulate the whole buffer.

A cursor is a triple consisting of: a backwards list of elements before the current position, the object at the current position, and a forward list of elements after the current position.

```
type Cursor a m = ([a], m, [a])
```

The elements of a StringCursor are characters.

```
type StringCursor = Cursor Char ()
```

The elements of a TextCursor are strings. The object at the current position is a StringCursor.

```
type TextCursor = Cursor String StringCursor
```

The deactivate and activate functions convert between a unit cursor and a pair of a list and its length.

```
deactivate :: Cursor a () → (Int, [a])
deactivate c = outward 0 c  where
  outward i ([], (), xs)        = (i, xs)
  outward i (x : xz, (), xs) = outward (i + 1) (xz, (), x : xs)

activate :: (Int, [a]) → Cursor a ()
activate (i, xs) = inward i ([], (), xs)  where
  inward _ c@(_, (), [])   = c
  inward 0 c                    = c
  inward i (xz, (), x : xs) = inward (i − 1) (x : xz, (), xs)
```

The whatAndWhere function uses deactivate and wrapStrings to generate a well-formed existentially quantified box from a TextCursor.

```
whatAndWhere :: TextCursor → (WSizeCharBox, (Int, Int))
whatAndWhere (czz, cur, css) = (wrapStrings strs, (x, y))
   where
     (x, cs) = deactivate cur
     (y, strs) = deactivate (czz, (), cs : css)
```

7.4 The Inner Loop

We give a brief overview of the editor's inner loop. The full code is available as literate Haskell at `https://github.com/slindley/dependent-haskell/tree/master/Hasochism/Editor.lhs`

The current position in the text buffer is represented using a zipper structure over an unindexed list of strings. The current position and size of the screen is represented as two pairs of integers. On a change to the buffer, the inner loop proceeds as follows.

- Wrap the current screen position and size as a singleton region using wrapRegion.
- Unravel the zipper structure using whatAndWhere to reveal the underlying structure of the buffer as a list of strings.
- This invokes wrapStrings to wrap the list of strings as an existential over a suitably indexed CharBox.
- Crop the wrapped CharBox according to the wrapped singleton region.
- Render the result as a list of strings using stringsOfCharMatrix∘renderCharBox.

We take advantage of dependent types to ensure that cropping yields boxes of the correct size. The rest of the editor does not use dependent types. The wrapping functions convert non-dependent data into equivalent dependent data. Rendering does the opposite.

We expect that converting back and forth between raw and indexed data every time something changes is expensive. We leave a full performance evaluation to future work. One might hope to use indexed data everywhere. This is infeasible in practice, because of the need to interact with the outside world, and in particular foreign APIs (including the curses library we use for our text editor).

8. Conclusion

We have constructed and explored the use of the static-versus-dynamic/explicit-versus-implicit matrix of value-dependent quantifiers in Haskell. We have observed the awkwardness, but enjoyed the mere possibility, of dynamic quantification and used it to build substantial examples of sorting and box-tiling, where the establishment and maintenance of invariants is based not just on propagation of static indices, but on dynamic generation of evidence.

After some fairly hairy theorem proving, the worst of which we have spared you, we learned how to package proofs which follow a similar pattern inside GADTs of useful evidence. GHC's constraint solver is a good enough automatic theorem prover to check the proof steps corresponding to the recursion structure of the evidence-generating program. Case analysis on the resulting evidence is sufficient to persuade GHC that sorting invariants hold and that boxes snap together. In this respect, Haskell handles simple proofs much more neatly than Agda, where proving is as explicit as programming because it is programming. There is still room for improvement: we do not yet have a *compositional* way to express just the *fact* that properties follow by a common proof pattern in a way that GHC will silently check.

There is room for improvement also in the treatment of dependent quantification, both in Haskell and in dependently typed programming languages. Haskell naturally gives good support for quantifying over data which are purely static, whilst Agda insists on retaining these data at run time. Meanwhile, the singletons shenanigans required to support the dynamic quantifiers are really quite painful, both conceptually—with the explosion of Nat, Natty, NATTY and WNat—and in the practicalities of shuffling between them, spending effort on converting values into singletons and singletons into dictionaries containing exact copies of those singletons. If we want to build a scalable technology with the precision of indexing we have shown in our examples, we had better look for foundations which allow the elimination of this complexity, not just the encoding of it.

The key step which we must take is to move on from Milner's alignment of coincidences and stop working as if a single dependent static implicit quantifier over types is all we need. We have encoded quantification over the same type in different ways by abstracting over different types in the same way, and the result is predictably and, we hope, preventably unpleasant. The Strathclyde team are actively exploring the remedy—generalizing the quantifier to reflect its true diversity, and allowing each type to be used unduplicated wherever it is meaningful. The best thing about banging your head off a brick wall is *stopping*.

References

[1] A. Abel and G. Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1), 2012.

[2] W. Aitken and J. Reppy. Abstract value constructors. Technical Report TR 92-1290, Cornell University, 1992.

[3] B. Barras and B. Bernardo. The implicit calculus of constructions as a programming language with dependent types. In *FoSSaCS*, volume 4962 of *LNCS*, pages 365–379, 2008.

[4] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

[5] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In *TYPES*, volume 3085 of *LNCS*, pages 115–129. Springer, 2003.

[6] S. Cui, K. Donnelly, and H. Xi. ATS: A language that combines programming with theorem proving. In *FroCoS*, volume 3717 of *LNCS*, pages 310–320. Springer, 2005.

[7] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell*, pages 117–130. ACM, 2012.

[8] G. P. Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[9] O. Kiselyov and C.-c. Shan. Lightweight static resources: Sexy types for embedded and systems programming. In *TFP*, 2007.

[10] J. P. Magalhães. The right kind of generic programming. In *WGP*, pages 13–24. ACM, 2012.

[11] C. McBride. A Case For Dependent Families. Seminar at LFCS, Edinburgh. `http://www.strictlypositive.org/a-case/`, 2000.

[12] C. McBride. Kleisli arrows of outrageous fortune, 2011. Accepted for publication.
`https://personal.cis.strath.ac.uk/conor.mcbride/Kleisli.pdf`.

[13] C. McBride. The Strathclyde Haskell Enhancement. `https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`, 2013.

[14] C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[15] C. McBride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[16] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[17] R. Milner, M. Tofte, and R. Harper. *The Definition of standard ML*. MIT Press, 1990.

[18] A. Miquel. The implicit calculus of constructions. In *TLCA*, LNCS, pages 344–359. Springer, 2001.

[19] S. Monnier and D. Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *PLPV*, pages 1–8. ACM, 2010.

[20] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[21] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *POPL*. ACM, 1989.

[22] R. Pollack. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, 1990.

[23] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[24] M. Sulzmann, M. M. T. Chakravarty, S. L. P. Jones, and K. Donnelly. System F with type equality coercions. In *TLDI*, pages 53–66. ACM, 2007.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76. ACM, 1989.

[26] S. Weirich, J. Hsu, and R. A. Eisenberg. Towards dependently typed Haskell: System FC with kind equality. In *ICFP*. ACM, 2013.

[27] B. A. Yorgey, S. Weirich, J. Cretin, S. L. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *TLDI*, pages 53–66. ACM, 2012.