Faking It

Simulating Dependent Types in Haskell

Conor McBride
Department of Computer Science
University of Durham
c.t.mcbride@durham.ac.uk

Abstract

Dependent types reflect the fact that validity of data is often a *relative* notion by allowing prior data to affect the types of subsequent data. Not only does this make for a *precise* type system, but also a highly *generic* one: both the type and the program for each instance of a family of operations can be computed from the data which *codes* for that instance.

Recent experimental extensions to the Haskell *type class* mechanism give us strong tools to relativize types to other *types*. We may simulate some aspects of dependent typing by making counterfeit type-level copies of data, with type constructors simulating data constructors and type classes simulating datatypes. This paper gives examples of the technique and discusses its potential.

1 Introduction

I am a relatively recent convert to Haskell, but my background gives me quite a curious perspective on my new-found friend: I spent my PhD (McBride, 1999) using Standard ML to implement a prototype tool for dependently typed programming based on the proof assistant Lego (Luo & Pollack, 1992). Haskell's post-Hindley-Milner features such as *rank 2* polymorphism, *nested* types and *polymorphic* recursion are, for me, a much-needed advance on ML. I was also pleased to find that recent extensions to Haskell's *type class* mechanism beyond the Haskell 98 standard (Peyton Jones *et al.*, 1997; Jones, 2000; Jones & Peterson, 1999) allowed me simulate aspects of the dependently typed programs which I continue to explore.

Parametric polymorphism, as supported by Haskell 98, allows us to define operations which work uniformly for any instantiation of their type (or type constructor) parameters. However, there are many generic families of operations which can be described 'systematically', but where the 'system' cannot be expressed by simply abstracting a parameter. The standard Haskell prelude defines many operations, such as *zipWith*, multiply for instances of the same scheme. We need a more powerful means of *programming* with types if we want to code up the schemes themselves.

Types are first class objects in dependent type systems: they may be passed as arguments and computed by functions from other types or from ordinary data. Type-level programming is just ordinary programming which happens to involve types, and the systematic construction of types for generic operations is correspondingly straightforward. Section 2 gives a brief overview of the techniques involved.

Haskell's developers did not set out to create a type-level programming facility, but non-standard extensions with multi-parameter type classes and functional dependencies nonetheless provide the rudiments of one, albeit serendipitously. Section 3 describes these extensions, and this curious way of using them. A collection of examples follows, including *zipWith* (section 4) and a suite of operations on arbitrary-length vectors (section 5).

The paper closes with a discussion of what can be learned from these examples. I have no illusions that the type class mechanism is the ideal way to implement the genericity which this paper illustrates. Rather, I hope that this happy discovery will lead to a more principled technology which allows us to 'do it for real'.

2 Dependent Types and Type-Level Programming

Dependent type systems have evolved over many years (Martin-Löf, 1971) to reach their current highly expressive form, and this paper is not the place for a full account. I can recommend Luo's 'Computation and Reasoning' (Luo, 1994) to the reader in search of more detail. This section serves to outline the application of dependent types to the programming issues addressed in this paper. I have kept the notation as close to Haskell as I can, and I make a typographical distinction between identifiers for *terms* and identifiers for types to clarify the levels at work.

The key contribution of dependent types is the idea of a *type family*, represented by a function $\mathsf{F} :: \mathsf{T} \to \mathsf{Type}$. To the machine, F is not a special kind of function, but we can see it as a collection of types, indexed by 'codes' in T. T can be any type we like—a datatype, a function space or even Type itself.[1] An ordinary application $\mathsf{F}\, t$ yields a type in the family, and we can generalise over just the types in the family by λ-abstracting over an arbitrary t in T. The type of such an abstraction binds t with a $\forall$-quantifier, allowing the range type to refer to it. For example, an equality test which works for any member of the family would have type

$$eqF :: \forall t :: \mathsf{T}.\ \mathsf{F}\, t \to \mathsf{F}\, t \to \mathsf{Bool}$$

Families of datatypes can be defined inductively, allowing us to equip data structures with built-in invariants. In this paper, we shall have need of the *vectors*—lists of a given length.

[1] A hierarchy of Type levels is maintained implicitly, to avoid paradox.

$$\begin{array}{ll}\textbf{data} & \mathsf{Vector} :: \mathsf{Nat} \to \mathsf{Type} \to \mathsf{Type} \\ \textbf{where} & \quad [] :: \forall \mathsf{A} :: \mathsf{Type}.\ \mathsf{Vector}\ \mathit{Zero}\ \mathsf{A} \\ & (:) :: \forall \mathsf{A} :: \mathsf{Type}.\ \forall n :: \mathsf{Nat}.\ \mathsf{A} \to \mathsf{Vector}\ n\ \mathsf{A} \to \mathsf{Vector}\ (\mathit{Suc}\ n)\ \mathsf{A}\end{array}$$

The presence of explicit length information allows us to enforce stricter static control on the usage of vector operations. For example, we can ensure that the 'tail' operation is applied only to *nonempty* vectors:

$$\begin{array}{l}\mathit{vTail} :: \forall \mathsf{A} :: \mathsf{Type}.\ \forall n :: \mathsf{Nat}.\ \mathsf{Vector}\ (\mathit{Suc}\ n)\ \mathsf{A} \to \mathsf{Vector}\ n\ \mathsf{A} \\ \mathit{vTail}\ (x : \mathit{xs}) \ = \ \mathit{xs}\end{array}$$

Programming with dependent types is much less convoluted in practice than it might seem at first glance because the machine can fill in details which are forced by type, such as the A and n arguments for vTail. In addition, the need for 'exception handling' code is greatly reduced: vTail has no [] case, because [] is not in its domain.

Data structures can be used to store types, just as they store other data. An inhabitant of $\mathsf{Vector}\ n\ \mathsf{Type}$ is a list of n types. A vector of 'source' types together with a 'target' type give a 'code' for a function space, which we can 'decode' as follows:[2]

$$\begin{array}{rcccl}(\rhd) :: \forall n :: \mathsf{Nat}.\ \mathsf{Vector}\ n\ \mathsf{Type} \to \mathsf{Type} \to \mathsf{Type} \\ [] & \rhd & \mathsf{T} & = & \mathsf{T} \\ (\mathsf{S} : \mathsf{Ss}) & \rhd & \mathsf{T} & = & \mathsf{S} \to \mathsf{Ss} \rhd \mathsf{T}\end{array}$$

We can now use $\rhd$ to give types to generic operators. For example, Haskell's *flip* operator (bringing the second argument of a function to the front), generalises to an operator which brings any chosen argument to the front:

$$\begin{array}{l}\mathit{nthFront} :: \forall n :: \mathsf{Nat}.\ \forall \mathsf{Ss} :: \mathsf{Vector}\ n\ \mathsf{Type}.\ \forall \mathsf{T}, \mathsf{U} :: \mathsf{Type}. \\ \qquad\qquad (\mathsf{Ss} \rhd \mathsf{T} \to \mathsf{U}) \to \mathsf{T} \to \mathsf{Ss} \rhd \mathsf{U} \\ \mathit{nthFront}\ []\ f = \ f \\ \mathit{nthFront}\ (\mathsf{S} : \mathsf{Ss})\ f\ t\ s \ = \ \mathit{nthFront}\ \mathsf{Ss}\ (f\ s)\ t\end{array}$$

Familiar utility functions acquire new uses at the type level—the *map* function, for example, allows this type for *vZipWith* (vectorised n-ary application):

$$\begin{array}{l}\mathit{vZipWith} :: \forall n, m :: \mathsf{Nat}.\ \forall \mathsf{Ss} :: \mathsf{Vector}\ n\ \mathsf{Type}.\ \forall \mathsf{T} :: \mathsf{Type}. \\ \qquad\qquad (\mathsf{Ss} \rhd \mathsf{T}) \to (\mathit{map}\ (\mathsf{Vector}\ m)\ \mathsf{Ss}) \to \mathsf{Vector}\ m\ \mathsf{T}\end{array}$$

Dependently typed programs are quite sophisticated, in that the well-typedness of one often relies on the computational behaviour of another. However, in order to understand such programs, we need only grasp one language, with a uniform operational semantics.

[2] $\rhd$ has the same precedence and as $\to$ and also associates rightwards.

3 Multi-Parameter Type Classes with Functional Dependencies

Both ghc and hugs have options, disabled by default, supporting *multiple parameter type classes* (Kaes, 1988; Wadler & Blott, 1989; Peyton Jones *et al.*, 1997). These allow us to define n-ary relations on types (and type constructors), equipped with overloaded *member* operations over the related types. A standard example, adapted from (Jones, 2000), is the relation $\mathsf{Collects\ ce\ e}$ which indicates that ce's elements can be seen as collections of e's, with members for insertion, and so on.

$$\begin{array}{l} \textbf{class}\ \mathsf{Collects\ ce\ e}\ \textbf{where} \\ \quad \mathit{insert} :: \mathsf{e} \to \mathsf{ce} \to \mathsf{ce} \\ \quad \mathit{element} :: \mathsf{e} \to \mathsf{ce} \to \mathsf{Bool} \end{array}$$

We can define systems of *instances* allowing various representations, provided the compiler can tell from any usage of a member which instance to employ. For example, lists or characteristic functions can collect elements of any equality type. We can also construct new collection structures, such as hash tables, from more basic ones. I omit the standard implementation details:

$$\begin{array}{l} \textbf{instance}\ \mathsf{Eq\ e} \Rightarrow \mathsf{Collects\ (e \to Bool)\ e}\ \textbf{where}\ \ldots \\ \textbf{instance}\ \mathsf{Eq\ e} \Rightarrow \mathsf{Collects\ [e]\ e}\ \textbf{where}\ \ldots \\ \textbf{instance}\ \mathsf{(Hashable\ e,\ Collects\ ce\ e)} \Rightarrow \mathsf{Collects\ (HashTable\ ce)\ e}\ \textbf{where}\ \ldots \end{array}$$

The opening part of the instance declaration is a Horn clause, indicating that the constraint right of the $\Rightarrow$ holds if those left of it hold also. The type of each member mentions everything related by the class, so determining which instance to use amounts to checking that the Horn clauses deliver a unique solution when all the parameters are known. The compiler both ensures and presumes that instance systems have this property.

However, suppose we want to add a member, $\mathit{empty} :: \mathsf{ce}$, to generate the empty collection for each instance. A usage of *empty* determines only one parameter, so we cannot presume to find a unique instance. This problem prompted Mark Jones to propose a system of annotations for multi-parameter classes, indicating *functional dependencies* (Jones, 2000). We may annotate the declaration of $\mathsf{Collects}$ to indicate that e must be *uniquely determined* by ce:[3]

$$\textbf{class}\ \mathsf{Collects\ ce\ e}\ \mid\ \mathsf{ce} \rightsquigarrow \mathsf{e}\ \textbf{where}\ \ldots$$

The compiler now enforces this dependency condition on the system of Horn clauses for $\mathsf{Collects}$, but also makes a stronger presumption about which member types it is safe to permit—e may be omitted, so *empty* is accepted.

[3] I use Jones's notation $(\mathsf{s}_1, \ldots, \mathsf{s}_m) \rightsquigarrow (\mathsf{t}_1, \ldots, \mathsf{t}_n)$, to indicate that the s's determine the t's. This is rendered in ASCII as `s1 ... sm -> t1 ... tn`

3.1 Classes for Type-level Programs and Data

Multi-parameter classes with functional dependencies are an effective way to achieve more flexible overloading. However, from my dependently typed perspective, I could not help noticing that they provide a way to say 'here is a (partial) function from types to types'. We now have a means to implement the type-level behaviour of systematic operations described by codes—if we can represent the codes as *types*. We have the programs, but where are the data?

A datatype contains exactly the values generated by its constructors.[4] We can make a type-level 'counterfeit' of a datatype by using a *class* to collect the types generated by some type constructors. Any first-order monomorphic datatype T can be lifted to a class $\mathsf{T}\,\mathsf{t}$, with each data constructor $C\,\mathsf{S}_1\,\ldots\,\mathsf{S}_n$ yielding both a type constructor and an instance declaration (where $\mathsf{S}_i\,\mathsf{s}_i$ is the class lifting each S_i):

$$\textbf{data}\ \mathsf{C}\,\mathsf{s}_1\ \ldots\ \mathsf{s}_n\ =\ C\,\mathsf{s}_1\ \ldots\ \mathsf{s}_n$$

$$\textbf{instance}\ (\mathsf{S}_1\,\mathsf{s}_1,\ldots,\mathsf{S}_n\,\mathsf{s}_n)\ \Rightarrow\ \mathsf{T}\,(\mathsf{C}\,\mathsf{s}_1\ \ldots\ \mathsf{s}_n)$$

For example, here are the type-level natural numbers:

$$\begin{array}{lrcl|l}
 & & & & \textbf{class}\ \mathsf{Nat\ n} \\
\textbf{data} & \mathsf{Zero} & = & \mathit{Zero} & \textbf{instance}\ \mathsf{Nat\ Zero} \\
\textbf{data} & \mathsf{Suc\ n} & = & \mathit{Suc}\ \mathsf{n} & \textbf{instance}\ \mathsf{Nat\ n} \Rightarrow \mathsf{Nat\ (Suc\ n)}
\end{array}$$

Each type in such a class has exactly one canonical inhabitant—the 'data-level code' for the type, where 'decoding' is just *type inference*. We can use this code in our programs when we want to pass some data to the typechecker. Some useful codes for Nat types:

$$\begin{array}{rcl}
\mathit{one} & = & \mathit{Suc\ Zero} :: \mathsf{Suc\ Zero} \\
\mathit{two} & = & \mathit{Suc\ one} :: \mathsf{Suc\ (Suc\ Zero)}
\end{array}$$

The members of these classes are less significant than their rôles as would-be inductive definitions. Of course, we can only pretend that Nat is *closed* under Zero and Suc: nothing but our consciences prevents subsequent spurious instances.

An n-ary type-level function becomes an $(n+1)$-parameter class with the last argument—the target type—depending functionally on the initial n source types. We may use class constraints to indicate that a parameter (source or target) is counterfeit data, or omit the constraint when we really mean to interpret a parameter as a type. A function Foo from T's to types is declared thus:

$$\textbf{class}\ \mathsf{T\ t}\ \Rightarrow\ \mathsf{Foo\ t\ u}\ \mid\ \mathsf{t} \rightsquigarrow \mathsf{u}$$

[4] and the undefined value, $\perp$

Foo's code is expressed in the Horn clauses of its instance declarations, and operations which exploit the type computed by Foo can be declared as members of Foo, with their code also scattered amongst the instances.

For example, let us define *nthFront*. Given a Natn and an appropriate input function type, we can compute required the output type by a multi-parameter class, of which *nthFront* becomes a method. I have written placeholders for the contents of the **where** clauses, in order to keep the type-level program together; the term-level program is written contiguously too, with placeholders left of each line to show where the code really goes:

$$\begin{array}{l}
\mathbf{class}\ \mathsf{Nat\ n} \Rightarrow \mathsf{NthFront\ n\ s\ t} \mid (\mathsf{n},\mathsf{s}) \leadsto \mathsf{t}\ \mathbf{where}\ \{\mathsf{NthFront}*\} \\
\mathbf{instance}\ \mathsf{NthFront\ Zero}\ (\mathsf{s} \to \mathsf{t})\ (\mathsf{s} \to \mathsf{t})\ \mathbf{where}\ \{\mathsf{NthFront}\dagger\} \\
\mathbf{instance}\ \mathsf{NthFront}\quad \mathsf{n} \qquad \mathsf{t} \qquad (\mathsf{a} \to \mathsf{b}) \quad \Rightarrow \\
\qquad\qquad \mathsf{NthFront}\ (\mathsf{Suc\ n})\ (\mathsf{s} \to \mathsf{t})\ (\mathsf{a} \to \mathsf{s} \to \mathsf{b})\ \mathbf{where}\ \{\mathsf{NthFront}\ddagger\}
\end{array}$$

$$\begin{array}{ll}
\{\mathsf{NthFront}*\} & \mathit{nthFront} :: \mathsf{n} \to \mathsf{s} \to \mathsf{t} \\
\{\mathsf{NthFront}\dagger\} & \mathit{nthFront}\ \mathit{Zero}\ f \ = \ f \\
\{\mathsf{NthFront}\ddagger\} & \mathit{nthFront}\ (\mathit{Suc}\ n)\ f\ a\ s \ = \ \mathit{nthFront}\ n\ (f\ s)\ a
\end{array}$$

The type class mechanism can pattern-match directly on types, so we do not need to split the input type into source vector and target. Of course, $\mathsf{NthFront}$ is not defined on all inputs: if the input type has insufficient arity, the program will fail to compute a target type. We can only get away with this sloppiness because both 'type errors' and 'run-time errors' in type-level code manifest themselves at compile-time for the overall program. The functional dependency is partial, but adequate to allow the following type inference:

$$\begin{array}{rl}
\text{given} \qquad \mathit{foldr} & :: (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to [\mathsf{a}] \to \mathsf{b} \\
\mathit{nthFront}\ \mathit{two}\ \mathit{foldr} & :: [\mathsf{a}] \to (\mathsf{a} \to \mathsf{b} \to \mathsf{b}) \to \mathsf{b} \to \mathsf{b}
\end{array}$$

On the other hand, applying *nthFront* to a number alone does not supply enough information for $\mathsf{NthFront}$ to do its job. The following can be *checked* but not inferred:

$$\mathit{nthFront}\ \mathit{one} :: (\mathsf{a} \to \mathsf{b} \to \mathsf{c}) \to \mathsf{b} \to \mathsf{a} \to \mathsf{c}$$

The inverse operation *frontNth*, pushing the first argument back to position $n+1$, can also be defined. Its type-level program just reverses the source and target type expressions. We do not need to define a new class $\mathsf{FrontNth}$: it is enough to tell the compiler that the program $\mathsf{NthFront\ n\ s\ t}$ has a second functional mode, $(\mathsf{n},\mathsf{t}) \leadsto \mathsf{s}$. We may now add another member:

$$\begin{array}{ll}
\{\mathsf{NthFront}*\} & \mathit{frontNth} :: \mathsf{n} \to \mathsf{t} \to \mathsf{s} \\
\{\mathsf{NthFront}\dagger\} & \mathit{frontNth}\ \mathit{Zero}\ f \ = \ f \\
\{\mathsf{NthFront}\ddagger\} & \mathit{frontNth}\ (\mathit{Suc}\ n)\ f\ s \ = \ \mathit{frontNth}\ n\ (\lambda a \mapsto f\ a\ s)
\end{array}$$

Again, the compiler can infer

$$\textit{frontNth two foldr} :: \mathsf{b} \rightarrow [\mathsf{a}] \rightarrow (\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{b}) \rightarrow \mathsf{b}$$

We can use these two operations to define more complex permutations on a function's arguments. For example, we may swap the front argument with any other, using this operation:

$$\textit{swapFrontArg}\; n\; f\; a \;=\; \textit{frontNth}\; n\; (\textit{nthFront}\; (\textit{Suc}\; n)\; f\; a)$$

The compiler successfully infers

$$\begin{aligned}\textit{swapFrontArg} :: (&\mathsf{NthFront}\;(\mathsf{Suc\;n})\;\mathsf{s}\;(\mathsf{a} \rightarrow \mathsf{t}),\; \mathsf{NthFront\;n\;t\;u}) \Rightarrow \\ &\mathsf{n} \rightarrow \mathsf{s} \rightarrow \mathsf{a} \rightarrow \mathsf{u}\end{aligned}$$

When we define a composite generic operators in this way, we acquire a class constraint corresponding to each usage of a class member. This can quickly lead to cumbersome types which make poor documentation of the operator, in comparison with the clearer descriptions afforded to their dependently typed counterparts via the $\rhd$ function. Nonetheless, type inference makes the burden of complex constraints more bearable, and it seems clear that we can readily use these this type class technique to build up a library of n-ary functional combinators.

4 *zipWith* rides again

The $\textit{zipWith}_n$ family of operators is a notorious example of a general pattern instantiated to different types and programs for different n, in a systematic way. $\textit{zipWith}_n$ takes a function $f :: \mathsf{s}_1 \rightarrow \cdots \mathsf{s}_n \rightarrow \mathsf{t}$ to a function in $[\mathsf{s}_1] \rightarrow \cdots [\mathsf{s}_n] \rightarrow [\mathsf{t}]$ which uses f to generate successive t's from successive batches of s's until one of the source lists $[\mathsf{s}_i]$ becomes empty.

The definition is notionally the following:[5]

$$\begin{aligned}&(\ll) :: [\mathsf{s} \rightarrow \mathsf{t}] \rightarrow [\mathsf{s}] \rightarrow [\mathsf{t}] \\ &(f : \textit{fs}) \ll (s : \textit{ss}) = (f\; s) : (\textit{fs} \ll \textit{ss}) \\ &\quad \_ \quad\; \ll \quad \_ \quad\; = []\end{aligned}$$

$$\begin{aligned}&\textit{zipWith}_n :: (\mathsf{s}_1 \rightarrow \cdots \mathsf{s}_n \rightarrow \mathsf{t}) \rightarrow [\mathsf{s}_1] \rightarrow \cdots [\mathsf{s}_n] \rightarrow [\mathsf{t}] \\ &\textit{zipWith}_n\; f\; \textit{ss}_1\; \ldots\; \textit{ss}_n \;=\; (\textit{repeat}\; f) \ll \textit{ss}_1 \ll \ldots \textit{ss}_n\end{aligned}$$

In (Fridlender & Indrika, 2000), Daniel Fridlender and Mia Indrika give an ingenious generic presentation of $\textit{zipWith}_n$ by means of 'numerals', as follows

[5] $\ll$ associates to the left, and *repeat* f returns an infinite list of f's

$$
\begin{aligned}
&\mathit{zero} :: [\mathsf{t}] \to [\mathsf{t}]\\
&\mathit{zero} = \mathit{id}\\
&\mathit{suc} :: ([\mathsf{s}] \to \mathsf{t}) \to [\mathsf{r} \to \mathsf{s}] \to [\mathsf{r}] \to \mathsf{t}\\
&\mathit{suc}\ n\ \mathit{fs}\ \mathit{ss} = n\ (\mathit{fs} \ll \mathit{ss})\\
\\
&\mathit{zipWith} :: ([\mathsf{s}] \to \mathsf{t}) \to \mathsf{s} \to \mathsf{t}\\
&\mathit{zipWith}\ n\ f = n\ (\mathit{repeat}\ f)
\end{aligned}
$$

It is indeed the case that if you apply *zipWith* to a numeral n generated from *zero* and *suc*, you get the appropriate member of the *zipWith* family. However, let nobody be under any illusion that such codings constitute a satisfactory alternative to dependent types. On the contrary, this definition of *zipWith* exploits polymorphism to *weaken* the type discipline—numerals are not *numbers*; numerals are *arbitrary functions on lists*! Consequently, the following is well-typed, even though it makes no sense (not to mention output):

$$\mathit{zipWith}\ \mathit{sum}\ 3$$

We could use an *abstract datatype* to package up the numerals, exporting only the *zero* and *suc* constructors, together with the 'eliminator' *zipWith*. This would make *zipWith* type-safe, but it would also significantly reduce our ability to manipulate numerals, the perennial disadvantage of abstract types—dependent types can express the same kind of security for the constructors without limiting the elimination behaviour. In effect, the answer to Fridlender and Indrika's question 'Are there generic numerals?' is 'Yes! The natural numbers!' Our type-level numbers can control the usage of *zipWith* precisely:

$$
\begin{aligned}
&\textbf{class}\ \mathsf{Nat\ n} \Rightarrow \mathsf{ZipWith\ n\ s\ t} \mid (\mathsf{n},\mathsf{s}) \rightsquigarrow \mathsf{t}\\
&\textbf{instance}\ \mathsf{ZipWith\ Zero\ t}\ [\mathsf{t}]\ \textbf{where}\ \{\mathsf{ZipWith}\dagger\}\\
&\textbf{instance}\ \mathsf{ZipWith}\quad \mathsf{n}\quad\ \mathsf{t}\quad\ \mathsf{ts}\quad \Rightarrow\\
&\qquad\quad \mathsf{ZipWith\ (Suc\ n)\ (s \to t)\ ([s] \to ts)}\ \textbf{where}\ \{\mathsf{ZipWith}\ddagger\}\\
\\
&\{\mathsf{ZipWith}*\}\quad \mathit{manyApp} :: \mathsf{n} \to [\mathsf{s}] \to \mathsf{t}\\
&\{\mathsf{ZipWith}\dagger\}\quad \mathit{manyApp}\ \mathit{Zero}\ \mathit{fs} = \mathit{fs}\\
&\{\mathsf{ZipWith}\ddagger\}\quad \mathit{manyApp}\ (\mathit{Suc}\ n)\ \mathit{fs}\ \mathit{ss} = \mathit{manyApp}\ n\ (\mathit{fs} \ll \mathit{ss})\\
\\
&\{\mathsf{ZipWith}*\}\quad \mathit{zipWith} :: \mathsf{n} \to \mathsf{s} \to \mathsf{t}\\
&\{\mathsf{ZipWith}*\}\quad \mathit{zipWith}\ n\ f = \mathit{manyApp}\ n\ (\mathit{repeat}\ f)
\end{aligned}
$$

What has happened? Not much has changed operationally. The type-level numbers stand as codes for the 'numerals' of the former implementation: the *manyApp* function decodes each n to its corresponding numeral. The point is that these codes give *exactly* the legitimate *zipWith*s and typechecking ensures valid codes.

However, I am still not satisfied. The expedient of stopping as soon as one of the argument lists becomes empty is necessary while we have no means of controlling the *lengths* of the lists. Argument lists of different length are more likely to arise by error than design; argument lists of the same length are easy to manufacture explicitly; it would seem desirable to ensure that all *zipwith*'s argument lists have the same length. We shall build this technology in the next section.

5 Vectors via Constructor Classes

We can use the type system to police the lengths of lists by using *vectors*. One way to achieve this is to represent a vector as an n-tuple over an element type a, computed from n by a multi-parameter class:

$$\begin{array}{l}
\mathbf{class}\ \mathsf{Nat\ n} \ \Rightarrow\ \mathsf{Vect\ n\ a\ v} \\
\mathbf{instance}\ \mathsf{Vect\ Zero\ a\ ()} \\
\mathbf{instance}\ \mathsf{Vect\ n\ a\ v} \ \Rightarrow\ \mathsf{Vect\ (Suc\ n)\ a\ (a, v)}
\end{array}$$

Whilst this approach is entirely reasonable in a dependent type system, it has a practical drawback in Haskell: we cannot integrate 'type constructors' generated by type class programming with Haskell's existing polymorphism at higher kinds. Although Vect corresponds to an operation in $\mathsf{Nat} \to \mathsf{Type} \to \mathsf{Type}$, we cannot use a 'curried class application', $\mathsf{Vect\ n}$, as a type constructor in the same sense as Maybe or the list constructor ($[]$). We cannot, for example, instantiate the $\mathsf{Functor}$ class for these vectors, although there is clearly a suitable *fmap* operator.

A better solution is to define the vectors directly, as the class of type constructors generated by a constant VNil constructor and a higher-order VCons constructor:

$$\begin{array}{l}
\mathbf{data}\ \mathsf{VNil\ a} \ =\ \mathit{VNil} \\
\mathbf{data}\ \mathsf{VCons\ v\ a} \ =\ \mathit{VCons}\ \mathsf{a\ (v\ a)} \\
\\
\mathbf{class}\ \mathsf{Vector\ v}\ \mathbf{where}\ \{\mathsf{Vector}{*}\} \\
\mathbf{instance}\ \mathsf{Vector\ VNil}\ \mathbf{where}\ \{\mathsf{Vector}\dagger\} \\
\mathbf{instance}\ \mathsf{Vector\ v} \ \Rightarrow\ \mathsf{Vector\ (VCons\ v)}\ \mathbf{where}\ \{\mathsf{Vector}\ddagger\}
\end{array}$$

Class Vector is thus another copy of the naturals, but this time our 'numbers' are type constructors rather than types. The corresponding term-level data they contain are not just codes for the type-level objects, but the actual vectors we seek to represent. This definition is much closer to the direct inductive definition of the vector family given in section 2 than the 'computed vectors' above.

We can define error-free head and tail operations for nonempty vectors without troubling the class mechanism:

$$\begin{array}{l|l}
\mathit{vHead} :: \mathsf{VCons\ v\ a} \ \to\ \mathsf{a} & \mathit{vTail} :: \mathsf{VCons\ v\ a} \ \to\ \mathsf{v\ a} \\
\mathit{vHead}\ (\mathit{VCons}\ x\ xs) \ =\ x & \mathit{vTail}\ (\mathit{VCons}\ x\ xs) \ =\ xs
\end{array}$$

Some recursive functions can be defined as members of Vector:

$$
\begin{array}{ll}
\{\mathsf{Vector}*\} & \mathit{vSnoc} :: \mathsf{v\ a} \to \mathsf{a} \to \mathsf{VCons\ v\ a} \\
\{\mathsf{Vector}\dagger\} & \mathit{vSnoc}\ \mathit{VNil}\ y \ = \ \mathit{VCons}\ y\ \mathit{VNil} \\
\{\mathsf{Vector}\ddagger\} & \mathit{vSnoc}\ (\mathit{VCons}\ x\ \mathit{xs})\ y \ = \ \mathit{VCons}\ x\ (\mathit{vSnoc}\ \mathit{xs}\ y)
\end{array}
$$

$$
\begin{array}{ll}
\{\mathsf{Vector}*\} & \mathit{vLast} :: \mathsf{VCons\ v\ a} \to \mathsf{a} \\
\{\mathsf{Vector}\dagger\} & \mathit{vLast}\ (\mathit{VCons}\ x\ \mathit{xs}) \ = \ x \\
\{\mathsf{Vector}\ddagger\} & \mathit{vLast}\ (\mathit{VCons}\ x\ \mathit{xs}) \ = \ \mathit{vLast}\ \mathit{xs}
\end{array}
$$

The latter may look nondeterministic; in fact, the *type* tells us when we have reached the end of the list, so there is no need to look ahead explicitly. The xs in the $\{\mathsf{Vector}\dagger\}$ case must be empty; the xs in the $\{\mathsf{Vector}\ddagger\}$ case must be nonempty[6]!

A small problem arises if we want to show that each vector type constructor is an instance of class $\mathsf{Functor}$. We can easily define *fmap* operators for each of VNil and $\mathsf{VCons\ v}$ (for functorial v):

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Functor\ VNil}\ \textbf{where} \\
\quad \mathit{fmap}\ f\ \mathit{VNil} \ = \ \mathit{VNil}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Functor\ v} \to \mathsf{Functor\ (VCons\ v)}\ \textbf{where} \\
\quad \mathit{fmap}\ f\ (\mathit{VCons}\ x\ \mathit{xs}) \ = \ \mathit{VCons}\ (f\ x)\ (\mathit{fmap}\ f\ \mathit{xs})
\end{array}
$$

However, this alone does not ensure that $\mathsf{Vector\ v}$ implies $\mathsf{Functor\ v}$. The trouble is that our counterfeit data structures are not really the closures of their constructors. We are still free to add new, non-functorial constructors to Vector. If we want every Vector to be a $\mathsf{Functor}$, we must modify the declaration of the Vector class itself, requiring functorial behaviour.

$$
\textbf{class}\ \mathsf{Functor\ v} \ \Rightarrow\ \mathsf{Vector\ v}\ \textbf{where}\ \ldots
$$

5.1 vZipWith

Let us now develop *vZipWith*, following what we did for lists. We shall need to write *vRepeat* and *vApply* so that

$$
\mathit{vZipWith}_n\ f\ \mathit{ss}_1\ \ldots\ \mathit{ss}_n \ = \ (\mathit{vRepeat}\ f)\ `\mathit{vApply}`\ \mathit{ss}_1\ `\mathit{vApply}`\ \ldots\ \mathit{ss}_n
$$

Instead of having one *repeat* function generating an infinite list, each vector type is equipped with a *vRepeat* member which replicates its argument an appropriate number of times—as with *vLast*, the 'appropriate number' is given by the type.

[6] for each case, presuming xs is not $\perp$

Meanwhile, the analogue of $\ll$ for vectors can rely on the vector of functions being exactly as long as the vector of arguments:

$$
\begin{array}{ll}
\{\mathsf{Vector}*\} & \mathit{vRepeat} :: \mathsf{a} \rightarrow \mathsf{v}\ \mathsf{a} \\
\{\mathsf{Vector}\dagger\} & \mathit{vRepeat}\ x \ = \ \mathit{VNil} \\
\{\mathsf{Vector}\ddagger\} & \mathit{vRepeat}\ x \ = \ \mathit{VCons}\ x\ (\mathit{vRepeat}\ x)
\end{array}
$$

$$
\begin{array}{ll}
\{\mathsf{Vector}*\} & \mathit{vApply} :: \mathsf{v}\ (\mathsf{s} \rightarrow \mathsf{t}) \rightarrow \mathsf{v}\ \mathsf{s} \rightarrow \mathsf{v}\ \mathsf{t} \\
\{\mathsf{Vector}\dagger\} & \mathit{vApply}\ \mathit{VNil}\ \mathit{VNil} \ = \ \mathit{VNil} \\
\{\mathsf{Vector}\ddagger\} & \mathit{vApply}\ (\mathit{VCons}\ f\ \mathit{fs})\ (\mathit{VCons}\ s\ \mathit{ss}) \ = \ \mathit{VCons}\ (f\ s)\ (\mathit{fs}\ \text{`}\mathit{vApply}\text{`}\ \mathit{ss})
\end{array}
$$

Now we can apply the technique of the previous section, adding a class parameter for the particular Vector v replacing ([]). Unfortunately, v does not appear in the type of *vZipWith*, but a second functional dependency resolves the ambiguity.

$$
\begin{array}{l}
\textbf{class}\ (\mathsf{Nat}\ \mathsf{n},\ \mathsf{Vector}\ \mathsf{v}) \Rightarrow \mathsf{VZipWith}\ \mathsf{n}\ \mathsf{v}\ \mathsf{s}\ \mathsf{t} \mid (\mathsf{n},\mathsf{v},\mathsf{s}) \leadsto \mathsf{t},\ (\mathsf{n},\mathsf{t}) \leadsto \mathsf{v} \\
\quad \textbf{where}\ \{\mathsf{VZipWith}*\} \\
\textbf{instance}\ \mathsf{Vector}\ \mathsf{v} \Rightarrow \mathsf{VZipWith}\ \mathsf{Zero}\ \mathsf{v}\ \mathsf{t}\ (\mathsf{v}\ \mathsf{t}) \\
\ 0.1in\textbf{where}\ \{\mathsf{VZipWith}\dagger\} \\
\textbf{instance}\ \mathsf{VZipWith}\quad \mathsf{n}\quad \mathsf{v}\quad \mathsf{t}\qquad \mathsf{ts}\quad \Rightarrow \\
\qquad\qquad \mathsf{VZipWith}\ (\mathsf{Suc}\ \mathsf{n})\ \mathsf{v}\ (\mathsf{s} \rightarrow \mathsf{t})\ (\mathsf{v}\ \mathsf{s} \rightarrow \mathsf{ts})\ \textbf{where}\ \{\mathsf{VZipWith}\ddagger\}
\end{array}
$$

$$
\begin{array}{ll}
\{\mathsf{VZipWith}*\} & \mathit{vManyApp} :: \mathsf{n} \rightarrow \mathsf{v}\ \mathsf{s} \rightarrow \mathsf{t} \\
\{\mathsf{VZipWith}\dagger\} & \mathit{vManyApp}\ \mathit{Zero}\ \mathit{fs} \ = \ \mathit{fs} \\
\{\mathsf{VZipWith}\ddagger\} & \mathit{vManyApp}\ (\mathit{Suc}\ n)\ \mathit{fs}\ \mathit{ss} \ = \ \mathit{vManyApp}\ n\ (\mathit{fs}\ \text{`}\mathit{vApply}\text{`}\mathit{ss})
\end{array}
$$

$$
\begin{array}{ll}
\{\mathsf{VZipWith}*\} & \mathit{vZipWith} :: \mathsf{n} \rightarrow \mathsf{s} \rightarrow \mathsf{t} \\
\{\mathsf{VZipWith}*\} & \mathit{vZipWith}\ n\ f \ = \ \mathit{vManyApp}\ n\ (\mathit{vRepeat}\ f)
\end{array}
$$

5.2 Fold Operators for Vector

It often proves useful to code up common patterns of recursion on datatypes as *fold operators*. Class Vector collects particular v's with kind $\star \rightarrow \star$, and there are many ways in which a recursive operation on v a might relate its return type to the particular choice of v. Unfortunately, the inhabitants of $(\star \rightarrow \star) \rightarrow \star$ expressible in Haskell are somewhat restricted—the language of types is not a programming language—hence we find ourselves writing a selection of fold operators, always with the same program, but with subtly different signatures.

The simplest of these fold operators throws away the structure of the vectors, yielding a uniform return value—this is sometimes known as a *crushing* fold:

$$
\begin{array}{ll}
\{\mathsf{Vector}*\} & \mathit{vCrush} :: \mathsf{t} \rightarrow (\mathsf{s} \rightarrow \mathsf{t} \rightarrow \mathsf{t}) \rightarrow \mathsf{v}\ \mathsf{s} \rightarrow \mathsf{t} \\
\{\mathsf{Vector}\dagger\} & \mathit{vCrush}\ n\ c\ \mathit{VNil} \ = \ n \\
\{\mathsf{Vector}\ddagger\} & \mathit{vCrush}\ n\ c\ (\mathit{VCons}\ x\ \mathit{xs}) \ = \ c\ x\ (\mathit{vCrush}\ n\ c\ \mathit{xs})
\end{array}
$$

For example, we can use *vCrush* to sum the elements of a numeric vector and thus compute the scalar product of two such vectors:

$$\begin{array}{l} vSum :: (\mathsf{Vector\ v},\ \mathsf{Num\ n}) \rightarrow \mathsf{v\ n} \rightarrow \mathsf{n} \\ vSum \ = \ vCrush\ 0\ (+) \\ \\ vDot :: (\mathsf{Vector\ v},\ \mathsf{Num\ n}) \rightarrow \mathsf{v\ n} \rightarrow \mathsf{v\ n} \rightarrow \mathsf{n} \\ vDot\ xs\ ys \ = \ vSum\ (vZipWith\ two\ (*)\ xs\ ys) \end{array}$$

A more complex fold operator preserves the vector structure v, but wraps it within a type constructor f and changes the element type from s to t. The type resembles an induction principle:

$$\begin{array}{ll} \{\mathsf{Vector}*\} & vWrap :: \mathsf{f\ (VNil\ t)} \rightarrow \\ & \qquad\qquad (\mathbf{forall}\ \mathsf{u.\ Vector\ u} \Rightarrow \mathsf{s} \rightarrow \mathsf{f\ (u\ t)} \rightarrow \mathsf{f\ (VCons\ u\ t)}) \rightarrow \\ & \qquad\qquad \mathsf{v\ s} \rightarrow \mathsf{f\ (v\ t)} \\ \{\mathsf{Vector}\dagger\} & vWrap\ n\ c\ VNil \ = \ n \\ \{\mathsf{Vector}\ddagger\} & vWrap\ n\ c\ (VCons\ x\ xs) \ = \ c\ x\ (vWrap\ n\ c\ xs) \end{array}$$

One operation ripe for definition with *vWrap* is *vTranspose*, which transposes a matrix represented as a vector of vectors.

$$\begin{array}{l} vTranspose :: (\mathsf{Vector\ v},\ \mathsf{Vector\ w}) \rightarrow \mathsf{v\ (w\ a)} \rightarrow \mathsf{w\ (v\ a)} \\ vTranspose \ = \ vWrap\ (vRepeat\ VNil)\ (vZipWith\ two\ VCons) \end{array}$$

From *vDot* and *vTranspose*, together with the functorial properties of vectors, we may define a precisely typed matrix multiplier:

$$\begin{array}{l} vMultiply :: (\mathsf{Vector\ u},\ \mathsf{Vector\ v},\ \mathsf{Vector\ w},\ \mathsf{Num\ n}) \rightarrow \\ \qquad\qquad\quad \mathsf{u\ (v\ n)} \rightarrow \mathsf{v\ (w\ n)} \rightarrow \mathsf{u\ (w\ n)} \\ vMultiply\ uv\ vw \ = \ fmap\ row\ uv\ \mathbf{where} \\ \quad row\ v \ = \ fmap\ (vDot\ v)\ wv \\ \quad wv \ = \ vTranspose\ vw \end{array}$$

These operations still display a very uniform relationship between the return type of the folds they employ and the vectors being traversed. If we want to go further, we will again need to use multi-parameter classes as a programming language. The following fold operator not only replaces each *VNil* with given *n* and *VCons* with given *c* at the *term* level, it repeats this pattern at the *type* level:

$$
\begin{array}{l}
\textbf{class}\ \mathsf{Vector\ v}\ \rightarrow\ \mathsf{VFold\ n\ c\ v\ t}\ \mid\ (\mathsf{n},\mathsf{c},\mathsf{v}) \rightsquigarrow \mathsf{t} \\
\ \textbf{where}\ \{\mathsf{VFold}*\} \\
\textbf{instance}\ \mathsf{VFold\ n\ c\ VNil\ n} \\
\ \textbf{where}\ \{\mathsf{VFold}\dagger\} \\
\textbf{instance}\ \mathsf{VFold\ n\ c}\qquad \mathsf{v}\qquad\qquad \mathsf{t}\qquad \Rightarrow \\
\qquad\quad \mathsf{VFold\ n\ c\ (VCons\ v)\ (c\ t)}\ \mathit{where}\ \{\mathsf{VFold}\ddagger\}
\end{array}
$$

$$
\begin{array}{ll}
\{\mathsf{VFold}*\} & \mathit{vFold} :: \mathsf{n\ b}\ \Rightarrow \\
 & \qquad (\textbf{forall}\ \mathsf{f.\ a}\ \rightarrow\ \mathsf{f\ b}\ \rightarrow\ \mathsf{c\ f\ b})\ \rightarrow \\
 & \qquad \mathsf{v\ a}\ \rightarrow\ \mathsf{t\ b} \\
\{\mathsf{VFold}\dagger\} & \mathit{vFold}\ n\ c\ \mathit{VNil}\ =\ n \\
\{\mathsf{VFold}\ddagger\} & \mathit{vFold}\ n\ c\ (\mathit{VCons}\ x\ xs)\ =\ c\ x\ (\mathit{vFold}\ n\ c\ xs)
\end{array}
$$

We can use *vFold* to define the 'append' operator for vectors. The structure of Vector is that of natural numbers—the type of *lengths*. As we append the vectors at the term level, we must add their lengths at the type level: *vFold* synchronizes the two.

$$
\begin{array}{l}
\mathit{vAppend} :: (\mathsf{Vector\ v},\ \mathsf{VFold\ v\ VCons\ u\ w})\ \rightarrow\ \mathsf{u\ a}\ \rightarrow\ \mathsf{v\ a}\ \rightarrow\ \mathsf{w\ a} \\
\mathit{vAppend}\ \mathit{us}\ \mathit{vs}\ =\ \mathit{vFold}\ \mathit{vs}\ \mathit{VCons}\ \mathit{us}
\end{array}
$$

The fact that the fold operators defined here have the *same* program but different *types* is a sure sign that the polymorphism available is too weak. Richard Bird and Ross Paterson come up against exactly the same problem when trying to define general fold operators for *nested* types (Bird & Paterson, 1999). We could write one truly general fold if only we would abstract over return types computed from Vector types and express the type-level functions we require—not just type constructors, but constant functions, recursively computed types and so on.

5.3 Vectors and Matrices via Nested Types?

No presentation of vectors and matrices could claim to be a proper appraisal of the issues involved without considering Chris Okasaki's deft and sophisticated treatment of these structures in (Okasaki, 1999). His vectors and matrices are defined using *nested* types, enforcing shape invariants such as *squareness* by typechecking. They are also cunningly optimised by using a *binary* representation of size, giving a logarithmic access time. However, I shall use a simplified *unary* presentation in order to focus on the expression of safety properties.

The key to the idea is to define a non-uniform type whose step cases are just embeddings which build tuple types and whose base case exploits the tuple in some way. We may define this structure by incorporating a little type-level 'continuation passing':

$$\begin{array}{l}\mathbf{data}\ \mathsf{WithTuple\ f\ t\ a}\ =\\ \qquad \mathit{Zero}\ (\mathsf{f\ t})\\ \ \ |\ \ \mathit{Suc}\ (\mathsf{WithTuple\ f\ (t, a)\ a})\end{array}$$

An element of $\mathsf{WithTuple\ f\ ()\ a}$ is effectively an inhabitant of $\mathsf{f\ a}^n$ wrapped up in a *header* which represents n as a unary number. The f can be thought of as a 'continuation'—what we do with the tuple type once we have constructed it. We can see vectors as tuples, so the identity function would be a suitable f: the closest Haskell allows is

$$\begin{array}{l}\mathbf{data}\ \mathsf{Id\ a}\ =\ \mathit{The}\ \mathsf{a}\\ \mathbf{type}\ \mathsf{Vector\ a}\ =\ \mathsf{WithTuple\ Id\ ()\ a}\end{array}$$

However, this type of vectors *conceals* their length. The most we can say is that

$$\mathsf{WithTuple\ Id\ a}^n\ \mathsf{a}$$

gives vectors of length *at least* n. We can thus achieve only limited precision by typechecking when attempting to relate vectors to other structures—we cannot demand a vector of length exactly 3, or that two vectors have the same length. That is, we cannot enforce shape invariants between elements of nested types, but only within them. If we want to define rectangular matrices as vectors of vectors, we must use the 'continuation' f to pass the outer vector structure inside:

$$\mathbf{type}\ \mathsf{Rect\ a}\ =\ \mathsf{WithTuple\ Vector\ ()\ a}$$

Once again, $\mathsf{Rect\ a}$ conceals the dimensions of the matrix within the header, which now represents a pair of numbers. We cannot express, say, the dimensional constraint *between* two matrices required to give a safe type to multiplication. Of course, we could use this continuation-passing style to define yet another data structure—pairs of matrices which can be multiplied—but we still cannot define a function on such pairs which makes explicit the dimensions of the resulting matrix, so that it can be used in further type-safe matrix operations.

It is the header data which codes for the precise structure of the information stored inside an element of a nested type: these codes are invisible at the type level. This makes nested types unsuitable as a basis for programming which is both type-safe and *compositional.* Okasaki may have given a type-safe definition of matrices which enforces their rectangularity, but his *projection* functions cannot enforce the required *bounds* on subscripting, and his operations on rectangular matrices cannot be used for his square matrices, which are necessarily of an *incompatible* type, rather than merely a more specific instance of an indexed family.

Our fake dependent families code their structure as type information, making it very easy to constrain them. Further, the data they contain is directly acessible, rather than buried under a header, allowing us to work in a straightforward first-order style.

6 Discussion

The techniques in this paper allow us to give types to many systematic operations which lie beyond the expressive power of parametric polymorphism, and to define classes of datatype, like the vectors, with systematically generated software. In this section, I discuss the limitations of this style of programming and make some suggestions as to how it might be better supported. I also discuss the practical implications of making a type system more complex, by whatever means.

6.1 The Trouble with Faking It

There are clearly practical difficulties with programs and data structures based on the type class mechanism. Horn clauses make a clumsy notation for functional programs, and the fragmentation of 'code' amongst datatype, class and instance declarations—whose complex interactions must be kept mutually consistent—is quite a burden for the programmer to bear. Further, at run-time, these programs are likely to put a much greater strain on the implementation of ad hoc polymorphism than it was ever intended to bear. Each polymorphic operation must be passed a *dictionary* containing the actual members it needs for particular instances, perforce including subdictionaries for the overloaded operations which get used in the process (Peterson & Jones, 1993): a large vector may well need a large dictionary.

The variety of datatype families which may be expressed by type classes is also quite limited. The type constructors in a class generate disjoint types in the family, and each term-level constructor targets only one type constructor. This is no problem if the term constructors naturally partition the family—we want *VNil* to make elements of a different type from *VCons*, so we can place them within separate VNil and VCons types. If we wanted to add vector concatenation as a constructor, it would need to target both. Such definitions are permitted in Lego, but we cannot have them in Haskell.

There is another way in which the simulation of dependent types described in this paper fails to measure up to the real thing: we cannot escape from the fact that our counterfeit type-level data and our actual term-level data are not interchangeable. Dependent datatypes like the 'lists of distinct elements' make essential use of the fact that the contents of a data structure can be used to restrict the types of operations over it. Here, we are forced to choose whether data belongs at compile-time in types, or at run-time in terms. The barrier represented by $::$ has not been broken, nor is it likely to be in the near future.

Even as things stand, though, there is one problem with quite serious implications: the incompatibility between different kinds of type-level function. Haskell's polymorphism allows us to abstract over type constructors, but we cannot abstract over type classes which happen to have a functional behaviour—indeed, it is not clear what such abstractions would mean with respect to member operations. This prevents us from defining a universal recursion operator for each counterfeit datatype, just as we cannot express them for nested types. As more support for *generic programming* is added to Haskell, there is a danger of still more incompatible notions of type-level function, unless we take steps to rationalise type-level computation.

6.2 What can be done?

There is much to commend the idea of dependent type systems, and the uniformity they offer between programming at term, type and kind levels. Indeed, systems like Luo's Extended Calculus of Constructions(Luo, 1994) provide an infinite hierarchy of levels, each syntactically alike, with one operational semantics for all, and with every type and program defined at lower levels made available for use higher up. Many kinds of genericity can be programmed directly by identifying a datatype code for the domain of types being addressed. Given a suitable type-level language, much needed extensions such as Tullsen's 'Zip Calculus' (Tullsen, 2000) and the 'polytypic' programming of Jansson, Jeuring, Hinze et al. (Jansson & Jeuring, 1997; Hinze & Jeuring, 2001) could be implemented as *libraries* for Haskell, rather than compiler extensions.

Nonetheless, I would be the first to acknowledge that the research in dependently typed programming has not yet matured sufficiently to be incorporated in Haskell. The designers of Haskell are right to protect its status as a well rationalised and solidly engineered vehicle for the best of functional programming technology. The potentially profound impact of greater interaction between terms and types should be carefully explored before Haskell allows the two to mix.

In the meantime, however, there is no reason why Haskell's type-level language should not move closer to a dependent system. Indeed, I am far from the first to suggest this: Peyton Jones and Meijer have already identified Pure Type Systems as a possible basis for rationalising the extensions to Haskell's type system at an intermediate compilation stage (Peyton Jones & Meijer, 1997). This paper is not the place for a formal proposal, but it is clear to me that the kind of programming I have illustrated in this paper could and should be presented directly, not by type class gymnastics.

The counterfeit datatypes introduced here via classes could be declared directly as inductive 'datakinds', in the same style as datatypes, but 'one level up'. The current notion of *type synonym* introduced by the **type** keyword could evolve to allow the definition of type-level programs over them in a conventional functional style, with kinds depending on type-level data just as the **forall** notation currently permits dependency on types. Such programs should be acceptable as parameters for operations and datatypes currently polymorphic only over type constructors: there is no reason why these parameters should be passed explicitly in cases where they can be inferred by Miller-style unification (Miller, 1992). Further, the current rigid separation of type and kind levels could be smoothed out and generalised by the introduction of a uniform hierarchy of levels, as found in ECC and seamlessly managed in Lego (Harper & Pollack, 1991).

These liberalisations would have three immediate consequences. Firstly, they would allow the definition of uniform recursion operators for type-level data and for term-level datatypes indexed over them. Secondly, domains of genericity currently implemented by ad hoc extensions of the type system could be rationalised by coding collections of types with type-level data: for example, a syntactic coding of the *regular* types is given in (McBride, 2001), and Pollack has given an extensive treatment of coding for extensible record types in (Pollack, 2000). Thirdly, a hierarchy uniformly capturing operations on types, kinds and beyond would enable layer on layer of generic programming at a single stroke: Hinze has already identified the need for 'polykinded' types to account for 'polytypic' programs (Hinze, 2000).

When research in programming with full dependent types delivers technology which is sufficiently stable to be incorporated in Haskell, dependency on type-level data can be replaced by dependency on its analogous term-level counterpart. We can readily combine the current drive to enhance Haskell's type system incrementally with sympathetic preparation for more radical change to come.

6.3 Must Type-Safe Programming be Difficult?

Many popular objections to dependent types stand just as well as objections to any complex type system with nontrivial interaction between levels. This paper provides the evidence that the current Haskell implementations have already reached this point. Type inference is no longer decidable in general, but Haskell's designers have sensibly maintained it for that fragment of the language where it is still possible. What we cannot do without is *typechecking*, and it is not unreasonable to demand that complex programs be given explicit signatures. On the other side of the Curry-Howard correspondence, we find very few mathematicians who write a proof without first stating the theorem.

Some suggest that even decidability of typechecking is not essential, but this is not the place to rehearse the arguments. The key to the decidability of typechecking, should we wish it, is ensuring that type-level programs *terminate*. The extended type class mechanism permits type-level general recursion, leaving Haskell's type system as undecidable as Cayenne's (Augustsson, 1998), and for exactly the same reason. However, we are free to design a type-level language with only *structural* recursion, leaving the term-level intact, retaining decidability, and allowing a vast and fascinating range of new programs.

On a more pragmatic level, complex type systems raise important issues for the practice of programming. Type- and term-level operations must be kept in step, and the more intricate the type, the harder this gets. However, when you search for the silver lining in this cloud, you discover that the cloud itself is made of silver.

More precise types constrain the search space for well-typed programs, which makes them easier to construct *interactively*. The need for type-directed programming tools is not peculiar to dependently typed languages—it addresses a problem which is inevitable no matter which route towards richer typing is adopted. Much of the required technology already exists, for example, in graphical proof editors like Agda (Coquand & Coquand, 1999). The reason it has evolved in the type theory community is that its necessity has been clearer for longer, simply for survival.

7 Conclusions

It is indeed a pleasure to see programming techniques previously only found in a dependently typed setting becoming available in Haskell. This paper, I hope, illustrates their potential for capturing both genericity and precision in typed functional programming. The facility for type-level computation via multi-parameter type classes with associated functional dependencies enables the exploration of both of these directions by programming within Haskell, rather than making ad hoc extensions to the compiler. Of course, the model of computation supplied by the type class mechanism is not designed to support the style of programming shown here, so we should not be surprised or disappointed to find that this unexpected application of the technology is, though encouraging, less effective than we might like.

Nonetheless, with the type class mechanism, Haskell has taken a significant step towards the kind of expressivity offered by dependent types. Accurate types and flexible programs have rightly been taken as achievable objectives for future work. At the same time, the type theory community's interest in and need for good *programming* technology is developing in earnest. I look forward to a fruitful synergy.

Acknowledgements

Many thanks to James McKinna, Randy Pollack, Zhaohui Luo, Magnus Carlsson and Peter Hancock for their encouragement and helpful comments on this work. This paper has also benefited greatly from the suggestions made by the editor, and by the anonymous referees. I would especially like to thank my colleague Paul Callaghan for his advice, and for pushing me towards Haskell in the first place.

References

Augustsson, Lennart. 1998 (Sept.). Cayenne—a language with dependent types. *ACM International Conference on Functional Programming*. ACM.

Backhouse, Roland, & Oliviera, José Nuno (eds). (2000). *Mathematics of Program Construction*. LNCS, vol. 1837. Springer-Verlag.

Bird, Richard, & Paterson, Ross. (1999). Generalised folds for nested datatypes. *Formal Aspects of Computing*, **11**(3).

Coquand, Catarina, & Coquand, Thierry. (1999). Structured Type Theory. *Workshop on Logical Frameworks and Metalanguages.*

Fridlender, Daniel, & Indrika, Mia. (2000). Do we need dependent types? *Journal of Functional Programming*, **10**(4), 409–415.

Harper, Robert, & Pollack, Robert. (1991). Type checking with universes. *Theoretical computer science*, **89**, 107–136.

Hinze, Ralf. (2000). Polytypic Values Possess Polykinded Types. *In:* (Backhouse & Oliviera, 2000).

Hinze, Ralf, & Jeuring, Johan. (2001). *Type-indexed datatypes.* In preparation.

Jansson, Patrik, & Jeuring, Johan. 1997 (Jan.). PolyP—a polytypic programming language extension. *Pages 470–482 of: Proceedings of popl '97.* ACM.

Jones, Mark P. (2000). Type Classes with Functional Dependencies. *Pages 230–244 of: Proceedings of the 9th European Symposium on Programming, esop 2000.* LNCS, vol. 1782. Springer-Verlag.

Jones, Mark P., & Peterson, John C. 1999 (September). *Hugs 98 User Manual.* Available from `http://www.cse.ogi.edu/PacSoft/projects/Hugs/pages/downloading.htm`.

Kaes, Stefan. (1988). Parametric overloading in polymorphic programming languages. *Pages 131–144 of: 15th ACM Symposium on Principles of Programming Languages.* ACM.

Luo, Z. (1994). *Computation and reasoning: A type theory for computer science.* OUP.

Luo, Zhaohui, & Pollack, Randy. 1992 (May). *LEGO Proof Development System: User's Manual.* Tech. rept. ECS-LFCS-92-211. Laboratory for Foundations of Computer Science, University of Edinburgh.

Martin-Löf, Per. (1971). *A theory of types.* manuscript.

McBride, Conor. (1999). *Dependently Typed Functional Programs and their Proofs.* Ph.D. thesis, University of Edinburgh.

McBride, Conor. (2001). *The Derivative of a Regular Type is its Type of One-Hole Contexts.* Submitted to LICS.

Miller, Dale. (1992). Unification under a mixed prefix. *Journal of symbolic computation*, **14**(4), 321–358.

Okasaki, Chris. (1999). From Fast Exponentiation to Square Matrices: An Adventure in Types. *ACM International Conference on Functional Programming99.*

Peterson, John, & Jones, Mark P. (1993). Implementing Type Casses. *Proceedings of ACM SIGPLAN Symposium on Programming Language Design and Implementation.* ACM SIGPLAN.

Peyton Jones, S., & Meijer, E. (1997). *Henk: a typed intermediate language.* ACM Workshop on Types in Compilation.

Peyton Jones, Simon, Jones, Mark, & Meijer, Erik. 1997 (June). Type classes: an exploration of the design space. *Proceedings of the Haskell Workshop.*

Pollack, Robert. (2000). Dependently Typed Records for Representing mathematical structure. Aagard, & Harrison (eds), *Theorem Proving in Higher Order Logics, TPHOLs 2000.* LNCS, vol. 1869. Springer-Verlag.

Tullsen, Mark. (2000). The Zip Calculus. *In:* (Backhouse & Oliviera, 2000).

Wadler, Philip, & Blott, Stephen. (1989). How to make ad-hoc polymorphism less ad hoc. *Pages 60–76 of: 16th ACM Symposium on Principles of Programming Languages.* ACM.