Number-parameterized types

Oleg Kiselyov
oleg@okmij.org

Fleet Numerical Meteorology and Oceanography Center, Monterey, CA 93943

Abstract. This paper describes practical programming with types parameterized by numbers: e.g., an array type parameterized by the array's size or a modular group type Z_n parameterized by the modulus. An attempt to add, for example, two integers of different moduli should result in a compile-time error with a clear error message. Number-parameterized types let the programmer capture more invariants through types and eliminate some run-time checks.

We review several encodings of the numeric parameter but concentrate on the phantom type representation of a sequence of *decimal* digits. The decimal encoding makes programming with number-parameterized types convenient and error messages more comprehensible. We implement arithmetic on decimal number-parameterized types, which lets us statically typecheck operations such as array concatenation.

Overall we demonstrate a practical dependent-type-like system that is just a Haskell library. The basics of the number-parameterized types are written in Haskell98.

Keywords: Haskell, number-parameterized types, type arithmetic, decimal types, type-directed programming.

1 Introduction

Discussions about types parameterized by values – especially types of arrays or finite groups parameterized by their size – reoccur every couple of months on functional programming languages newsgroups and mailing lists. The often expressed wish is to guarantee that, for example, we never attempt to add two vectors of different lengths. As one poster said [12], "This [feature] would be helpful in the crypto library where I end up having to either define new length Words all the time or using lists and losing the capability of ensuring I am manipulating lists of the same length." Number-parameterized types as other more expressive types let us tell the typechecker our intentions. The typechecker may then help us write the code correctly. Many errors (which are often trivial) can be detected at compile time. Furthermore, we no longer need to litter the code with array boundary match checks. The code therefore becomes more readable, reliable, and fast. Number-parameterized types when expressed in signatures also provide a better documentation of the code and let the invariants be checked across module boundaries.

In this paper, we develop realizations of number-parameterized types in Haskell that indeed have all the above advantages. The numeric parameter is specified in *decimal* rather than in binary, which makes types smaller and far easier to read. Type error messages also become more comprehensible. The programmer may write or the compiler can infer equality constraints (e.g., two argument vectors of a function must be of

the same size), arithmetic constraints (e.g., one vector must be larger by some amount), and inequality constraints (e.g., the size of the argument vector must be at least one). The violations of the constraints are detected at compile time. We can remove run-time tag checks in functions like `vhead`, which are statically assured to receive a non-empty vector.

Although we come close to the dependent-type programming, we do not extend either a compiler or the language. Our system is a regular Haskell library. In fact, the basic number-parameterized types can be implemented entirely in Haskell98. Advanced operations such as type arithmetic require commonly supported Haskell98 extensions to multi-parameter classes with functional dependencies and higher-ranked types.

Our running example is arrays parameterized over their size. The parameter of the vector type is therefore a non-negative integer number. For simplicity, all the vectors in the paper are indexed from zero. In addition to vector constructors and element accessors, we define a `zipWith`-like operation to map two vectors onto the third, element by element. An attempt to map vectors of different sizes should be reported as a type error. The typechecker will also guarantee that there is no attempt to allocate a vector of a negative size. In Section 6 we introduce operations `vhead`, `vtail` and `vappend` on number-parameterized vectors. The types of these operations exhibit arithmetic and inequality constraints.

The present paper describes several gradually more sophisticated number-parameterized Haskell libraries. We start by paraphrasing the approach by Chris Okasaki, who represents the size parameter of vectors in a sequence of data constructors. We then switch to the encoding of the size in a sequence of type constructors. The resulting types are phantom and impose no run-time overhead. Section 3 describes unary encoding of numerals in type constructors, Sections 4 and 5 discuss decimal encodings. Section 4 introduces a type representation for fixed-precision decimal numbers. Section 5 removes the limitation on the maximal size of representable numbers, at a cost of a more complex implementation and of replacing commas with unsightly dollars signs. The decimal encoding is extendible to other bases, e.g., 16 or 64. The latter can be used to develop practical realizations of number-parameterized cryptographically interesting groups.

Section 6 describes the first contribution of the paper. We develop addition and subtraction of “decimal types”, i.e., of the type constructor applications representing non-negative integers in decimal notation. The implementation is significantly different from that for more common unary numerals. Although decimal numerals are notably difficult to add, they make number-parameterized programming practical. We can now write arithmetic equality and inequality constraints on number-parameterized types.

Section 7 briefly describes working with number-parameterized types when the numeric parameter, and even its upper bound, are not known until run time. We show one, quite simple technique, which assures a static constraint by a run-time check – witnessing. The witnessing code, which must be trustworthy, is notably compact. The section uses the method of blending of static and dynamic assurances that was first described in [6].

Section 8 compares our approach with the phantom type programming in SML by Matthias Blume, with a practical dependent-type system of Hongwei Xi, with statically-

sized and generic arrays in Pascal and C, with the shape inference in array-oriented languages, and with C++ template meta-programming. Section 9 concludes.

2 Encoding the number parameter in data constructors

The first approach to vectors parameterized by their size encodes the size as a series of data constructors. This approach has been used extensively by Chris Okasaki. For example, in [9] he describes square matrixes whose dimensions can be proved equal at compile time. He digresses briefly to demonstrate vectors of statically known size. A similar technique has been described by McBride [8]. In this section, we develop a more naive encoding of the size through data constructors, for introduction and comparison with the encoding of the size via type constructors in the following sections.

Our representation of vectors of a statically checked size is reminiscent of the familiar representation of lists:

```
data List a = Nil | Cons a (List a)
```

`List a` is a recursive datatype. Lists of different sizes have the same recursive type. To make the types different (so that we can represent the size, too) we break the explicit recursion in the datatype declaration. We introduce two data constructors:

```
module UnaryDS where
data VZero a = VZero deriving Show

infixr 3 :+:
data Vecp tail a = a :+: (tail a) deriving Show
```

The constructor `VZero` represents a vector of a zero size. A value of the type `Vecp tail a` is a non-empty vector formed from an element of the type `a` and (a smaller vector) of the type `tail a`. We place our vectors into the class `Show` for expository purposes. Thus vectors holding one element have the type `Vecp VZero a`, vectors with two elements have the type `Vecp (Vecp VZero) a`, with three elements `Vecp (Vecp (Vecp VZero)) a`, etc. We should stress the separation of the shape type of a vector, `Vecp (Vecp VZero)` in the last example, from the type of vector elements. The shape type of a vector clearly encodes vector's size, as repeated applications of a type constructor `Vecp` to the type constructor `VZero`, i.e., as a Peano numeral. We have indeed designed a number-parameterized *type*.

To generically manipulate the family of differently-sized vectors, we define a class of polymorphic functions:

```
class Vec t where
    vlength::  t a -> Int
    vat::      t a -> Int -> a
    vzipWith:: (a->b->c) -> t a -> t b -> t c
```

The method `vlength` gives us the size of a vector; the method `vat` lets us retrieve a specific element, and the method `vzipWith` produces a vector by an element-by-element combination of two other vectors. We can use `vzipWith` to add two vectors elementwise. We must emphasize the type of `vzipWith`: the two argument vectors may hold elements of different types, but the vectors must have the same shape, i.e., size.

The implementation of the class `Vec` has only two instances:

```
instance Vec VZero where
    vlength  = const 0
    vat      = error "null array or index out of range"
    vzipWith f a b = VZero

instance (Vec tail) => Vec (Vecp tail) where
    vlength (_ :+: t) = 1 + vlength t
    vat (a :+: _)  0 = a
    vat (_ :+: ta) n = vat ta (n-1)
    vzipWith f (a :+: ta) (b :+: tb) =
               (f a b) :+: (vzipWith f ta tb)
```

The second instance makes it clear that a value of a type `Vecp tail a` is a vector `Vec` if and only if `tail a` is a vector `Vec`. Our vectors, instances of the class `Vec`, are recursively defined too. Unlike lists, our vectors reveal their sizes in their types.

That was the complete implementation of the number-parameterized vectors. We can now define a few sample vectors:

```
v3c = 'a' :+: 'b' :+: 'c' :+: VZero
v3i = 1 :+: 2 :+: 3 :+: VZero
v4i = 1 :+: 2 :+: 3 :+: 4 :+: VZero
```

and a few simple tests:

```
test1 = vlength v3c
test2 = [vat v3c 0, vat v3c 1, vat v3c 2]
```

We can load the code into a Haskell system and run the tests. Incidentally, we can ask the Haskell system to tell us the inferred type of a sample vector:

```
*UnaryDS> :t v3c
Vecp (Vecp (Vecp VZero)) Char
```

The inferred type indeed encodes the size of the vector as a Peano numeral. We can try more complex tests, of element-wise operations on two vectors:

```
test3 = vzipWith (\c i -> (toEnum $ fromEnum c + fromIntegral i)::Char)
                 v3c v3i
test4 = vzipWith (+) v3i v3i
*UnaryDS> test3
'b' :+: ('d' :+: ('f' :+: VZero))
```

In particular, `test3` demonstrates an operation on two vectors of the same shape but of different element types.

An attempt to add, by mistake, two vectors of different sizes is revealing:

```
test5 = vzipWith (+) v3i v4i

Couldn't match 'VZero' against 'Vecp VZero'
   Expected type: Vecp (Vecp (Vecp VZero)) a
   Inferred type: Vecp (Vecp (Vecp (Vecp VZero))) a1
In the third argument of 'vzipWith', namely 'v4i'
In the definition of 'test5': vzipWith (+) v3i v4i
```

We get a type error, with a clear error message (the quoted message, here and elsewhere in the paper, is by GHCi. The Hugs error message is essentially the same). The typechecker, at the compile time, has detected that the sizes of the vectors to add elementwise do not match. To be more precise, the sizes are off by one.

For vectors described in this section, the element access operation, `vat`, takes `O(n)` time where `n` is the size of the vector. Chris Okasaki [9] has designed more sophisticated number-parameterized vectors with element access time `O(log n)`. Although this is an improvement, the overhead of accessing an element adds up for many operations. Furthermore, the overhead of data constructors, `:+:` in our example, becomes noticeable for longer vectors. When we encode the size of a vector as a sequence of data constructors, the latter overhead cannot be eliminated.

Although we have achieved the separation of the shape type of a vector from the type of its elements, we did so at the expense of a sequence of data constructors, `:+:`, at the term level. These constructors add time and space overheads, which increase with the vector size. In the following sections we show more efficient representations for number-parameterized vectors. The structure of their type will still tell us the size of the vector; however there will be no corresponding term structure, and, therefore, no space overhead of storing it nor run-time overhead of traversing it.

3 Encoding the number parameter in type constructors, in unary

To improve the efficiency of number-parameterized vectors, we choose a better run-time representation: Haskell arrays. The code in the present section is in Haskell98.

```
module UnaryT (..elided..) where
import Data.Array
```

First, we need a type structure (an infinite family of types) to encode non-negative numbers. In the present section, we will use an unary encoding in the form of Peano numerals. The unary type encoding of integers belongs to programming folklore. It is also described in [2] in the context of a foreign-function interface library of SML.

```
data Zero = Zero
data Succ a = Succ a
```

That is, the term `Zero` of the type `Zero` represents the number 0. The term `(Succ (Succ Zero))` of the type `(Succ (Succ Zero))` encodes the number two. We call these numerals Peano numerals because the number `n` is represented as a repeated application of `n` type (data) constructors `Succ` to the type (term) `Zero`. We observe a one-to-one correspondence between the types of our numerals and the terms. In fact, a numeral term looks precisely the same as its type. This property is crucial as we shall see on many occasions below. It lets us "lift" number computations to the type level. The property also makes error messages lucid.[1]

We place our Peano numerals into a class `Card`, which has a method `c2num` to convert a numeral into the corresponding number.

```
class Card c where
    c2num:: (Num a) => c -> a -- convert to a number

cpred::(Succ c) -> c
cpred = undefined

instance Card Zero where
    c2num _ = 0
instance (Card c) => Card (Succ c) where
    c2num x = 1 + c2num (cpred x)
```

The function `cpred` determines the predecessor for a positive Peano numeral. The definition for that function may seem puzzling: it is undefined. We observe that the callers do not need the value returned by that function: they merely need the type of that value. Indeed, let us examine the definitions of the method `c2num` in the above two instances. In the instance `Card Zero`, we are certain that the argument of `c2num` has the type `Zero`. That type, in our encoding, represents the number zero, which we return. There can be only one non-bottom value of the type `Zero`: therefore, once we know the type, we do not need to examine the value. Likewise, in the instance `Card (Succ c)`, we know that the type of the argument of `c2num` is `(Succ c)`, where `c` is itself a `Card` numeral. If we could convert a value of the type `c` to a number, we can convert the value of the type `(Succ c)` as well. By induction we determine that `c2num` never examines the value of its argument. Indeed, not only `c2num (Succ (Succ Zero))` evaluates to 2, but so does `c2num (undefined::(Succ (Succ Zero)))`.

The same correspondence between the types and the terms suggests that the numeral type alone is enough to describe the size of a vector. We do not need to store the value of the numeral. The shape type of our vectors could be *phantom* [2].

```
newtype Vec size a = Vec (Array Int a) deriving Show
```

That is, the type variable `size` does not occur on the right-hand size of the `Vec` declaration. More importantly, at run-time our `Vec` is indistinguishable from an `Array`, thus

[1] We could have declared `Succ` as `newtype Succ a = Succ a` so that `Succ` is just a tag and all non-bottom Peano numerals share the same run-time representation. As we shall see however, we hardly ever use the values of our numerals.

incurring no additional overhead and providing constant-time element access. As we mentioned earlier, for simplicity, all the vectors in the paper are indexed from zero. The data constructor `Vec` is not exported from the module, so one has to use the following functions to construct vectors.

```
listVec':: (Card size) => size -> [a] -> Vec size a
listVec' size elems = Vec $ listArray (0,(c2num size)-1) elems

listVec:: (Card size) => size -> [a] -> Vec size a
listVec size elems | not (c2num size == length elems) =
               error "listVec: static/dynamic sizes mismatch"
listVec size elems = listVec' size elems

vec:: (Card size) => size -> a -> Vec size a
vec size elem = listVec' size $ repeat elem
```

The private function `listVec'` constructs the vector of the requested size initialized with the given values. The function makes no check that the length of the list of the initial values `elems` is equal to the length of the vector. We use this non-exported function internally, when we have proven that `elems` has the right length, or when truncating such a list is appropriate. The exported function `listVec` is a safe version of `listVec'`. The former assures that the constructed vector is consistently initialized. The function `vec` initializes all elements to the same value. For example, the following expression creates a boolean vector of two elements with the initial values `True` and `False`.

```
*UnaryT> listVec (Succ (Succ Zero)) [True,False]
Vec (array (0,1) [(0,True),(1,False)])
```

A Haskell interpreter created the requested value, and printed it out. We can confirm that the inferred type of the vector encodes its size:

```
*UnaryT> :type listVec (Succ (Succ Zero)) [True,False]
Vec (Succ (Succ Zero)) Bool
```

We can now introduce functions to operate on our vectors. The functions are similar to those in the previous section. As before, they are polymorphic in the shape of vectors (i.e., their sizes). This polymorphism is expressed differently however. In the present section we use just the parametric polymorphism rather than typeclasses.

```
vlength_t:: Vec size a -> size
vlength_t _ = undefined

vlength:: Vec size a -> Int
vlength (Vec arr) = let (0,last) = bounds arr in last+1

velems:: Vec size a -> [a]
velems (Vec v) = elems v
```

```
vat::  Vec size a -> Int -> a
vat (Vec arr) i = arr ! i
vzipWith:: Card size =>
   (a->b->c) -> Vec size a -> Vec size b -> Vec size c
vzipWith f va vb =
   listVec' (vlength_t va) $ zipWith f (velems va) (velems vb)
```

The functions `vlength_t` and `vlength` tell the size of their argument vector. The function `vat` returns the element of a vector at a given zero-based index. The function `velems`, which gives the list of vector's elements, is the left inverse of `listVec`. The function `vzipWith` elementwise combines two vectors into the third one by applying a user-specified function `f` to the corresponding elements of the argument vectors. The polymorphic types of these functions indicate that the functions generically operate on number-parameterized vectors of any `size`. Furthermore, the type of `vzipWith` expresses the constraint that the two argument vectors must have the same size. The result will be a vector of the same size as that of the argument vectors. We rely on the fact that the function `zipWith`, when applied to two lists of the same size, gives the list of that size. This justifies our use of `listVec'`.

We have introduced two functions that yield the size of their argument vector. One is the function `vlength_t`: it returns a value whose type represents the size of the vector. We are interested only in the type of the return value – which we extract statically from the type of the argument vector. The function `vlength_t` is a *compile-time* function. Therefore, it is no surprise that its body is `undefined`. The type of the function *is* its true definition. The function `vlength` in contrast retrieves vector's size from the run-time representation as an array. If we export `listVec` from the module `UnaryT` but do not export the constructor `Vec`, we can guarantee that `c2num . vlength_t` is equivalent to `vlength`: our number-parameterized vector type is sound.

From the practical point of view, passing terms such as `(Succ (Succ Zero))` to the functions `vec` or `listVec` to construct vectors is inconvenient. The previous section showed a better approach. We can implement it here too: we let the user enumerate the values, which we accumulate into a list, counting them at the same time:

```
infixl 3 &+
data VC size a = VC size [a]

vs:: VC Zero a; vs = VC Zero []
(&+):: VC size a -> a -> VC (Succ size) a
(&+) (VC size lst) x = VC (Succ size) (x:lst)
vc:: (Card size) => VC size a -> Vec size a
vc (VC size lst) = listVec' size (reverse lst)
```

The counting operation is effectively performed by a typechecker at compile time. Finally, the function `vc` will allocate and initialize the vector of the right size – and of the right type. Here are a few sample vectors and operations on them:

```
v3c = vc $ vs &+ 'a' &+ 'b' &+ 'c'
```

```
v3i = vc $ vs &+ 1 &+ 2 &+ 3
v4i = vc $ vs &+ 1 &+ 2 &+ 3 &+ 4

test1 = vlength v3c; test1' = vlength_t v3c
test2 = [vat v3c 0, vat v3c 1, vat v3c 2]
test3 = vzipWith (\c i -> (toEnum $ fromEnum c + fromIntegral i)::Char)
                 v3c v3i
test4 = vzipWith (+) v3i v3i
```

We can run the tests as follows:

```
*UnaryT> test3
Vec (array (0,2) [(0,'b'),(1,'d'),(2,'f')])
*UnaryT> :type test3
Vec (Succ (Succ (Succ Zero))) Char
```

The type of the result bears the clear indication of the size of the vector. If we attempt to perform an element-wise operation on vectors of different sizes, for example:

```
test5 = vzipWith (+) v3i v4i
Couldn't match 'Zero' against 'Succ Zero'
   Expected type: Vec (Succ (Succ (Succ Zero))) a
   Inferred type: Vec (Succ (Succ (Succ (Succ Zero)))) a1
In the third argument of 'vzipWith', namely 'v4i'
In the definition of 'test5': vzipWith (+) v3i v4i
```

we get a message from the typechecker that the sizes are off by one.

4 Fixed-precision decimal types

Peano numerals adequately represent the size of a vector in vector's type. However, they make the notation quite verbose. We want to offer a programmer a familiar, decimal notation for the terms and the types representing non-negative numerals. This turns out possible even in Haskell98. In this section, we describe a fixed-precision notation, assuming that a programmer will never need a vector with more than 999 elements. The limit is not hard and can be readily extended. The next section will eliminate the limit altogether.

We again will be using Haskell arrays as the run-time representation for our vectors. In fact, the implementation of vectors is the same as that in the previous section. The only change is the use of decimal rather than unary types to describe the sizes of our vectors.

```
module FixedDecT (..export list elided..) where
import Data.Array
```

Since we will be using the decimal notation, we need the terms and the types for all ten digits:

```
data D0 = D0
data D1 = D1
 ...
data D9 = D9
```

For clarity and to save space, we elide repetitive code fragments. The full code is available from [3]. To manipulate the digits uniformly (e.g., to find out the corresponding integer), we put them into a class `Digit`. We also introduce a class for non-zero digits. The latter has no methods: we use `NonZeroDigit` as a constraint on allowable digits.

```
class Digit d where     -- class of digits
    d2num:: (Num a) => d -> a   -- convert to a number

instance Digit D0 where d2num _ = 0
instance Digit D1 where d2num _ = 1
...
instance Digit D9 where d2num _ = 9

class Digit d => NonZeroDigit d
instance NonZeroDigit D1
instance NonZeroDigit D2
...
instance NonZeroDigit D9
```

We define a class of non-negative numerals. We make all single-digit numerals the members of that class:

```
class Card c where
    c2num:: (Num a) => c -> a -- convert to a number

-- Single-digit numbers are non-negative numbers
instance Card D0 where c2num _ = 0
instance Card D1 where c2num _ = 1
...
instance Card D9 where c2num _ = 9
```

We define a two-digit number, a tuple `(d1,d2)` where `d1` is a non-zero digit, a member of the class `Card`. The class `NonZeroDigit` makes expressing the constraint lucid. We also introduce three-digit decimal numerals `(d1,d2,d3)`:

```
instance (NonZeroDigit d1,Digit d2) => Card (d1,d2) where
    c2num c = 10*(d2num $ t12 c) + (d2num $ t22 c)

instance (NonZeroDigit d1,Digit d2,Digit d3) =>
         Card (d1,d2,d3) where
    c2num c = 100*(d2num $ t13 c) + 10*(d2num $ t23 c)
              + (d2num $ t33 c)
```

The instance constraints of the `Card` instances guarantee the uniqueness of our representation of numbers: the major decimal digit of a multi-digit number is not zero. It will be a type error to attempt to form such an number:

```
*FixedDecT> vec (D0,D1) 'a'
<interactive>:1:
    No instance for (NonZeroDigit D0)
```

The auxiliary compile-time functions `t12`...`t33` are tuple selectors. We could have avoided them in GHC with Glasgow extensions, which supports local type variables. We feel however that keeping the code Haskell98 justifies the extra hassle:

```
t12::(a,b)   -> a; t12 = undefined
t22::(a,b)   -> b; t22 = undefined
...
t33::(a,b,c) -> c; t33 = undefined
```

The rest of the code is as before, e.g.:

```
newtype Vec size a = Vec (Array Int a) deriving Show

listVec':: Card size => size -> [a] -> Vec size a
listVec' size elems = Vec $ listArray (0,(c2num size)-1) elems
```

The implementations of the polymorphic functions `listVec`, `vec`, `vlength_t`, `vlength`, `vat`, `velems`, and `vzipWith` are precisely the same as those in Section 3. We elide the code for the sake of space. We introduce a few sample vectors, using the decimal notation this time:

```
v12c = listVec (D1,D2) $ take 12 ['a'..'z']
v12i = listVec (D1,D2) [1..12]
v13i = listVec (D1,D3) [1..13]
```

The decimal notation is so much convenient. We can now define long vectors without pain. As before, the type of our vectors – the size part of the type – looks precisely the same as the corresponding size term expression:

```
*FixedDecT> :type v12c
Vec (D1, D2) Char
```

We can use the sample vectors in the tests like those of the previous section, [3]. If we attempt to elementwise add two vectors of different sizes, we get a type error:

```
test5 = vzipWith (+) v12i v13i

Couldn't match `D2' against `D3'
   Expected type: Vec (D1, D2) a
   Inferred type: Vec (D1, D3) a1
In the third argument of `vzipWith', namely `v13i'
In the definition of `test5': vzipWith (+) v12i v13i
```

The error message literally says that 12 is not equal to 13: the typechecker expected a vector of size 12 but found a vector of size 13 instead.

5 Arbitrary-precision decimal types

From the practical point of view, the fixed-precision number-parameterized vectors of the previous section are sufficient. The imposition of a limit on the width of the decimal numerals – however easily extended – is nevertheless intellectually unsatisfying. One may wish for an encoding of arbitrarily large decimal numbers within a framework that has been set up once and for all. Such an SML framework has been introduced in [2], to encode the sizes of arrays in their types. It is interesting to ask if such an encoding is possible in Haskell. The present section demonstrates a representation of arbitrary large decimal numbers in *Haskell98*. We also show that typeclasses in Haskell have made the encoding easier and precise: our decimal types are in bijection with non-negative integers. As before, we use the decimal types as phantom types describing the shape of number-parameterized vectors.

We start by defining the types for the ten digits:

```
module ArbPrecDecT (..export list elided..) where
import Data.Array

data D0 a = D0 a
data D1 a = D1 a
...
data D9 a = D9 a
```

Unlike the code in the previous section, `D0` through `D9` are type constructors of one argument. We use the composition of the constructors to represent sequences of digits. And so we introduce a class for arbitrary sequences of digits:

```
class Digits ds where
    ds2num:: (Num a) => ds -> a -> a
```

with a method to convert a sequence to the corresponding number. The method `ds2num` is designed in the accumulator-passing style: its second argument is the accumulator. We also need a type, which we call `Sz`, to represent an empty sequence of digits:

```
data Sz = Sz -- zero size (or the Nil of the sequence)
instance Digits Sz where
    ds2num _ acc = acc
```

We now inductively define arbitrarily long sequences of digits:

```
instance (Digits ds) => Digits (D0 ds) where
    ds2num dds acc = ds2num (t22 dds) (10*acc)
instance (Digits ds) => Digits (D1 ds) where
    ds2num dds acc = ds2num (t22 dds) (10*acc + 1)
```

```
...
instance (Digits ds) => Digits (D9 ds) where
    ds2num dds acc = ds2num (t22 dds) (10*acc + 9)

t22::(f x)   -> x; t22 = undefined
```

The type and the term `Sz` is an empty sequence; `D9 Sz` – that is, the application of the constructor `D9` to `Sz` – is a sequence of one digit, digit 9. The application of the constructor `D1` to the latter sequence gives us `D1 (D9 Sz)`, a two-digit sequence of digits one and nine. Compositions of data/type constructors indeed encode sequences of digits. As before, the terms and the types look precisely the same. The compositions can of course be arbitrarily long:

```
*ArbPrecDecT> :type D1$ D2$ D3$ D4$ D5$ D6$ D7$ D8$ D9$ D0$ D9$
D8$ D7$ D6$ D5$ D4$ D3$ D2$ D1$ Sz
D1 (D2 (D3 (D4 (D5 (D6 (D7 (D8 (D9 (D0 (D9 (D8 (D7
(D6 (D5 (D4 (D3 (D2 (D1 Sz))))))))))))))))))
*ArbPrecDecT> ds2num (D1$ D2$ D3$ D4$ D5$ D6$ D7$ D8$ D9$ D0$ D9$
D8$ D7$ D6$ D5$ D4$ D3$ D2$ D1$ Sz) 0
12345678909876543 21
```

We should point out a notable advantage of Haskell typeclasses in designing of sophisticated type families – in particular, in specifying constraints. Nothing prevents a programmer from using our type constructors, e.g., `D1`, in unintended ways. For example, a programmer may form a value of the type `D1 Bool`: either by applying a data constructor `D1` to a boolean value, or by casting a polymorphic value, `undefined`, into that type:

```
*ArbPrecDecT> :type D1 True
D1 Bool
*ArbPrecDecT> :type (undefined::D1 Bool)
D1 Bool
```

However, such types do *not* represent decimal sequences. Indeed, an attempt to pass either of these values to `ds2num` will result in a type error:

```
*ArbPrecDecT> ds2num (undefined::D1 Bool) 0
    No instance for (Digits Bool)
    arising from use of ‘ds2num’ at <interactive>:1
    In the definition of ‘it’: ds2num (undefined :: D1 Bool) 0
```

In contrast, the approach in [2] prevented the user from constructing (non-bottom) values of these types by a careful design and export of value constructors. That approach relied on SML's module system to preclude the overt mis-use of the decimal type system. Yet the user can still form a (latent, in SML) bottom value of the "bad" type, e.g., by attaching an appropriate type signature to an empty list, error function or other suitable polymorphic value. In a non-strict language like Haskell such values would make

our approach, which relies on phantom types, unsound. Fortunately, we are able to eliminate ill-formed decimal types at the type level rather than at the term level. Our class `Digits` admits those and *only* those types that represent sequences of digits.

To guarantee the bijection between non-negative numbers and sequences of digits, we need to impose an additional restriction: the first, i.e., the major, digit of a sequence must be non-zero. Expressing such a restriction is surprisingly straightforward in Haskell, even Haskell98.

```
class (Digits c) => Card c where
    c2num:: (Num a) => c -> a
    c2num c = ds2num c 0

instance Card Sz
instance (Digits ds) => Card (D1 ds)
instance (Digits ds) => Card (D2 ds)
...
instance (Digits ds) => Card (D9 ds)
```

As in the previous sections, the class `Card` represents non-negative integers. A non-negative integer is realized here as a sequence of decimal digits – provided, as the instances specify, that the sequence starts with a digit other than zero. We can now define the type of our number-parameterized vectors:

```
newtype Vec size a = Vec (Array Int a) deriving Show
```

which looks precisely as before, and polymorphic functions `vec`, `listVec`, `vlength_t`, `vlength`, `velems`, `vat`, and `vzipWith` – which are identical to those in Section 3. We can define a few sample vectors:

```
v12c = listVec (D1 $ D2 Sz) $ take 12 ['a'..'z']
v12i = listVec (D1 $ D2 Sz) [1..12]
v13i = listVec (D1 $ D3 Sz) [1..13]
```

we should note a slight change of notation compared to the corresponding vectors of Section 4. The tests are not changed and continue to work as before:

```
test4 = vzipWith (+) v12i v12i

*ArbPrecDecT> :type test4
Vec (D1 (D2 Sz)) Int
*ArbPrecDecT> test4
Vec (array (0,11) [(0,2),(1,4),(2,6),...(11,24)])
```

The compiler has been able to infer the size of the result of the `vzipWith` operation. The size is lucidly spelled in decimal in the type of the vector. Again, an attempt to elementwise add vectors of different sizes leads to a type error:

```
test5 = vzipWith (+) v12i v13i
Couldn't match 'D2 Sz' against 'D3 Sz'
   Expected type: Vec (D1 (D2 Sz)) a
   Inferred type: Vec (D1 (D3 Sz)) a1
In the third argument of 'vzipWith', namely 'v13i'
In the definition of 'test5': vzipWith (+) v12i v13i
```

The typechecker complains that 2 is not equal to 3: it found the vector of size 13 whereas it expected a vector of size 12. The decimal types make the error message very clear.

We must again point out a significant difference of our approach from that of [2]. We were able to state that only those types of digital sequences that start with a non-zero digit correspond to a non-negative number. SML, as acknowledged in [2], is unable to express such a restriction directly. The paper [2], therefore, prevents the user from building invalid decimal sequences by relying on the module system: by exporting carefully-designed value constructors. The latter use an auxiliary phantom type to keep track of "nonzeroness" of the major digit. Our approach does not incur such a complication. Furthermore, by the very inductive construction of the classes `Digits` and `Card`, there is a one-to-one correspondence between *types*, the members of `Card`, and the integers in decimal notation. In [2], the similar mapping holds only when the family of decimal types is restricted to the types that correspond to constructible values. A user of that system may still form bottom values of invalid decimal types, which will cause run-time errors. In our case, when the digit constructors are misapplied, the result will no longer be in the class `Card`, and so the error will be detected *statically* by the typechecker:

```
*ArbPrecDecT> vec (D1$ D0$ D0$ True) 0
    No instance for (Digits Bool)
    arising from use of 'vec' at <interactive>:1
    In the definition of 'it': vec (D1 $ (D0 $ (D0 $ True))) 0

*ArbPrecDecT> vec (D0$ D1$ D0 Sz) 0
    No instance for (Card (D0 (D1 (D0 Sz))))
    arising from use of 'vec' at <interactive>:1
    In the definition of 'it': vec (D0 $ (D1 $ (D0 Sz))) 0
```

6 Computations with decimal types

The previous sections gave many examples of functions such as `vzipWith` that take two vectors *statically* known to be of equal size. The signature of these functions states quite detailed invariants whose violations will be reported at compile-time. Furthermore, the invariants can be inferred by the compiler itself. This use of the type system is not particular to Haskell: Matthias Blume [2] has derived a similar parameterization of arrays in SML, which can express such equality of size constraints. Matthias Blume however cautions one not to overstate the usefulness of the approach because the type system can express only fairly simple constraints: "There is still no type that, for example, would force two otherwise arbitrary arrays to differ in size by exactly one." That was written in

the context of SML however. In Haskell with common extensions we *can* define vector functions whose type contains arithmetic constraints on the sizes of the argument and the result vectors. These constraints can be verified statically and sometimes even inferred by a compiler. In this section, we consider the example of vector concatenation. We shall see that the inferred type of `vappend` manifestly affirms that the size of the result is the sum of the sizes of two argument vectors. We also introduce the functions `vhead` and `vtail`, whose type specifies that they can only be applied to non-empty vectors. Furthermore, the type of `vtail` says that the size of the result vector is less by one than the size of the argument vector. These examples are quite unusual and almost cross into the realm of dependent types.

We must note however that the examples in this section require the Haskell98 extension to multi-parameter classes with functional dependencies. That extension is activated by flags `-98` of Hugs and `-fglasgow-exts -fallow-undecidable-instances` of GHCi.

We will be using the arbitrary precision decimal types introduced in the previous section. We aim to design a 'type addition' of decimal sequences. Our decimal types spell the corresponding non-negative numbers in the conventional (i.e., big-endian) decimal notation: the most-significant digit first. However, it is more convenient to add such numbers starting from the least-significant digit. Therefore, we need a way to reverse digital sequences, or more precise, types of the class `Digits`. We use the conventional sequence reversal algorithm written in the accumulator-passing style.

```
class DigitsInReverse' df w dr | df w -> dr

instance DigitsInReverse' Sz acc acc
instance (Digits (d drest), DigitsInReverse' drest (d acc) dr)
          => DigitsInReverse' (d drest) acc dr
```

We introduced the class `DigitsInReverse' df w dr` where `df` is the source sequence, `dr` is the reversed sequence, and `w` is the accumulator. The three digit sequence types belong to `DigitsInReverse'` if the reverse of `df` appended to `w` gives the digit sequence `dr`. The functional dependency and the two instances spell this constraint out. We can now introduce a class that relates a sequence of digits with its reverse:

```
class DigitsInReverse df dr | df -> dr, dr -> df

instance (DigitsInReverse' df Sz dr, DigitsInReverse' dr Sz df)
         => DigitsInReverse df dr
```

Two sequences of digits `df` and `dr` belong to the class `DigitsInReverse` if they are the reverse of each other. The functional dependencies make the "each other" part clear: one sequence uniquely determines the other. The typechecker will verify that given `df`, it can find `dr` so that both `DigitsInReverse' df Sz dr` and `DigitsInReverse' dr Sz df` are satisfied. To test the reversal process, we define a function `digits_rev`:

```
digits_rev:: (Digits ds, Digits dsr, DigitsInReverse ds dsr)
             => ds -> dsr
digits_rev = undefined
```

It is again a compile-time function specified entirely by its type. Its body is therefore undefined. We can now run a few examples:

```
*ArbArithmT> :t digits_rev (D1$D2$D3 Sz)
D3 (D2 (D1 Sz))
*ArbArithmT> :t (\v -> digits_rev v `asTypeOf` (D1$D2$D3 Sz))
D3 (D2 (D1 Sz)) -> D1 (D2 (D3 Sz))
```

Indeed, the process of reversing sequences of decimal digits works both ways. Given the type of the argument to `digits_rev`, the compiler infers the type of the result. Conversely, given the type of the result the compiler infers the type of the argument.

A sequence of digits belongs to the class `Card` only if the most-significant digit is not a zero. To convert an arbitrary sequence to `Card` we need a way to strip leading zeros:

```
class NoLeadingZeros d d0 | d -> d0
instance NoLeadingZeros Sz Sz
instance (NoLeadingZeros d d') => NoLeadingZeros (D0 d) d'
instance NoLeadingZeros (D1 d) (D1 d)
...
instance NoLeadingZeros (D9 d) (D9 d)
```

We are now ready to build the addition machinery. We draw our inspiration from the computer architecture: the adder of an arithmetical-logical unit (ALU) of the CPU is constructed by chaining of so-called full-adders. A full-adder takes two summands and the carry-in and yields the result of the summation and the carry-out. In our case, the summands and the result are decimal rather than binary. Carry is still binary.

```
class FullAdder d1 d2 cin dr cout
      | d1 d2 cin -> cout, d1 d2 cin -> dr,
        d1 dr cin -> cout, d1 dr cin -> d2
  where
    _unused:: (d1 xd1) -> (d2 xd2) -> cin -> (dr xdr)
    _unused = undefined
```

The class `FullAdder` establishes a relation among three digits `d1`, `d2`, and `dr` and two carry bits `cin` and `cout`: `d1 + d2 + cin = dr + 10*cout`. The digits are represented by the type constructors `D0` through `D9`. The sole purpose of the method `_unused` is to cue the compiler that `d1`, `d2`, and `dr` are type constructors. The functional dependencies of the class tell us that the summands and the input carry uniquely determine the result digit and the output carry. On the other hand, if we know the result digit, one of the summands, `d1`, and the input carry, we can determine the other summand. The same relation `FullAdder` can therefore be used for addition and for subtraction. In the latter case, the carry bits should be more properly called borrow bits.

```
data Carry0
data Carry1
```

```
instance FullAdder D0 D0 Carry0 D0 Carry0
instance FullAdder D0 D0 Carry1 D1 Carry0
instance FullAdder D0 D1 Carry0 D1 Carry0
...
instance FullAdder D9 D8 Carry1 D8 Carry1
instance FullAdder D9 D9 Carry0 D8 Carry1
instance FullAdder D9 D9 Carry1 D9 Carry1
```

The full code [3] indeed contains 200 instances of `FullAdder`. The exhaustive enumeration verifies the functional dependencies of the class. The number of instances could be significantly reduced if we availed ourselves to an overlapping instances extension. For generality however we tried to use as few Haskell98 extensions as possible. Although 200 instances seems like quite many, we have to write them only once. We place the instances into a module and separately compile it. Furthermore, we did not write those instances by hand: we used Haskell itself:

```
make_full_adder
    = mapM_ putStrLn
            [unwords $ doit d1 d2 cin | d1<-[0..9],
                                        d2<-[0..9], cin<-[0..1]]
  where
       doit d1 d2 cin
          = ["instance FullAdder", tod d1, tod d2, toc cin,
             tod d12, toc cout]
         where
             (d12,cout) = let sum = d1 + d2 + cin
                          in if sum >= 10 then (sum-10,1) else (sum,0)
       tod n | (n >= 0 && 9 >= n) = "D" ++ (show n)
       toc 0 = "Carry0"; toc 1 = "Carry1"
```

That function is ready for Template Haskell. Currently we used a low-tech approach of cutting and pasting from an Emacs buffer with GHCi into the Emacs buffer with the code.

We use `FullAdder` to build the full adder of two little-endian decimal sequences `ds1` and `ds2`. The relation `DigitsSum ds1 ds2 cin dsr` holds if `ds1+ds2+cin = dsr`. We add the digits from the least significant onwards, and we propagate the carry. If one input sequence turns out shorter than the other, we pad it with zeros. The correctness of the algorithm follows by simple induction.

```
class DigitsSum ds1 ds2 cin dsr | ds1 ds2 cin -> dsr
instance DigitsSum Sz Sz Carry0 Sz
instance DigitsSum Sz Sz Carry1 (D1 Sz)
instance (DigitsSum (D0 Sz) (d2 d2rest) cin (d12 d12rest)) =>
         DigitsSum Sz (d2 d2rest) cin (d12 d12rest)
instance (DigitsSum (d1 d1rest) (D0 Sz) cin (d12 d12rest)) =>
         DigitsSum (d1 d1rest) Sz cin (d12 d12rest)
```

```
instance (FullAdder d1 d2 cin d12 cout,
          DigitsSum d1rest d2rest cout d12rest) =>
   DigitsSum (d1 d1rest) (d2 d2rest) cin (d12 d12rest)
```

We also need the inverse relation: `DigitsDif ds1 ds2 cin dsr` holds on precisely the same condition as `DigitsSum`. Now, however, the sequences `ds1`, `dsr` and the input carry `cin` determine one of the summands, `ds2`. The input carry actually means the input borrow bit. The relation `DigitsDif` is defined only if the output sequence `dsr` has at least as many digits as `ds1` — which is the necessary condition for the result of the subtraction to be non-negative.

```
class DigitsDif ds1 ds2 cin dsr | ds1 dsr cin -> ds2
instance DigitsDif Sz ds Carry0 ds
instance (DigitsDif (D0 Sz) (d2 d2rest) Carry1 (d12 d12rest)) =>
         DigitsDif Sz (d2 d2rest) Carry1 (d12 d12rest)
instance (FullAdder d1 d2 cin d12 cout,
          DigitsDif d1rest d2rest cout d12rest) =>
   DigitsDif (d1 d1rest) (d2 d2rest) cin (d12 d12rest)
```

The class `CardSum` with a single instance puts it all together:

```
class (Card c1, Card c2, Card c12) =>
    CardSum c1 c2 c12 | c1 c2 -> c12, c1 c12 -> c2
instance (Card c1, Card c2, Card c12,
          DigitsInReverse c1 c1r,
          DigitsInReverse c2 c2r,
          DigitsSum c1r c2r Carry0 c12r,
          DigitsDif c1r c2r' Carry0 c12r,
          DigitsInReverse c2r' c2', NoLeadingZeros c2' c2,
          DigitsInReverse c12r c12)
          => CardSum c1 c2 c12
```

The class establishes the relation between three `Card` sequences `c1`, `c2`, and `c12` such that the latter is the sum of the formers. The two summands determine the sum, or the sum and one summand determine the other. The class can be used for addition and subtraction of sequences. The dependencies of the sole `CardSum` instance spell out the algorithm. We reverse the summand sequences to make them little-endian, add them together with the zero carry, and reverse the result. We also make sure that the subtraction and summation are the exact inverses. The addition algorithm `DigitsSum` never produces a sequence with the major digit zero. The subtraction algorithm however may result in a sequence with zero major digits, which have to be stripped away, with the help of the relation `NoLeadingZeros`. We introduce a compile-time function `card_sum` so we can try the addition out:

```
card_sum:: CardSum c1 c2 c12 => c1 -> c2 -> c12
card_sum = undefined
```

```
*ArbArithmT> :t card_sum (D1 Sz) (D9$D9 Sz)
D1 (D0 (D0 Sz))
*ArbArithmT> :t \v -> card_sum (D1 Sz) v `asTypeOf` (D1$D0$D0 Sz)
D9 (D9 Sz) -> D1 (D0 (D0 Sz))
*ArbArithmT> :t \v -> card_sum (D9$D9 Sz) v `asTypeOf` (D1$D0$D0 Sz)
D1 Sz -> D1 (D0 (D0 Sz))
```

The typechecker can indeed add and subtract with carry and borrow. Now we define the function `vappend` to concatenate two vectors.

```
vappend va vb = listVec (card_sum (vlength_t va) (vlength_t vb))
                $ (velems va) ++ (velems vb)
```

We could have used the function `listVec'`; for illustration, we chose however to perform a run-time check and avoid proving the theorem about the size of the list concatenation result. We did not declare the type of `vappend`; still the compiler is able to infer it:

```
*ArbArithmT> :t vappend
vappend :: (CardSum size size1 c12) =>
           Vec size a -> Vec size1 a -> Vec c12 a
```

which literally says that the size of the result vector is the sum of the sizes of the argument vectors. The constraint is spelled out patently, as part of the type of `vappend`. The sizes may be arbitrarily large decimal numbers: for example, the following expression demonstrates the concatenation of a vector of 25 elements and a vector of size 979:

```
*ArbArithmT> :t vappend (vec (D2$D5 Sz) 0) (vec (D9$D7$D9 Sz) 0)
(Num a) => Vec (D1 (D0 (D0 (D4 Sz)))) a
```

We introduce the deconstructor functions `vhead` and `vtail`. The type of the latter is exactly what was listed in [2] as an unattainable wish.

```
vhead:: CardSum (D1 Sz) size1 size => Vec size a -> Vec (D1 Sz) a
vhead va = listVec (D1 Sz) $ [head (velems va)]
vtail:: CardSum (D1 Sz) size1 size => Vec size a -> Vec size1 a
vtail va = result
  where result = listVec (vlength_t result) $ tail (velems va)
```

Although the body of `vtail` seem to refer to that function result, the function is not divergent and not recursive. Recall that `vlength_t` is a compile-time, 'type' function. Therefore the body of `vtail` refers to the statically known type of `result` rather than to its value. The type of `vhead` is also noteworthy: it essentially specifies an *inequality* constraint: the input vector is non-empty. The constraint is expressed via an implicitly existentially quantified variable `size1`: the type of `vhead` says that there must exist a non-negative number `size1` such that incrementing it by one should give the size of the input vector.

We can now run a few examples. We note that the compiler could correctly infer the type of the result, which includes the size of the vector after appending or truncating it.

```
*ArbArithmT> let v = vappend (vec (D9 Sz) 0) (vec (D1 Sz) 1)
*ArbArithmT> :t v
Vec (D1 (D0 Sz)) Integer
*ArbArithmT> v
Vec (array (0,9) [(0,0),(1,0),...,(8,0),(9,1)])
*ArbArithmT> :type vhead v
Vec (D1 Sz) Integer
*ArbArithmT> :type vtail v
Vec (D9 Sz) Integer
*ArbArithmT> vtail v
Vec (array (0,8) [(0,0),(1,0),...,(7,0),(8,1)])
*ArbArithmT> :type (vappend (vhead v) (vtail v))
Vec (D1 (D0 Sz)) Integer
```

The types of `vhead` and `vtail` embed a non-empty argument vector constraint. Indeed, an attempt to apply `vhead` to an empty vector results in a type error:

```
*ArbArithmT> vtail (vec Sz 0)
<interactive>:1:0:
    No instances for (DigitsInReverse' c2' Sz c2r',
                      DigitsInReverse' c2r' Sz c2',
                      DigitsDif (D1 Sz) c2r' Carry0 Sz,
                      DigitsSum (D1 Sz) c2r Carry0 Sz,
                      DigitsInReverse' c2r Sz size1,
                      DigitsInReverse' size1 Sz c2r)
      arising from use of 'vtail' at <interactive>:1:0-4
```

The error message essentially says that there is no such decimal type `c2r` such that `DigitsSum (D1 Sz) c2r Carry0 Sz` holds. That is, there is no non-negative number that gives zero if added to one.

We can form quite complex expressions from the functions `vappend`, `vhead`, and `vtail`, and the compiler will *infer* and verify the corresponding constraints on the sizes of involved vectors. For example:

```
testc1 =
  let va = vec (D1$D2 Sz) 0
      vb = vec (D5 Sz) 1
      vc = vec (D8 Sz) 2
  in vzipWith (+) va (vappend vb (vtail vc))
*ArbArithmT> testc1
Vec (array (0,11) [(0,1),...,(4,1),(5,2),(6,2),...,(11,2)])
```

The size of the vector `va` must be the sum of the sizes of `vb` and `vc` minus one. Furthermore, the vector `vc` must be non-empty. The compiler has inferred this non-trivial constraint and checked it. Indeed, if we by mistake write `vc = vec (D9 Sz) 2`, as we actually did when writing the example, the compiler will instantly report a type error:

```
Couldn't match 'D9 Sz' against 'D8 Sz'
   Expected type: D9 Sz
   Inferred type: D8 Sz
When using functional dependencies to combine
   DigitsSum (D1 Sz) c2r Carry0 (D9 Sz),
     arising from use of 'vtail' at ArbArithmT.hs:420:34-38
   DigitsSum (D1 Sz) c2r Carry0 (D8 Sz),
     arising from use of 'vtail' at ArbArithmT.hs:411:34-38
```

The result `12 - 5 + 1` is expected to be 8 rather than 9.

We can define other operations that extend or shrink our vectors. For example, Section 3 introduced the operator `&+` to make the entering of vectors easier. It is straightforward to implement such an operator for decimally-typed vectors.

We must point out that the type system guarantees that `vhead` and `vtail` are applied to non-empty vectors. Therefore, we no longer need the corresponding run-time check. The bodies of `vhead` and `vtail` may *safely* use unsafe versions of the library functions `head` and `tail`, and hence increase the performance of the code without compromising its safety.

7 Statically-sized vectors in a dynamic context

In the present version of the paper, we demonstrate the simplest method of handling number-parameterized vectors in the dynamic context. The method involves run-time checks. The successful result of a run-time check is marked with the appropriate static type. Further computations can therefore rely on the result of the check (e.g., that the vector in question definitely has a particular size) and avoid the need to do that test over and over again. The net advantage is the reduction in the number of run-time checks. The complete elimination of the run-time checks is quite difficult (in general, may not even be possible) and ultimately requires a dependent type system.

For our presentation we use an example of dynamically-sized vectors: reversing a vector by the familiar accumulator-passing algorithm. Each iteration splits the source vector into the head and the tail, and prepends the head to the accumulator. The sizes of the vectors change in the course of the computation, to be precise, on each iteration. We treat vectors as if they were lists. Most of the vector processing code does not have such a degree of variation in vector sizes. The code is quite simple:

```
vreverse v = listVec (vlength_t v) $ reverse $ velems v
```

whose inferred type is obviously

```
*ArbArithmT> :t vreverse
vreverse :: (Card size) => Vec size a -> Vec size a
```

The use of `listVec` implies a dynamic test – as a witness to 'acquire' the static type `size`, the size type of the input vector. We do this test only once, at the conclusion of the algorithm. We can treat the result as any other number-parameterized vector, for example:

```
testv = let v  = vappend (vec (D3 Sz) 1) (vec (D1 Sz) 4)
            vr = vreverse v
        in vhead (vtail (vtail vr))
```

using the versions of `vhead` and `vtail` without any further run-time size checks.

8 Related work

This paper was inspired by Matthias Blume's messages on the newsgroup comp.lang.functional in February 2002. Many ideas of this paper were first developed during the USENET discussion, and posted in a series of three messages at that time. In more detail Matthias Blume described his method in [2], although that paper uses binary rather than decimal types of array sizes for clarity. The approaches by Matthias Blume and ours both rely on phantom types to encode additional information about a value (e.g., the size of an array) in a manner suitable for a typechecker. The paper [2] exhibits the most pervasive and thorough use of phantom types: to represent the size of arrays and the constness of imported C values, to encode C structure tag *names* and C function prototypes.

However, the paper [2] was written in the context of SML, whereas we use Haskell. The language has greatly influenced the method of specifying and enforcing complex static constraints, e.g., that digit sequences representing non-negative numbers must not have leading zeros. The SML approach in [2] relies on the sophisticated module system of SML to restrict the availability of value constructors so that users cannot build values of outlawed types. Haskell typeclasses on the other hand can directly express the constraint, as we saw in Section 5. Furthermore, Haskell typeclasses let us specify arithmetic equality and inequality constraints – which, as admitted in [2], seems quite unlikely to be possible in SML.

Arrays of a statically known size – whose size is a part of their type – are a fairly popular feature in programming languages. Such arrays are present in Fortran, Pascal, C [2]. Pascal has the most complete realization of statically sized arrays. A Pascal compiler can therefore typecheck array functions like our `vzipWith`. Statically sized arrays also contribute to expressiveness and efficiency: for example, in Pascal we can copy one instance of an array into another instance of the same type by a single assignment, which, for small arrays, can be fully inlined by the compiler into a sequential code with no loops or range checks. However, in a language without the parametric polymorphism statically sized arrays are a great nuisance. If the size of an array is a part of its type, we cannot write generic functions that operate on arrays of any size. We can only write functions dealing with arrays of specific, fixed sizes. The inability to build generic array-processing libraries is one of the most serious drawbacks of Pascal. Therefore, Fortran and C introduce "generic" arrays whose size type is not statically known. The compiler silently converts a statically-sized array into a generic one when passing arrays as arguments to functions. We can now build generic array-processing

[2] C does permit truly statically-sized arrays like those in Pascal. To achieve this, we should make a C array a member of a C structure. The compiler preserves the array size information when passing such a wrapped array as an argument. It is even possible to assign such "arrays".

libraries. We still need to know the size of the array. In Fortran and C, the programmer must arrange for passing the size information to a function in some other way, e.g., via an additional argument, global variable, etc. It becomes then the responsibility of a programmer to make sure that the size information is correct. The large number of Internet security advisories related to buffer overflows and other array-management issues testify that programmers in general are not to be relied upon for correctly passing and using the array size information. Furthermore, the silent, irreversible conversion of statically sized arrays into generic ones negate all the benefits of the former.

A different approach to array processing is a so-called shape-invariant programming, which is a key feature of array-oriented languages such as APL or SaC [11]. These languages let a programmer define operations that can be applied to arrays of arbitrary shape/dimensionality. The code becomes shorter and free from explicit iterations, and thus more reusable, easier to read and to write. The exact shape of an array has to be known, eventually. Determining it at run-time is greatly inefficient. Therefore, high-performance array-oriented languages employ shape inference [10], which tries to statically infer the dimensionalities or even exact sizes of all arrays in a program. Shape inference is, in general, undecidable, since arrays may be dynamically allocated. Therefore, one can either restrict the class of acceptable shape-invariant programs to a decidable subset, resort to a dependent-type language like Cayenne [1], or use "soft typing." The latter approach is described in [10], which introduces a non-unique type system based on a hierarchy of array types: from fully specialized ones with the statically known sizes and dimensionality, to a type of an array with the known dimensionality but not size, to a fully generic array type whose shape can only be determined at run-time. The system remains decidable because at any time the typechecker can throw up hands and give to a value a fully generic array type. Shape inference of SaC is specific to that language, whose type system is otherwise deliberately constrained: SaC lacks parametric polymorphism and higher-order functions. Using shape inference for compilation of shape-invariant array operations into a highly efficient code is presented in [7]. Their compiler tries to generate as precise shape-specific code as possible. When the shape inference fails to give the exact sizes or dimensionalities, the compiler emits code for a dynamic shape dispatch and generic loops.

There is however a great difference in goals and implementation between the shape inference of SaC and our approach. The former aims at accepting more programs than can statically be inferred shape-correct. We strive to express assertions about the array sizes and enforcing the programming style that assures them. We have shown the definitions of functions such as `vzipWith` whose the argument and the result vectors are all of the same size. This constraint is assured at compile-time – even if we do not statically know the exact sizes of the vectors. Because SaC lacks parametric polymorphism, it cannot express such an assertion and statically verify it. If a SaC programmer applies a function such as `vzipWith` to vectors of unequal size, the compiler will not flag that as an error but will compile a generic array code instead. The error will be raised at run time during a range check.

The approach of the present paper comes close to emulating a dependent type system, of which Cayenne [1] is the epitome. We were particularly influenced by a practical dependent type system of Hongwei Xi [14] [15], which is a conservative extension of

SML. In [14], Hongwei Xi et al. demonstrated an application of their system to the elimination of array bound checking and list tag checking. The related work section of that paper lists a number of other dependent and pseudo-dependent type systems. Using the type system to avoid unnecessary run-time checks is a goal of the present paper too.

C++ templates provide parametric polymorphism and indexing of types by true integers. A C++ programmer can therefore define functions like `vzipWith` and `vtail` with equality and even arithmetic constraints on the sizes of the argument vectors. Blitz++ [13] was the first example of using a so-called template meta-programming for generating efficient and safe array code. The type system of C++ however presents innumerable hurdles to the functional style. For example, the result type of a function is not used for the overloading resolution, which significantly restricts the power of the type inference. Templates were introduced in C++ ad hoc, and therefore, are not well integrated with its type system. Violations of static constraints expressed via templates result in error messages so voluminous as to become incomprehensible.

McBride [8] gives an extensive survey of the emulation of dependent type systems in Haskell. He also describes number-parameterized arrays that are similar to the ones discussed in Section 2. The paper by Fridlender and Indrika [4] shows another example of emulating dependent types within the Hindley-Milner type system: namely, emulating variable-arity functions such as generic `zipWith`. Their technique relies on ad hoc codings for natural numbers which resemble Peano numerals. They aim at defining more functions (i.e., multi-variate functions), whereas we are concerned with making functions more restrictive by expressing sophisticated invariants in functions' types. Another approach to multivariate functions – multivariate composition operator – is discussed in [5].

9 Conclusions

Throughout this paper we have demonstrated several realizations of number-parameterized types in Haskell, using arrays parameterized by their size as an example. We have concentrated on techniques that rely on phantom types to encode the size information in the type of the array value. We have built a family of infinite types so that different values of the vector size can have their own distinct type. That type is a decimal encoding of the corresponding integer (rather than the more common unary, Peano-like encoding). The examples throughout the paper illustrate that the decimal notation for the number-parameterized vectors makes our approach practical.

We have used the phantom size types to express non-trivial constraints on the sizes of the argument and the result arrays in the type of functions. The constraints include the size equality, e.g., the type of a function of two arguments may indicate that the arguments must be vectors of the same size. More importantly, we can specify arithmetical constraints: e.g., that the size of the vector after concatenation is the sum of the source vector sizes. Furthermore, we can write inequality constraints by means of an implicit existential quantification, e.g., the function `vhead` must be applied to a non-empty vector. The programmer should benefit from more expressive function signatures and from the ability of the compiler to statically check complex invariants in all applications of the vector-processing functions. The compiler indeed infers and checks non-trivial

constraints involving addition and subtraction of sizes – and presents readable error messages on violation of the constraints.

References

1. Augustsson, L. Cayenne – a language with dependent types. Proc. ACM SIGPLAN International Conference on Functional Programming, pp. 239–250, 1998.
2. Matthias Blume: No-Longer-Foreign: Teaching an ML compiler to speak C "natively." In BABEL'01: First workshop on multi-language infrastructure and interoperability, September 2001, Firenze, Italy. http://people.cs.uchicago.edu/~blume/pub.html
3. The complete source code for the article. August 9, 2005. http://pobox.com/~oleg/ftp/Haskell/number-param-vector-code.tar.gz
4. Daniel Fridlender and Mia Indrika: Do we Need Dependent Types? BRICS Report Series RS-01-10, March 2001. http://www.brics.dk/RS/01/10/
5. Oleg Kiselyov: Polyvariadic composition. October 31, 2003. http://pobox.com/~oleg/ftp/Haskell/types.scm#polyvar-comp
6. Oleg Kiselyov: Polymorphic stanamically balanced AVL trees. April 26, 2003. http://pobox.com/~oleg/ftp/Haskell/types.scm#stanamic-AVL
7. Dietmar Kreye: A Compilation Scheme for a Hierarchy of Array Types. Proc. 3th International Workshop on Implementation of Functional Languages (IFL'01).
8. Conor McBride: Faking it—simulating dependent types in Haskell. Journal of Functional Programming, 2002, v.12, pp. 375-392 http://www.cs.nott.ac.uk/~ctm/faking.ps.gz
9. Chris Okasaki: From fast exponentiation to square matrices: An adventure in types. Proc. fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, pp. 28 - 35, 1999 http://www.eecs.usma.edu/Personnel/okasaki/pubs.html#icfp99
10. Sven-Bodo Scholz: A Type System for Inferring Array Shapes. Proc. 3th International Workshop on Implementation of Functional Languages (IFL'01). http://homepages.feis.herts.ac.uk/~comqss/research.html
11. Singe-Assignment C homepage. http://www.sac-home.org/
12. Dominic Steinitz: Re: Polymorphic Recursion / Rank-2 Confusion. Message posted on the Haskell mailing list on Sep 21 2003. http://www.haskell.org/pipermail/haskell/2003-September/012726.html
13. Todd L. Veldhuizen: Arrays in Blitz++. Proc. 2nd International Scientifi c Computing in Object-Oriented Parallel Environments (ISCOPE'98). Santa Fe, New Mexico, 1998. http://www.oonumerics.org/blitz/manual/blitz.html
14. Hongwei Xi, Frank Pfenning: Eliminating Array Bound Checking Through Dependent Types. Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 249–257, 1998. http://www-2.cs.cmu.edu/~hwxi/
15. Hongwei Xi: Dependent Types in Practical Programming. Ph.D thesis, Carnegie Mellon University, September 1998. http://www.cs.bu.edu/~hwxi/