From Fast Exponentiation to Square Matrices: An Adventure in Types

Chris Okasaki
Department of Computer Science
Columbia University
New York, NY 10027
(cdo@cs.columbia.edu)

Abstract

Square matrices serve as an interesting case study in functional programming. Common representations, such as lists of lists, are both inefficient—at least for access to individual elements—and error-prone, because the compiler cannot enforce "squareness". Switching to a typical balanced-tree representation solves the first problem, but not the second. We develop a representation that solves both problems: it offers logarithmic access to each individual element and it captures the shape invariants in the type, where they can be checked by the compiler. One interesting feature of our solution is that it translates the well-known fast exponentiation algorithm *to the level of types*. Our implementation also provides a stress test for today's advanced type systems—it uses nested types, polymorphic recursion, higher-order kinds, and rank-2 polymorphism.

1 Introduction

How would you represent square matrices in your favorite functional language? One common answer is as lists of lists. Unfortunately, this simple approach suffers from at least two weaknesses. First, it is inefficient—accessing an individual element in an $n \times n$ matrix takes $O(n)$ time.[1] Second, it is error-prone—the compiler has no way of enforcing that your square matrices are actually square. The type fails to constrain the inner lists to have the same length as the outer list, or even the same length as each other!

Switching to a typical balanced-tree representation solves only half the problem. It allows logarithmic access to individual elements, but again fails to enforce the desired shape invariants.

In this paper, we develop a representation of square matrices that captures the shape invariants in the type, where they can be checked at compile-time, while retaining logarithmic access to each individual element. One interesting feature of our solution is that it applies the well-known fast exponentiation algorithm *at the level of types*. Our implementation also provides a stress test for today's advanced type systems—it uses nested types, polymorphic recursion, higher-order kinds, and rank-2 polymorphism.

We begin by reviewing the *quadtree* data structure, which we then modify to capture the shape invariants in the type. However, this data structure only supports square matrices whose dimensions are powers of two. We then adapt the ideas of *fast exponentiation* to provide for square matrices of arbitrary dimensions. Finally, we discuss some possible variations and conclude.

[1] On the other hand, a list-of-lists representation can be quite efficient for *bulk* operations.

To appear in ICFP'99.

2 Quadtrees

A *quadtree* is a representation of square matrices that recursively decomposes each non-singleton matrix into four quadrants, each of which is itself a square matrix [10]. David Wise has studied functional quadtrees extensively [11, 4]. See also [3]. Quadtrees are easily implemented in Haskell [9] as

```
> type Quad a = (a,a,a,a)
> data Square1 a = Zero a
>                | Succ (Quad (Square1 a))
```

The names `Zero` and `Succ` refer to powers of two. `Zero` represents a $2^0 \times 2^0$ matrix (i.e., a singleton), and `Succ` represents a $2^{k+1} \times 2^{k+1}$ matrix that is decomposed into four $2^k \times 2^k$ matrices. For now, we disallow matrices whose dimensions are not powers of two.

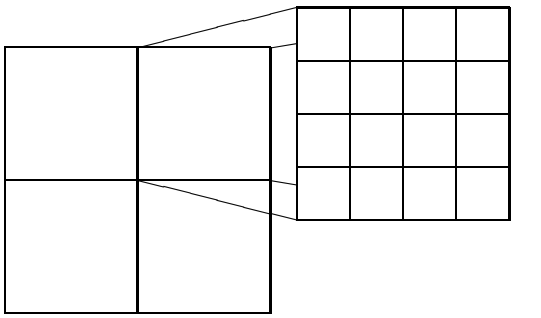
Although Wise has shown that this implementation is efficient and quite convenient for many applications, it fails to enforce the desired shape invariants. The reason is easy to see: in the type `Quad (Square1 a)`, there is no guarantee that the submatrices are the same size.

We can fix this problem by decomposing matrices bottom-up, rather than top-down. That is, instead of decomposing a $2^{k+1} \times 2^{k+1}$ matrix into a single 2×2 grid of $2^k \times 2^k$ matrices, we will now decompose it into a single $2^k \times 2^k$ matrix containing 2×2 grids of elements. These two views are illustrated in Figure 1.

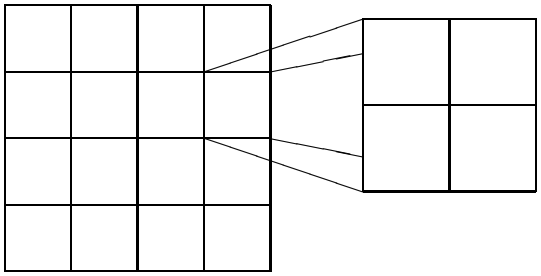
We can implement the new type as

```
> data Square2 a = Zero a
>                | Succ (Square2 (Quad a))
```

This is identical to the old type except that we have reversed the order of the type constructors in the `Succ` case: `Quad (Square1 a)` is now `Square2 (Quad a)`. Figure 2 illustrates a sample 4x4 matrix in both representations.



(top-down)



(bottom-up)

Figure 1: Decomposition of top-down and bottom-up quadtrees.

a	b	e	f
c	d	g	h
i	j	m	n
k	l	o	p

(sample matrix)

```
Succ (Succ (Zero a,Zero b,Zero c,Zero d),
      Succ (Zero e,Zero f,Zero g,Zero h),
      Succ (Zero i,Zero j,Zero k,Zero l),
      Succ (Zero m,Zero n,Zero o,Zero p))
```

(top-down)

```
Succ (Succ (Zero ((a,b,c,d),
                  (e,f,g,h),
                  (i,j,k,l),
                  (m,n,o,p))))
```

(bottom-up)

Figure 2: A sample 4x4 matrix in the top-down and bottom-up representations.

The most striking difference between the two representations is that a tree in the bottom-up representation contains many fewer constructors; the bottom-up approach effectively collapses all the constructors at each level of the tree into a single constructor. In fact, this explains how the bottom-up representation enforces the desired invariants. By allowing multiple constructors at each level of the tree, the top-down representation takes the risk that the constructors may not all be the same (either all `Succ` or all `Zero`. By allowing only a single constructor at each level of the tree, the bottom-up representation avoids this risk.

Finally, note that the bottom-up representation is an example of a *nested* (or *non-regular*) type [1], in which the type `Square2 a` is defined recursively in terms of `Square2 (Quad a)`, rather than in terms of itself. Paterson has also considered bottom-up quadtrees as an example of nested types (personal communication). See [8, 6, 2] for many other examples of nested datatypes.

3 Fast Exponentiation, Vectors, and Rectangular Matrices

Next, we briefly review the well-known *fast exponentiation* algorithm, which raises a number to a non-negative integer power.

```
> exp b n = fastexp 1 b n
>
> fastexp c b n    -- = c * b^n
>   | n == 0 = c
>   | even n = fastexp   c   (b*b) (half n)
>   |  odd n = fastexp (c*b) (b*b) (half n)
```

The iterative version of this function, `fastexp`, works by repeatedly squaring the base `b` and halving the exponent `n`. The accumulating parameter `c` collects the product of all the bases for which `n` was odd. When `n` reaches `0`, the final answer is contained in `c`. The main function `exp` simply calls `fastexp` with `c` initialized to `1`.

Now, what does fast exponentiation have to do with square matrices? Well, instead of raising a number b to a power n, suppose we were to raise a type t to the power n. One way to think of the type t^n is as the type of t-vectors of length n. Then, we can think of the type $(t^n)^n$ as $n \times n$ matrices of t's. More specifically, if we have a type constructor v such that $va = a^n$, then $v(va)$ is the type of $n \times n$ square matrices.

But, we are skipping ahead. For now, let us concentrate on the simpler problem of implementing vectors based on the fast exponentiation algorithm. The following type definitions mirror the definitions of `exp` and `fastexp` except that they are missing the integer argument n.

```
> type Vector a = Vector_ () a
> data Vector_ v w =
>     Zero v
>   | Even (Vector_   v   (w,w))
>   | Odd  (Vector_ (v,w) (w,w))
```

Note that we have replaced multiplication with the product type constructor, and `1` with the unit type `()`. The parameters `v` and `w` are now the types of fixed-length vectors of `a`'s. The product types `(v,w)` and `(w,w)` represent the combination of two fixed-length vectors into another vector whose length is the sum of the other two. The size of each top-level vector is an integer value that is supplied when the vector is created.

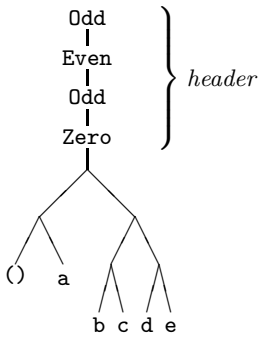


Figure 3: A sample vector containing the elements a,b,c,d,e. Note the header that precedes the actual tree.

```
> create x n = create_ () x n
> create_ v w n
>   | n == 0 = Zero v
>   | even n = Even (create_   v   (w,w) (half n))
>   |  odd n = Odd  (create_ (v,w) (w,w) (half n))
```

Interestingly, the resulting trees bear the same relationship to the *rightist right-perfect trees* of Kaldewaij and Dielissen [7] as bottom-up quadtrees do to ordinary (top-down) quadtrees. Figure 3 shows a sample vector of five elements. Note the path of `Even`, `Odd`, and `Zero` constructors that precedes the actual tree. We call such a path a *header*; it determines the type of the final tree. Such headers appear in many, but by no means all, nested datatypes.

`Vector_` is another example of a *nested type*. As is typical of functions over nested types, the helper function `create_` requires *polymorphic recursion*, because the recursive calls in the function occur at different types. For example, a call to `create_` at type `v -> w -> Int -> Vector_ v w` might make a recursive call to `create_` at type `v -> (w,w) -> Int -> Vector_ v (w,w)` (in the even case) or at type `(v,w) -> (w,w) -> Int -> Vector_ (v,w) (w,w)` (in the odd case). Type inference is undecidable in the presence of polymorphic recursion [5], so Haskell insists that functions involving polymorphic recursion be accompanied by an explicit type signature. Therefore, the above definition is not legal Haskell without the signature

```
> create_ :: v -> w -> Int -> Vector_ v w
```

It is trivial to adapt this implementation of vectors to support *rectangular* matrices. When we finally reach the `Zero` constructor, the type `v` represents fixed-length vectors of the appropriate size. We obtain rectangular matrices simply by considering variable-length vectors of these fixed-length vectors.

```
> type Rect a = Rect_ () a
> data Rect_ v w =
>     ZeroR (Vector v)
>   | EvenR (Rect_   v   (w,w))
>   | OddR  (Rect_ (v,w) (w,w))
```

We now know that all the inner vectors have the same length, but we still do not know whether the outer vector has the same length.

4 Square Matrices

In the previous section, we hinted that the key to representing square matrices is to obtain a *type constructor* `v` for fixed-length vectors. Then, we could apply `v` twice to obtain the type `v (v a)`. However, in the type for rectangular matrices, we obtained a type `v` rather than a type constructor. Hence, to get square matrices, we have to shift all of our type manipulations from the level of types to the level of type constructors. In other words, we need to work at kind `* -> *` rather than kind `*`.

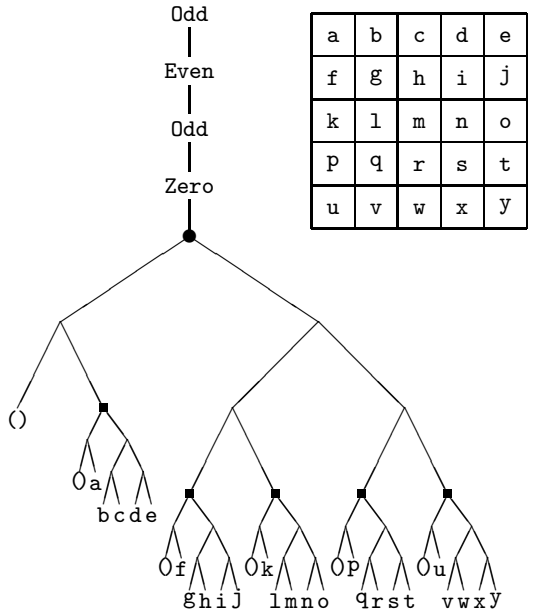


Figure 4: A sample 5×5 square matrix. The circle indicates the root of the outer vector. The square indicates the roots of the inner vectors.

We begin with the following auxiliary types:

```
> newtype Empty a = E ()
> newtype Id a = I a
> newtype Pair v w a = P (v a, w a)
```

`Empty` is the type constructor for empty vectors; `Id` is the type constructor for singleton vectors. If `v` and `w` are type constructors for vectors of length m and n, respectively, then `Pair v w` is the type constructor for vectors of length $m+n$.

With these auxiliary types, we can now write `Square` in the same style as `Vector` and `Rect`, except that we need to carry the element type `a` around until we are ready to apply the desired type constructor.

```
> type Square a = Square_ Empty Id a
> data Square_ v w a =
>     Zero (v (v a))
>   | Even (Square_     v      (Pair w w) a)
>   | Odd  (Square_ (Pair v w) (Pair w w) a)
```

This type remedies both of our original complaints about the lists-of-lists representation. It supports logarithmic access to each individual element, and it captures the "squareness" invariants in the type, where they can be checked by the type checker.[2] Figure 4 shows a sample 5×5 matrix.

Now that we have the type, we can begin to define some functions on this type. Here we consider only the indexing function `sub`; Appendix A contains implementations of

[2] This is true only for the `Square a` type. If the programmer accesses the `Square_` type constructor directly, she can still do all sorts of horrible things.

several other useful functions. We begin with indexing functions for the auxiliary vector types.

```
> subE i (E ()) = fail
> subI 0 (I x) = x
> subP subv subw vsize i (P (v,w))
>   | i <  vsize = subv i v
>   | i >= vsize = subw (i-vsize) w
```

Indexing into the empty vector always fails. Indexing into a singleton vector succeeds only when the index is 0. Indexing into a pair of vectors requires that we know how to index into each of the component vectors. We also need to know the size of the left vector. Indices below that size are in the left vector; indices equal to or above that size are in the right vector.

Using these basic indexing functions as building blocks, `sub` walks the header path of `Even`'s and `Odd`'s, gradually constructing the appropriate indexing functions for the type constructors `v` and `w`. When it finally reaches the `Zero`, it applies the indexing function for `v` twice: once with the row and once with the column. Most of this work is done by the helper function `sub_`.

```
> sub (i,j) m = sub_ subE subI 0 1 (i,j) m
>
> sub_ subv subw vsize wsize (i,j) (Zero vv) =
>    subv i (subv j vv)
> sub_ subv subw vsize wsize (i,j) (Even m) =
>    sub_ subv (subP subw subw wsize)
>         vsize (wsize+wsize) (i,j) m
> sub_ subv subw vsize wsize (i,j) (Odd m) =
>    sub_ (subP subv subw vsize)
>         (subP subw subw wsize)
>         (vsize+wsize) (wsize+wsize) (i,j) m
```

We are dealing with a nested type, so we might expect to use polymorphic recursion. And, indeed, the `Even` and `Odd` cases of `sub_` do require polymorphic recursion. But something even more interesting happens in the `Zero` case. The indexing function `subv` is applied twice: once to a vector of vectors and once to a vector of elements. To support this, `subv` must be polymorphic. But `subv` is not the function being defined, so polymorphic recursion does not help us here. Instead, we need something called *rank-2 polymorphism*, that is, the ability to say that an argument to a function is itself a polymorphic value. The Haskell standard does not support rank-2 polymorphism, but at least two popular implementions (GHC and Hugs) do. In the syntax of those implementations, we would write the signature for `sub_` as

```
> sub_ :: (forall b. Int -> v b -> b)
>            -> (forall b. Int -> w b -> b)
>            -> Int -> Int
>            -> (Int,Int)
>            -> Square_ v w a -> a
```

GHC and Hugs do not attempt to infer rank-2 types, so this type signature is mandatory.[3]

Bird and Paterson [2] also consider how rank-2 types arise when dealing with nested datatypes.

[3]Even without the rank-2 types, the type signature would still be mandatory because of the polymorphic recursion.

5 Performance

Preliminary benchmarks indicate that this implementation runs about twice as slow as an implementation that uses essentially the same algorithms, but with an ordinary representation that does not enforce the shape invariants. The main tradeoffs are these:

- (Less Space) A particular matrix requires many fewer constructors in the nested type than in the equivalent non-nested type. For example, compare the matrices in Figure 2. Assume that there is a non-zero space overhead to represent the tag in a datatype with more than one constructor, and that there is no such overhead in newtypes or tuples. Then the nested representation of an $n \times n$ matrix has an $O(\log n)$ space overhead, compared to an $O(n^2)$ space overhead in the non-nested representation.

- (More Time) Accessing an element in the nested representation may require 50% more pointer chasing than in the equivalent non-nested representation. In the nested representation, accessing an element traverses a total of approximately $3 \log n$ nodes: approximately $\log n$ nodes each in the header, in the outer vector, and in the inner vector. In the non-nested representation, we only traverse the nodes in the outer vector and the inner vector—there is no separate header. (On the other hand, if it there is a time cost associated with pattern matching on a datatype with more than one constructor as opposed to a newtype or a tuple, then the nested representation may come out ahead. It uses non-trivial pattern matching only in the approximately $\log n$ nodes of the header, whereas the non-nested representation uses non-trivial pattern matching at all $2 \log n$ steps through the outer and inner vectors.)

- (More Time) The biggest culprit in the relatively slow speed of the nested representation is the way that it builds up the indexing function `subv` incrementally every time the `sub` function is called. Building all those closures at run-time takes a large amount of time compared to executing a compiled loop. (It also requires a great deal of space, but this demand is transient.) In contrast, the non-nested representation allows the `subv` function to be written as an ordinary loop.

Modifying the implementation to cache a copy of the `subv` function rather than rebuilding it every time, as in Variation #1 in the next section, improves the running time to only about 10% slower than the non-type-safe version.

6 Variations

In this section, we consider a handful of possible variations on this theme, some that work and some that do not. The ones that do not give us insight into which features of our solution are essential.

Variation #1 One of the dominant costs in executing functions such as `sub` is incrementally building the appropriate functions on vectors. These functions have to be built up incrementally, because matrices of different sizes will need functions for vectors of different types. However, multiple calls to `sub` on the same matrix will always build the same vector functions. Ideally, we would be able to share these

functions across the various calls. We can do just that by storing the function as an extra field in the `Zero` constructor. We will again need a rank-2 type signature for the function.

```
> data Square_ v w a =
>     Zero (v (v a)) (forall b. Int -> v b -> b)
>   | Even (Square_     v      (Pair w w) a)
>   | Odd  (Square_ (Pair v w) (Pair w w) a)
```

Now the indexing function is built once when the matrix is initially created, and shared among all matrices descended from this original matrix via a chain of updates. Of course, we may want to keep specialized copies of other functions as well, such as `update`, `map`, etc.

Variation #2 Another variation we might consider is using triples or quadruples rather than pairs. For example, we could rewrite the fast exponentiation algorithm to use base 4

```
> exp b n = fastexp 1 b n
> fastexp c b n
>   | n == 0         = c
>   | n `mod` 4 == 0 =
>       fastexp     c     (b^4) (b `div` 4)
>   | n `mod` 4 == 1 =
>       fastexp   (c*b)   (b^4) (b `div` 4)
>   | n `mod` 4 == 2 =
>       fastexp  (c*b*b)  (b^4) (b `div` 4)
>   | n `mod` 4 == 3 =
>       fastexp (c*b*b*b) (b^4) (b `div` 4)
```

leading to the vector type

```
> type Vector a = Vector_ () a
> data Vector_ v w =
>     Zero v
>   | Mod0 (Vector_     v     (Quad w))
>   | Mod1 (Vector_   (v,w)   (Quad w))
>   | Mod2 (Vector_  (v,w,w)  (Quad w))
>   | Mod3 (Vector_ (v,w,w,w) (Quad w))
```

We could then easily extend the same ideas to get square matrices.

A representation based on quadruples uses approximately one-third as many internal nodes as one based on pairs. If we assume that each quadruple requires 5 words of memory, and each pair requires 3 words, then we would expect the representation based on quadruples to take about $\frac{1}{3} \cdot \frac{5}{3} = \frac{5}{9} \approx 56\%$ as much space as the representation based on pairs. Of course, this does not count the space required for the elements themselves.

Variation #3 A variation that seems like it should work, but that does not, is to switch to the simpler, non-iterative version of fast exponentiation.

```
> exp b n
>   | n == 0 = 1
>   | even n =     exp (b*b) (half n)
>   |  odd n = b * exp (b*b) (half n)
```

Converting this to a type yields

```
> data Vector a =
>     Zero
>   | Even  (Vector (a,a))
>   | Odd a (Vector (a,a))
```

which some readers may recognize as the binary random-access lists of Okasaki [8]. Unfortunately, this type cannot be adapted to yield square matrices. The problem lies in the odd case, where a single element is represented outside the recursive vector type. When we move from the level of types to the level of type constructors, we find that we never get our hands on the vector type constructor of the right length, except in the special case that the length is a power of 2. Hence, we cannot apply this constructor twice to ensure that the outer and inner vectors are all the same length.

Variation #4 A similar non-variation might be based on the naive exponentition algorithm that simply performs n multiplications.

```
> exp b 0 = 1
> exp b n = b * exp b (n-1)
```

Converting this to a type yields

```
> data Vector a =
>     Zero
>   | Succ a (Vector a)
```

We have just re-invented lists! We already know that lists fail to support the invariants we want, but we are now in a better position to say why. At the top level, lists support variable-length data, as they should—so do all the vector types we have considered. However, variable-length data is *all* that lists support, even internally; they have no internal structure that we can exploit.

Variation #5 On the other hand, the iterative version of the naive exponentiation algorithm does provide the guarantees we want.

```
> exp b n = naiveexp 1 b n
>
> naiveexp c b 0 = c
> naiveexp c b n = naiveexp (b*c) b (n-1)
```

From this, we get the type

```
> type Vector a = Vector_ () a
> data Vector_ v a =
>     Zero v
>   | Succ (Vector_ (a,v) a)
```

which is similar to ordinary lists, except that the constructors all appear before any of the data rather than being interleaved with the data. For example, the list containing the elements `1,2,3` would be written in this representation as

```
Succ (Succ (Succ (Zero (1, (2, (3, ()))))))
```

Now, when we adapt this implementation to support square matrices

```
> type Square a = Square_ Empty a
> data Square_ v a =
>     Zero (v (v a))
>   | Succ (Pair Id v) a
```

we find that `Zero` constructor does have access to the vector type constructor of the right length, and hence can apply it twice. Paterson has considered a similar implementation of square matrices (personal communication).

Comparing the last three variations with the solution in Section 4, we see that the variations based on iterative (that is, *tail-recursive*) algorithms work, whereas the variations based on non-iterative algorithms fail. Looking closer, we see that the type inherits the tail recursion of the algorithm, and it is this tail recursion *at the level of types* that ultimately allows the trick of applying the built-up type constructor twice to obtain square matrices.

Variation #6 As a final variation, note that it is trivial to adapt our implementation to higher dimensions, simply by applying the final vector type constructor more than twice. For example, here is the type of cubic matrices.

```
> type Cubic a = Cubic_ Empty Id a
> data Cubic_ v w a =
>     Zero (v (v (v a)))
>   | Even (Cubic_     v      (Pair w w) a)
>   | Odd  (Cubic_ (Pair v w) (Pair w w) a)
```

7 Discussion

We have shown how to implement efficient type-safe square matrices, in which the shape invariants are captured in the type and can be checked by the compiler. In fairness, however, that type-safety may be illusory. The advantage of capturing the invariants in the type is that the compiler has a better chance of catching errors statically. But, this particular application uses such sophisticated concepts that there is probably a much greater chance of introducing bugs in the first place. Whether the greater chance of introducing bugs outweighs the greater chance of detecting bugs, we do not know.

More generally, our case study highlights several interesting concepts in langugage design and programming methodology. First, we illustrate the motivation and use of a number of exotic features of the type system—nested types, polymorphic recursion, higher-order kinds, and rank-2 polymorphism—in a fairly natural setting. Second, we show how and why types can be directly derived from algorithms, especially tail-recursive algorithms. Although we have considered only the family of exponentiation algorithms, we expect that this idea can be usefully applied to other families of algorithms as well.

References

[1] Richard S. Bird and Lambert Meertens. Nested datatypes. In *Conference on Mathematics of Program Construction*, volume 1422 of *LNCS*, June 1998.

[2] Richard S. Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, To appear.

[3] F. Warren Burton and John G. Kollias. Functional programming with quadtrees. *IEEE Software*, 6(1):90–97, January 1989.

[4] Jeremy D. Frens and David S. Wise. Matrix inversion using quadtrees implemented in gofer. Technical Report 433, Computer Science Department, Indiana University, May 1995.

[5] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[6] Ralf Hinze. Polytypic functions over nested datatypes (extended abstract). In *Latin-American Conference on Functional Programming*, March 1999.

[7] Anne Kaldewaij and Victor J. Dielissen. Leaf trees. *Science of Computer Programming*, 26(1–3):149–165, May 1996.

[8] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[9] Simon Peyton Jones et al. Haskell 98: A non-strict, purely functional language. `http://haskell.org/onlinereport/`, February 1999.

[10] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[11] David S. Wise. Matrix algorithms using quadtrees. Technical Report 357, Computer Science Department, Indiana University, June 1992.

A Additional Source Code

A.1

Create an $n \times n$ matrix.

```
> mkE x = E ()
> mkI x = I x
> mkP mkv mkw x = P (mkv x, mkw x)
>
> create x n = create_ mkE mkP n
>
> create_ :: (forall b. b -> v b)
>               -> (forall b. b -> w b)
>               -> a
>               -> Int
>               -> Square_ v w a
> create_ mkv mkw x n
>   | n == 0 = Zero (mkv (mkv x))
>   | even n = Even (create_      mkv      (mkP mkw mkw) x (half n))
>   |  odd n = Odd  (create_ (mkP mkv mkw) (mkP mkw mkw) x (half n))
```

A.2

Calculate the dimensions of a matrix.

```
> dims m = dims_ 0 1 m
>
> dims_ :: Int -> Int -> Square_ v w a -> (Int,Int)
> dims_ nv nw (Zero vv) = (nv,nv)
> dims_ nv nw (Even m)  = dims_   nv    (nw+nw) m
> dims_ nv nw (Odd m)   = dims_ (nv+nw) (nw+nw) m
```

A.3

Apply an update function to an individual element of a matrix.

```
> upE i f (E ()) = fail
> upI 0 f (I x) = I (f x)
> upP upv upw nv i f (P (v,w))
>   | i <  nv = P (upv i f v,w)
>   | i >= nv = P (v,upw (i-nv) f w)
>
> update (i,j) f m = upd_ upE upI 0 1 i j f m
>
> upd_ :: (forall b. Int -> (b -> b) -> v b -> v b)
>            -> (forall b. Int -> (b -> b) -> w b -> w b)
>            -> Int -> Int -> Int -> Int -> (a -> a)
>            -> Square_ v w a -> Square_ v w a
> upd_ upv upw nv nw i j f (Zero vv) = Zero (upv j (upv i f) vv)
> upd_ upv upw nv nw i j f (Even m)  = Even (upd_ upv (upP upw upw nw) nv (nw+nw) i j f m)
> upd_ upv upw nv nw i j f (Odd m)   = Odd (upd_ (upP upv upw nv) (upP upw upw nw)
>                                                (nv+nw) (nw+nw) i j f m)
```

A.4

Map a function across the elements of a matrix.

```
> mapE f (E ()) = E ()
> mapI f (I x) = I (f x)
> mapP mapv mapw f (P (v,w)) = P (mapv f v,mapw f w)
>
> mapMat f m = mapMat_ mapE mapI f m
>
> mapMat_ :: (forall c d. (c -> d) -> v c -> v d)
>               -> (forall c d. (c -> d) -> w c -> w d)
```

```
>               -> (a -> b)
>               -> Square_ v w a -> Square_ v w b
> mapMat_ mapv mapw f (Zero vv) = Zero (mapv (mapv f) vv)
> mapMat_ mapv mapw f (Even m)  = Even (mapMat_       mapv       (mapP mapw mapw) f m)
> mapMat_ mapv mapw f (Odd m)   = Odd  (mapMat_ (mapP mapv mapw) (mapP mapw mapw) f m)
```