# The Functional Imperative: Shape!

C.B. Jay  P.A. Steckler
School of Computing Sciences,
University of Technology, Sydney,
P.O. Box 123, Broadway NSW 2007, Australia;
email: {cbj,steck}@socs.uts.edu.au
fax: 61 (02) 9514 1807

December 10, 1997

## 1 Introduction

FiSh is a new programming language for array computation that compiles higher-order polymorphic programs into simple imperative programs expressed in a sub-language Turbot, which can then be translated into, say, C. Initial tests show that the resulting code is extremely fast: two orders of magnitude faster than Haskell, and two to four times faster than Objective Caml, one of the fastest ML variants for array programming.

Every functional program must ultimately be converted into imperative code, but the mechanism for this is often hidden. FiSh achieves this transparently, using the "equation" from which it is named:

$$\text{Functional} = \text{Imperative} + \text{Shape}$$

Shape here refers to the structure of data, e.g. the length of a vector, or the number of rows and columns of a matrix. The FiSh compiler reads the equation from left to right: it converts functions into procedures by using *shape analysis* to determine the shapes of all array expressions, and then allocating appropriate amounts of storage on which the procedures can act.

We can also read the equation from right to left, constructing functions from shape functions and procedures. Let us consider some examples.

```
>-|> let v = [| 0;1;2;3 |] ;;
```

declares `v` to be a vector whose entries are $0, 1, 2$ and $3$. The compiler responds with both the the type and shape of `v`, that is,

```
v : vec[int]
#v = (~4,int_shape)
```

This declares $v$ has the type of an expression for a vector of integers whose shape `#v` is given by the pair `(~4,int_shape)` which determines the length of `v`, `~4`, and the common shape `int_shape` of its entries. The tilde in ~4 indicates that the value is a *size* used to describe array lengths, which is evaluated statically. By contrast, integers may be used as array entries, and are typically computed dynamically. Arrays can be arbitrarily nested. For example

```
>-|> let x = [| [| 0; 1 |] ; [| 2; 3 |] |] ;;
x : vec[vec[int]]
#x = (~2,(~2,int_shape))
```

is a vector of vectors (or 2×2 matrix). Here too, the entries of the outer vector all have the same shape, namely `(~2,int_shape)`.

FISH primitives can be used to construct higher-order functions. For example, `sum_int` adds up a vector of integers.

```
>-|> let w = sum_int v ;;
w : int
#w = int_shape
>-|> %run w ;;
Shape = int_shape
Value = 6
```

Also, we can map functions across arrays. For example, to sum the rows of `x` above we have

```
>-|> let y = map sum_int x ;;
y : vec[int]
#y = (~2,int_shape)
>-|> %run y ;;
Shape = (~2,int_shape)
Value = [| 1; 5 |]
```

Notice that the compiler was able to compute the shape of `y` before computing any of the array entries. This shape analysis lies at the heart of the FISH compilation strategy. Let us consider how mapping is achieved in more detail.

Consider the mapping `map f x` of a function `f` across a vector `x`. The resulting vector has the same length as `x` but what is the common shape of its entries? In FISH, each function `f` has a shape `#f` which maps the shape of its argument to the shape of its result. Letting `fsh` represent `#f` and `xsh` represent `#x` we can describe the shape of `map f` by the following function

```
>-|> let map_sh fsh xsh = (fst xsh, fsh (snd xsh)) ;;
map_sh : (q -> r) -> s * q -> s * r
```

This function will be used to determine the storage required for `y`. The procedure for assigning values to its entries is given by the following for-loop

```
>-|> let map_pr f y x =
            for (i< @(len_var x))
            {
              y[i] := f !x[i]
            } ;;
map_pr : (a -> b) -> var[vec[b]] -> var[vec[a]] -> comm
```

Here `x:var[vec[a]]` is a *phrase variable*, i.e. an assignable location for a vector whose entries are of data type `a`. Similarly, `y:var[vec[b]]`. Now `len_var x` is the length of `x`, while `@` converts from a size to an integer. Also, `!` extracts the expression associated to a phrase variable, i.e. de-references it. Note that both `x` and `x[i]` are phrase variables.

Finally, the shape function for mapping and its procedure are combined using the function

```
proc2fun : (#[a] -> #[b]) ->
             (var[b] -> var[a] -> comm) -> a -> b
```

It is not a primitive constant of FiSh, but itself is defined using simpler constructions, that act directly on shapes and commands. In particular, it is designed to avoid unnecessary copying of its vector argument. Thus, we have the higher-order, polymorphic function

```
>-|> let map f = proc2fun (map_sh #f) (map_pr f)  ;;
map : (a -> b) -> vec[a] -> vec[b]
```

This same strategy is also used to define other polymorphic array functions for reduction, folding, and so on.

Partial evaluation converts arbitrary FiSh programs into imperative programs in a sub-language called Turbot, which can then be translated to the imperative language of choice, currently ANSI C. For example, the Turbot for `map sum_int x` is given in Figure 1. The variables `B` and `A` store `x` and `y`, respectively.

It might be objected that this approach is not about functional programming at all, since `map` is "just" a disguised procedure. In response, we point out that our `map` is still a higher-order polymorphic function. All functional programs are ultimately compiled into imperative machine code. The difference is that FiSh performs the compilation simply, efficiently and transparently, by making clear the role of shapes. Also, note that FiSh, like Forsythe [Rey96], supports "call-by-name" evaluation. That is, beta-reduction is always legitimate, despite the presence of commands, and side-effects.

Indeed, there is no obligation to program with the imperative features of FiSh directly; we are constructing a library of FiSh functions, called Gold-FiSh, that includes the usual combinators of the Bird-Meertens Formalism [BW88], such as `map`, `reduce`, and many of the standard array operations, such as linear algebra operations and array-based sorting algorithms.

```
new #A = (~2,int_shape) in
  new #B = (~2,(~2,int_shape)) in
    B[0][0] := 0;
    B[0][1] := 1;
    B[1][0] := 2;
    B[1][1] := 3;

    for (0 <= i < 2) {
      new #C = int_shape in
        C := 0;
        A[i] := !C;
        for (0 <= j < 2) {
          A[i] := !A[i] + !B[i][j]
        }
      end
    }
  end
return A
```

Figure 1: TURBOT program for mapping summation.

The chief goal of partial evaluation is to compute the shapes of all intermediate expressions. Such static shape analysis has a number of additional benefits over a comparable dynamic analysis. First, many program errors, such as attempting to multiply matrices whose sizes do not match, show up as compile-time *shape errors*. The basic techniques of static shape analysis were developed in the purely functional language Vec [JS97a]. Its shape analyser is able to detect *all* array bound errors statically, at the cost of unrolling all loops. FISH takes a more pragmatic approach: array indices are treated as integers, not sizes, and so some array bound errors escape detection. However, combinations of functions which are free of array-bound errors, such as those in GOLDFISH, produce programs without bound errors.

Second, knowledge of shapes supports compile-time strategies for data layout. On the small scale, this means not having to box array entries, a significant efficiency gain. On a large scale, FISH satisfies many of the pre-requisites for being a portable parallel programming language. Array combinators have been advocated as a powerful mechanism for structuring parallel programs [Ski94], but the inability to determine the appropriate data distributions has proved a major barrier in the quest for efficiency. Shape-based cost analysis offers a solution to this problem [JCSS97]. Also, FISH is a natural candidate for a co-ordination language for parallel programming (see [DGHY95], for example) since it supports both the second-order combinators useful for data distribution,

and the imperative code used on individual processors.

Static shape analysis requires significant amounts of function in-lining, so we have adopted the strategy of in-lining all function calls. This might be expected to lead to code explosion, but has not been a problem to date. The ability to analyse shapes means that array arguments can be stored, with only references being passed to functions (as in Figure 1) rather than copying of array expressions. Also, many of the standard functions, such as `map`, generate for-loops which only require one copy of the function body. Thus we have eliminated the costs associated with closures with little cost in copying of either code or data. This is a major source of the speed gains in our benchmarks.

As FISH evaluation is "call-by-name" the most natural comparison is with another pure language HASKELL[HPJW92] (which is call-by-need). Our tests show FISH to be over 100 times faster than HASKELL on simple mapping and reducing problems. Of course, it is more common to use a "call-by-value" language with reference types for array computation, such as ML. Different implementations of ML vary significantly in their speed on array programs. O'CAML [Obj] is one of the fastest on arrays. But FISH is two to four times faster than O'CAML on a range of typical problems. Detailed comparisons can be found in Section 3.

Of course, programs built by combining functions will often be slower than hand-built procedures due to the copying inherent in the algorithm, but initial results suggest that the additonal overhead is not so large. We have implemented three matrix multiplication algorithms, using a triple for-loop (pure imperative code), a double for-loop plus inner product (mixed mode) and a purely combinatory algorithm. The combinatory algorithm takes approximately five times longer than the imperative algorithm. If these results scale up for more complex combinations of functions, then the performance penalty for using the functional style will be tolerable for a much larger range of applications than at present.

Finally, observe that FISH supports a smooth transition from prototype to final program. Prototypes written in GOLDFISH can be refined by successively replacing critical parts of the program until the desired efficiency is reached. Hence, there is no need to abandon the prototype when moving to production. Also, because TURBOT is so simple, it can be easily translated to the imperative language of choice, and native code can be embedded in FISH programs.

Thus, FISH is a novel combination of expressive programming style (supporting higher-order, polymorphically typed, functions) and efficient execution that can be integrated with existing imperative languages.

## 2 The FISH Language

FISH is essentially an Algol-like language [Rey81, OT97]. In particular, it supports a sub-language TURBOT that is a simple imperative language equipped with local variables that obey a stack discipline. Commands act on a store, and

function application is call-by-name. To this base, we add the structure of a typed functional language, which can also be used to model procedures.

As in all Algol-like languages, the *data types* (meta-variable $\tau$) are distinguished from the *phrase types* (meta-variable $\theta$). Data types represent storable values, while phrase types represent meaningful program fragments. All of the fundamental types are at play in the rule for typing an assignment

$$\frac{? \vdash A : \mathsf{var}[\alpha] \quad ? \vdash e : \mathsf{exp}[\alpha]}{? \vdash A := e : \mathsf{comm}}$$

$A$ is a *phrase variable* for an array of data type $\alpha$ and $e$ is an *expression* for such an array. Phrase variables can be coerced to expressions, e.g. $!A : \mathsf{exp}[\alpha]$. The assignment itself is a *command*.

In most treatments of Algol-like languages the data types are limited to a fixed set of primitives or *datum types* (meta-variable $\delta$), such as integers and booleans (though see Tennent [Ten89] for another approach). Arrays are treated as processes of phrase type, in which a suitable number of local variables are constructed to store the entries. This contradicts our usual notion of arrays, as storable quantities. However, allowing for array data types introduces an additional complexity. If $\alpha = \mathsf{vec}[\mathsf{int}]$ above then the lengths of $A$ and $e$ may differ, and the assignment generate an array bound error. FISH is able to handle this situation since the compiler is able to determine the shapes of all arrays statically, and check for shape equality for assignments.

This is achieved by dividing the data types into data structures, here just *array types* (meta-variable $\alpha$) and *shape types* (meta-variable $\sigma$). Values of shape type are determined by the compiler. The structured types also support structured phrase variables, as shown by the rule

$$\frac{? \vdash A : \mathsf{var}[\mathsf{vec}[\alpha]] \quad ? \vdash i : \mathsf{exp}[\mathsf{int}]}{? \vdash A[i] : \mathsf{var}[\alpha]}$$

Thus, FISH supports assignment to whole arrays, and to their sub-arrays. More generally, the array data types support a generous class of polymorphic operations, such as mapping and reducing.

FISH v 0.3 is available at `ftp.socs.uts.edu.au/Users/cbj/Fish` by anonymous ftp. It is implemented in OBJECTIVE CAML and currently runs on Sun SparcStations under Solaris, or using the O'CAML byte-code interpreter.

## 2.1 Types

The FISH types are given in Figure 2. Most of the constructions have been introduced above. Note that phrase variables are always of array type $\alpha$, never of shape type. This is because shapes, once declared, cannot be changed by an assignment. There are *type variables* for arrays, shapes or phrases whose kind is indicated by subscripting. Subscripts will often be elided when the kind

**Types and their Shapes**

$$
\begin{array}{rcl}
\delta & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{float} \mid \ldots \\
\alpha & ::= & X_\alpha \mid \delta \mid \mathsf{vec}[\alpha] \\
\beta & ::= & \mathsf{size} \mid \mathsf{fact} \\
\sigma & ::= & X_\sigma \mid \#[X_\alpha] \mid \#[\delta] \mid \beta \mid \sigma \times \sigma \\
\tau & ::= & \alpha \mid \sigma \\
\theta & ::= & X_\theta \mid \#[X_\theta] \mid \mathsf{var}[\alpha] \mid \mathsf{exp}[\tau] \\
 & & \mid \mathsf{comm} \mid \#[\mathsf{comm}] \mid \theta \to \theta \\
\phi & ::= & \theta \mid \forall X_\alpha.\phi \mid \forall X_\sigma.\phi \mid \forall X_\theta.\phi
\end{array}
\qquad
\begin{array}{rcl}
\#\delta & = & \#[\delta] \\
\#X & = & \#[X] \\
\#\#[X] & = & \#[X] \\
\#\mathsf{vec}[\alpha] & = & \mathsf{size} \times \#\alpha \\
\#\sigma & = & \sigma \\
\#\mathsf{var}[\tau] & = & \mathsf{exp}[\#\tau] \\
\#\mathsf{exp}[\tau] & = & \mathsf{exp}[\#\tau] \\
\#\mathsf{comm} & = & \#[\mathsf{comm}] \\
\#\#[\mathsf{comm}] & = & \#[\mathsf{comm}] \\
\#(\theta \to \theta') & = & \#\theta \to \#\theta'
\end{array}
$$

Figure 2: FISH types.

is clear from the context. *Type schemes* are constructed from phrase types by quantification of type variables. The shape type constructor $\#[-]$ may be applied to array and phrase variables, datum types and the command type. It is then extended to an idempotent function $\#$ that acts on all types. Such types are used to represent the shapes of terms. Only two cases of its definition deserve comment. The shape of a variable is an expression, not a variable, since it cannot be assigned. The shape of a vector is a pair, consisting of its length, and the common shape of its entries. Since all array entries have the same shape, the regularity of arrays is enforced. For example, $\#\mathsf{vec}[\mathsf{vec}[\mathsf{int}]] = \mathsf{size} \times (\mathsf{size} \times \#[\mathsf{int}])$. In other words, the entries in a vector of vectors must all have the same length, so that any vector of vectors is a matrix.

The types of the purely imperative sub-language TURBOT are those FISH types that can be constructed without the function arrow constructor, type variables, and the shape of type variables.

In the displays provided by the FISH implementation, type quantifiers are elided, array type variables are constructed using the letters `a,b,c,d`, shape type variables using `q,r,s,t` and phrase type variables using `x,y,z`. Also, `exp[a]` is displayed as `a`. For example, compare the type of `map` given in the introduction with its type according to the language definition:

$$\forall X_\alpha, Y_\alpha.(\mathsf{exp}[X] \to \mathsf{exp}[Y]) \to \mathsf{exp}[\mathsf{vec}[X]] \to \mathsf{exp}[\mathsf{vec}[Y]].$$

## 2.2 Terms

The TURBOT terms are given in Figure 4. The FISH terms extend those of TURBOT by the additional rules in Figure 5. Many of the introduction rules for terms are standard, so we will only address the novelties here.

Shape constants and operations are distinguished from their datum counterparts by pre-fixing a tilde (˜). For example, ˜3 : exp[size] is a size numeral, distinct from the integer 3 and ˜true:fact is a `fact`, distinct from the boolean `true`. One may think of terms of `fact` type as "shape booleans"; alternatively, `fact` terms are compile-time assertions about shape. For example, comparing the shapes of two vectors yields a `fact`:

```
>-|> let v1 = [| 1;2 |];;
v1 : vec[int]
#v1 = (~2,int_shape)
>-|> let v2 = [| 3;4 |];;
v2 : vec[int]
#v2 = (~2,int_shape)
>-|> let w = #v1 == #v2;;
w : fact
#w = ~true
```

Each phrase variable $A$ : var[$\alpha$] has an associated expression $!A$ : exp[$\alpha$] which represents its value. There are also coercions from sizes to integers, and facts to booleans (but not the reverse!). In practice, such coercions will often be inferred during parsing or type inference, but the presence of type variables means that they cannot be abandoned altogether.

FISH supports three forms of conditional expression, with key words if, ife and ifsh which express conditional commands and expressions, and shape-based conditionals, respectively. The command form is standard. Expression conditionals ife $b$ then $e_1$ else $e_2$ have a special status. Both $e_1$ and $e_2$ must have the same shape, which is then the shape of the whole conditional. This is necessary for static shape analysis. If the branches are to have different shapes, then the condition must be resolved statically, i.e. must be a fact $f$. This is done using a *shape conditional* ifsh $f$ then $e_1$ else $e_2$ which is sugar for condsh $f$ $e_1$ $e_2$.

Each (well-shaped) datum of type $\delta$ has the same shape $\delta$_shape which may be thought of as describing the storage requirements for such a datum, e.g. the number of bits required for a floating point real number. Similarly, every (well-shaped) command has the same shape hold. This reflects the principle that commands must leave the store in the same shape as they found it.

The command block new $\#x = e$ in $C$ end introduces a local phrase variable $x$ whose shape is $e$. The combinator newexp is applied to create a local block which returns an expression. The user syntax for newexp's is the same as for command blocks, but with end replaced by return $x$.

The output construction converts an expression to a command which has no effect on the store, but outputs the value of its argument.

Now let us consider the additional FISH terms in Figure 5. Polymorphic terms are constructed by where-clauses. Although we have not proved it, we believe that polymorphism need not be restricted to syntactic values, as in [MHTM97]. The error combinator arises when shape analysis detects a shape

error. The shape combinator # returns the shape of its argument. Its one-sided inverse is null which returns an array expression of the same shape as its argument, but whose entries are undefined. This is mainly for internal use by the compiler. prim_rec is a primitive recursion combinator (see below). hold is the shape of a command without shape errors, while heq compares the shapes of two commands. ap2e allows a procedure to be applied to an expression. Its effect can be modelled using newexp but at the cost of some unnecessary copying.

The FISH primitives can be used to construct a large variety of higher-order polymorphic functions, such as the typical BMF combinators, fundamental linear algebra operations, e.g. matrix multiplication, and other fundamental array operations, e.g. quicksort. A suite of these is being assembled as a purely functional sub-language of FISH, called GOLDFISH. Some of its combinators are given in Figure 6 with their types as presented in the FISH implementation.

General recursion introduces some delicate issues for shape analysis. In the most common cases the shapes produced are independent of the number of iterations involved. This is the case for while loops, since commands are not allowed to change any shapes. These are quite adequate for datum-level recursion. Recursion on arrays is more problematic, since the iterated function may produce a new array of different shape to the original, so that the shape of the result depends on the number of iterations. This is not a problem in the most common cases, e.g. mapping and reducing, which are handled directly in GOLDFISH. The prim_rec combinator handles the general, shape-changing operation by requiring that the number of iterations be a size, and using this to unfold the recursion during compilation. The general case, iterating a shape-changing function for an unbounded number of iterations, is not currently handled.

Type unification and inference proceeds by a variant of the usual Algorithm W [Mil78]. The only difficulty is that the idempotence of the shape function # on types means that most general unifiers, and hence principal types, do not always exist. For example, taken alone, the FISH function

```
fun x -> #x == (~2,int_shape)
```

has ambiguous type, since x could have either type `size * #[int]` or `vec[int]`. But when applied to a term with known kind, the function's type becomes unambiguous:

```
>-|> (fun x -> #x == (~2,int_shape)) (~3,int_shape) ;;
J : fact
#J = ~false
>-|> (fun x -> #x == (~2,int_shape)) [| 1;2 |] ;;
J : fact
#J = ~true
```

The ambiguities vanish once the kinds are known, so that the ambiguities do not present great practical difficulties. We plan to allow programmer annotations to resolve those that do arise.

### 2.3 Evaluation

The evaluation rules are not presented, for lack of space. Most are standard, the only novelties being those for applying the shape combinator #. For example, evaluation of #(`entry` $i$ $v$) proceeds by evaluating `snd` #$v$. Efficiency of the resulting code is greatly enhanced by partial evaluation.

Partial evaluation is used to inline all function calls, apply any constant folding, and evaluate all shapes. It is guaranteed to terminate. This is done without knowledge of the store, other than its shape. Inlining succeeds because we do not support a general fixpoint operator (see above). Shape evaluation succeeds because we have maintained a careful separation of shapes from datum values. The key result is that every FISH term of type `comm` partially evaluates to a TURBOT command, i.e. a simple imperative program without any function calls or procedures.

The FISH implementation takes the resulting TURBOT and performs a syntax-directed translation of TURBOT to C, which is compiled and executed in the usual way. The C code produced is quite readable. The translation uses the known shapes of TURBOT arrays in their C declarations. If there is sufficient space they will be allocated to the run-time stack. Otherwise they are allocated to the heap using a checked form of C's `malloc()` called `xmalloc()`, as in Figure 6.

## 3 Benchmarks

We have compared the run-time speed of FISH with a number of other polymorphic languages for several array-based problems. All tests were run on a Sun SparcStation 4 running Solaris 2.5. C code for FISH was generated using GNU C 2.7.2 with the lowest optimization level using the `-O` flag and all floating-point variables of type `double` (64 bits).

As FISH is a call-by-name language, the most natural comparison is with HASKELL. Preliminary tests using ghc 0.29, the Glasgow HASKELL compiler showed it to be many times slower than FISH. For example, reducing a 100,000-element array in ghc 0.29 took over 16 seconds of user time, compared to 0.04 seconds for FISH. We attribute such lethargy to the fact that Haskell builds its arrays from lists, and may well be boxing its data. Bigloo 1.6, which like FISH compiles a functional language to C, is much slower than FISH.

Now let us consider eager functional languages with mutable arrays, such as ML. O'CAML is one of the faster implementations of ML-related languages, especially for arrays, setting a high standard of performance against which to test our claims about FISH. [1] Hence, we confine further comparisons to O'CAML.

[1] While there is a good deal of contention about which ML implementation is fastest overall, O'CAML appears to be superior at array processing. See the thread in the USENET newsgroup `comp.lang.ml`, beginning with the posting by C. Fecht on October 14, 1996.

| | Map | Reduce | MM loops | MM semi | MM combr |
|---|---|---|---|---|---|
| FiSh | 1.02 | 0.37 | 0.41 | 0.70 | 1.11 |
| O'Caml | 3.67 | 2.18 | 0.71 | 5.38 | 3.45 |

| | Leroy FFT | Quicksort mono | Quicksort poly |
|---|---|---|---|
| FiSh | 3.17 | 8.41 | 8.41 |
| O'Caml | 4.07 | 12.10 | 57.27 |

Figure 3: Benchmarks: FiSh vs. O'Caml.

For O'Caml code, we used ocamlopt, the native-code compiler, from the 1.05 distribution, using the flag `-unsafe` (eliminating arrays bounds checks), and also `-inline 100`, to enable any inlining opportunities. O'Caml also uses 64 bit floats. Also, O'Caml requires all arrays to be initialised, which will impose a small penalty, especially in the first two examples. None of the benchmarks includes I/O, in order to focus comparison on array computation.

We timed seven kinds of array computations: mapping division by a constant across a floating-point array, reduction of addition over a floating-point array, multiplication of floating-point matrices (three ways), the Fast Fourier Transform, and quicksort. The matrix multiplication algorithms use, respectively: a triply-nested for-loop; a doubly-nested loop of inner-products; and a fully combinatory algorithm using explicit copying, transposition, mapping and zipping. Its FiSh form is

```
let mat_mult_float x y =
    zipop (zipop inner_prod_float))
      (map (copy (cols y)) x)
      (copy (rows x) (transpose y)) ;;
```

For each benchmark, the times for FiSh were at least 25% faster than for O'Caml. Where function parameters are being passed, FiSh is from 3 to 7 times faster. The results for the largest input sizes we tried are summarised in Figure 3. Graphs of results for different input sizes are displayed in Figure 6. The times indicated are user time as reported by Solaris `/usr/bin/time`.

In the O'Caml code, one of the costs is the application of the function passed to map, or reduce, which the FiSh compiler inlines. To isolate this effect, we also in-lined the function calls in O'Caml code by hand. This reduced the slow-down for mapping to a factor of two, and for reducing to a factor of three.

Now let us consider the benchmarks in more detail.

The mapping and reduction times are for one-million element float vectors. Each of the matrix multiplications was performed on two $100 \times 100$ square matrices. Our FFT test is based on Leroy's benchmark code for O'Caml, which we

translated mechanically to FISH. For quicksort, we implemented two variants of a non-recursive version of the algorithm, one where the comparison operation was fixed (mono), and another where the operation was passed as an argument (poly). The times are for sorting a 10,000-element vector of floats.

For mapping division over a million-element float array, the FISH benchmark code is

```
let vecTest k = diagarray_float (k ~* ~100000) ;;
let mapVecTest = map (fun x -> x /. 7.0 ) ;;
let maptest10 = mapVecTest (vecTest ~10) ;;
```

where `~*` indicates multiplication on sizes, and `/.` is used for floating-point division. `diagarray_float` builds an array where each element's value is the same as its index, represented as a float. Its definition and those for `map` and other essential combinators here and below are provided in the file `gold.fsh`, which is loaded on startup of the interactive FISH shell. The O'CAML code for the mapping test is nearly identical, except that it uses ordinary integers in place of sizes.

Next we consider reduction of a float array. The FISH code for the million-element array is:

```
let redVecTest = reduce plus_float 0.0 ;;
let redtest10 = redVecTest (vecTest ~10) ;;
```

Since O'CAML has no reduction operation on arrays, we implemented the operation with a `for`-loop that updates a reference accumulator.

In each case, FISH is faster — in the combinatory case by a factor of five. Also note that the FISH combinatory code took approximately five times longer than the loops version. (Surprisingly, the O'CAML semi-combinatory code was faster than its loops version.)

# 4 Relations to other work

As mentioned, the FISH language borrows many features from ALGOL-like languages. [OT97] is a collection of much of the historically significant work in this area. Non-strict evaluation and type polymorphism are also features of HASKELL [HPJW92]. While HASKELL does feature arrays, speed of array operations is not a particular concern. Key features of FISH are its collection of polymorphic combinators for array programming, and the ability to define new polymorphic array operations. OBJECTIVE CAML also offers some polymorphic operations in its `Array` module, though no compile-time checking of shape and array bound errors is provided [Obj]. APL is a programming language specialized for array operations, though it does not offer higher-order functions [Ive62]. ZPL is a recent language and system for array programming on parallel machines [ZPL].

Turning to compilation techniques, general results on partial evaluation can be found in [NN92, JGS93]. Inlining is still a subject of active research, e.g. using flow analysis to detect inlining opportunities, [JW96, Ash97]. Inlining in FISH is complete, so that there are no functions left in the generated code. Thus, FISH code is specialised for the shapes of its arguments, providing a more refined specialisation than one based just on the type [HM95]. In contrast to the approximations provided by program analyses based on abstract interpretation [CC77], FISH obtains exact shapes of data.

Similarly, unboxing of polymorphic function parameters (e.g. [Ler97]) becomes moot, so our system gives a kind of "radical unboxing". Also, FISH's stack discipline obviates the need for garbage collection. Tofte and Talpin's *regions* inference is an attempt to gain the benefits of stack discipline in ML, while still allowing a global store [TT94]. An implementation of this approach based on the ML Kit has recently become available [TBE+97]. In the regions approach, the best sizes of the regions needed are not easily determined. The ML Kit with Regions has tools for allowing programmers to tune their region allocations. In the FISH system, shape analysis assures that allocations are always of exactly the right size.

# 5 Future Work

Future work will proceed in several directions. Here are two major goals. First, to extend the language to support more data types, e.g. trees, whose shapes are more complex. We have already begun work on supporting *higher-order arrays*, which have an arbitrary finite number of dimensions, and may contain arrays as elements. Such extensions bring the possibility of combining shape analysis with the other major application of shape, shape polymorphism [BJM96], or polytypism [JJ97]. Second, we are implementing support for general recursion for first-order procedures. Third, FISH is ideally suited to play the role of a co-ordination language for parallel programming, since the GOLDFISH combinators are ideal for distribution and co-ordination, while the imperative code runs smoothly on the individual processors.

# 6 Conclusions

FISH combines the best features of the functional and imperative programming styles in a powerful and flexible fashion. This leads to significant efficiency gains when executing functional code, and to significant gains in expressiveness for imperative programs. The key to the smooth integration of these two paradigms is shape analysis, which allows polymorphic functional programs to be constructed from imperative procedures and, conversely, functional programs to be compiled into imperative code.

Static shape analysis determines the shapes of all intermediate data structures (i.e. arrays) from those of the arguments, and the shape properties of the program itself. In conventional functional languages this is impossible, since array shapes are given by integers whose value may not be known until execution. FiSh avoids this problem by keeping shape types separate from datum types.

**Acknowledgements** We would like to thank Chris Hankin, Bob Tennent and Milan Sekanina for many productive discussions.

# References

[Ash97] J.M. Ashley. The effectiveness of flow analysis for inlining. In *Proc. 1997 ACM SIGPLAN International Conf. on Functional Programming (ICFP '97)*, pages 99–111, June 1997.

[BJM96] G. Bellé, C. B. Jay, and E. Moggi. Functorial ML. In *PLILP '96*, volume 1140 of *LNCS*, pages 32–46. Springer Verlag, 1996. TR SOCS-96.08.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* International Series in Computer Science. Prentice Hall, 1988.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Conf. Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[DGHY95] J. Darlington, Y.K Guo, To H.W., and Jing Y. Functional skeletons for parallel coordination. In *Proceedings of Europar 95*, 1995.

[HM95] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, January 1995.

[HPJW92] P. Hudak, S. Peyton-Jones, and P. Wadler. Report on the programming language Haskell: a non-strict, purely functional language. *SIGPLAN Notices*, 1992.

[Ive62] K.E. Iverson. *A Programming Language.* Wiley, 1962.

[JCSS97] C.B. Jay, M.I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 650–661. Springer, August 1997.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall, 1993.

[JJ97] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[JS97a] C.B. Jay and M. Sekanina. Shape checking of array programs. In *Computing: the Australasian Theory Seminar, Proceedings, 1997*, volume 19 of *Australian Computer Science Communications*, pages 113–121, 1997.

[JW96] S. Jagannathan and A. Wright. Flow-directed inlining. In *Proc. ACM SIGPLAN 1996 Conf. on Programming Language Design and Implementation*, pages 193–205, 1996.

[Ler97] X. Leroy. The effectiveness of type-based unboxing. In *Abstracts from the 1997 Workshop on Types in Compilation (TIC97).* Boston College Computer Science Department, June 1997.

[MHTM97] R. Milner, R. Harper, M. Tofte, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978.

[NN92] F. Nielson and H.R. Nielson. *Two-level functional languages.* Cambridge University Press, 1992.

[Obj] OBJECTIVE CAML home page on the World-Wide Web. `http://pauillac.inria.fr/ocaml`.

[OT97] P.W. O'Hearn and R.D. Tennent, editors. *Algol-like Languages, Vols I and II.* Progress in Theoretical Computer Science. Birkhauser, 1997.

[Rey81] J.C. Reynolds. The essence of ALGOL. In J.W. de Bakker and J.C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. IFIP, North-Holland Publishing Company, 1981.

[Rey96] John C. Reynolds. Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, June 1996.

[Ski94] D.B. Skillicorn. *Foundations of Parallel Programming.* Number 6 in Cambridge Series in Parallel Computation. Cambridge University Press, 1994.

[TBE+97] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T.H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ML kit. Technical Report 97/12, Univ. of Copenhagen, 1997.

[Ten89] R.D. Tennent. Elementary data structures in Algol-like languages. *Science of Computer Programming*, 13:73–110, 1989.

[TT94] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value $\lambda$-calculus using a stack of regions. In *Conf. Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, January 1994.

[ZPL] ZPL home page. `http://www.cs.washington.edu/research/zpl`.

**Term and Phrase Variables**

$$\frac{?\,(x) \succ \theta}{?\ \vdash x : \theta} \qquad \frac{?\ \vdash A : \mathsf{var}[\mathsf{vec}[\alpha]] \quad ?\ \vdash i : \mathsf{exp}[\mathsf{int}]}{?\ \vdash A[i] : \mathsf{var}[\alpha]}$$

**Expressions**

$$\frac{}{?\ \vdash d : \mathsf{exp}[\delta]} \qquad \frac{?\ \vdash e_1 : \mathsf{exp}[\delta_1] \quad ?\ \vdash e_2 : \mathsf{exp}[\delta_2]}{?\ \vdash e_1\ \oplus\ e_2 : \mathsf{exp}[\delta_3]}$$

$$\frac{}{?\ \vdash sh : \mathsf{exp}[\beta]} \qquad \frac{?\ \vdash sh_1 : \mathsf{exp}[\beta_1] \quad ?\ \vdash sh_2 : \mathsf{exp}[\beta_2]}{?\ \vdash sh_1\ \oplus\ sh_2 : \mathsf{exp}[\beta_3]}$$

$$\frac{?\ \vdash i : \mathsf{exp}[\mathsf{size}]}{?\ \vdash @i : \mathsf{exp}[\mathsf{int}]} \qquad \frac{?\ \vdash sh : \mathsf{exp}[\mathsf{fact}]}{?\ \vdash \mathsf{fact2bool}(sh) : \mathsf{exp}[\mathsf{bool}]}$$

$$\frac{?\ \vdash A : \mathsf{var}[\alpha]}{?\ \vdash !A : \mathsf{exp}[\alpha]} \qquad \frac{?\ \vdash b : \mathsf{exp}[\mathsf{bool}] \quad ?\ \vdash e_1 : \mathsf{exp}[\alpha] \quad ?\ \vdash e_2 : \mathsf{exp}[\alpha]}{?\ \vdash \mathsf{ife}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathsf{exp}[\alpha]}$$

$$\frac{}{?\ \vdash \delta\_\mathsf{shape} : \mathsf{exp}[\#[\delta]]} \qquad \frac{?\ \vdash e_1 : \mathsf{exp}[\sigma_1] \quad ?\ \vdash e_2 : \mathsf{exp}[\sigma_2]}{?\ \vdash \langle e_1, e_2 \rangle : \mathsf{exp}[\sigma_1 \times \sigma_2]}$$

**Commands**

$$\frac{}{?\ \vdash \mathsf{skip} : \mathsf{comm}} \qquad \frac{?\ \vdash C_1 : \mathsf{comm} \quad ?\ \vdash C_2 : \mathsf{comm}}{?\ \vdash C_1; C_2 : \mathsf{comm}}$$

$$\frac{?\ \vdash A : \mathsf{var}[\alpha] \quad ?\ \vdash e : \mathsf{exp}[\alpha]}{?\ \vdash A := e : \mathsf{comm}} \qquad \frac{}{?\ \vdash \mathsf{abort} : \mathsf{comm}}$$

$$\frac{?\ \vdash b : \mathsf{exp}[\mathsf{bool}] \quad ?\ \vdash C_1 : \mathsf{comm} \quad ?\ \vdash C_2 : \mathsf{comm}}{?\ \vdash \mathsf{if}\ b\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : \mathsf{comm}}$$

$$\frac{?\ \vdash m : \mathsf{exp}[\mathsf{int}] \quad ?\ \vdash n : \mathsf{exp}[\mathsf{int}] \quad ?, i : \mathsf{exp}[\mathsf{int}] \vdash C : \mathsf{comm}}{?\ \vdash \mathsf{for}\ (m \le i < n)\ C : \mathsf{comm}}$$

$$\frac{?\ \vdash b : \mathsf{exp}[\mathsf{bool}] \quad ?\ \vdash C : \mathsf{comm}}{?\ \vdash \mathsf{while}\ b\ \mathsf{do}\ C : \mathsf{comm}}$$

$$\frac{?\ \vdash e : \mathsf{exp}[\#\alpha] \quad ?, x : \mathsf{var}[\alpha] \vdash C : \mathsf{comm}}{?\ \vdash \mathsf{new}\ \#x = e\ \mathsf{in}\ C\ \mathsf{end}\ : \mathsf{comm}} \qquad \frac{?\ \vdash e : \mathsf{exp}[\alpha]}{?\ \vdash \mathsf{output}(e) : \mathsf{comm}}$$

Figure 4: TURBOT terms.

**Functions**

$$\frac{?, x : \theta_1 \vdash t : \theta_2}{? \vdash \lambda x.\ t : \theta_1 \to \theta_2} \qquad \frac{? \vdash t : \theta_1 \to \theta_2 \quad ? \vdash t_1 : \theta_1}{? \vdash t\ t_1 : \theta_2}$$

$$\frac{? \vdash t_2 : \theta_2 \quad ?,\ x : \mathsf{Clos}_\Gamma(\theta_2) \vdash t_1 : \theta_1}{? \vdash t_1 \text{ where } x = t_2 : \theta_1} \qquad \frac{c_\phi \succ \theta}{? \vdash c : \theta}$$

**Combinators**

| | | |
|---|---|---|
| error | : | $\forall X_\theta.\ X$ |
| condsh | : | $\forall X_\theta.\ \mathsf{exp}[\mathsf{fact}] \to X \to X \to X$ |
| # | : | $\forall X_\theta.\ X \to \#[X]$ |
| null | : | $\forall X_\theta.\#X \to X$ |
| prim_rec | : | $\forall X_\theta.\ (\mathsf{exp}[\mathsf{size}] \to X \to X) \to X \to \mathsf{exp}[\mathsf{size}] \to X$ |
| equal | : | $\forall X_\alpha.\ \#X \to \#X \to \mathsf{exp}[\mathsf{fact}]$ |
| fst | : | $\forall X_\sigma, Y_\sigma.\ \mathsf{exp}[X \times Y] \to \mathsf{exp}[X]$ |
| snd | : | $\forall X_\sigma, Y_\sigma.\ \mathsf{exp}[X \times Y] \to \mathsf{exp}[Y]$ |
| entry | : | $\forall X_\alpha.\ \mathsf{exp}[\mathsf{size}] \to \mathsf{exp}[\mathsf{vec}[X]] \to \mathsf{exp}[X]$ |
| newexp | : | $\forall X_\alpha.\ \mathsf{exp}[\#[X]] \to (\mathsf{var}[X] \to \mathsf{comm}) \to \mathsf{exp}[X]$ |
| hold | : | $\#[\mathsf{comm}]$ |
| heq | : | $\#[\mathsf{comm}] \to \#[\mathsf{comm}] \to \mathsf{exp}[\mathsf{fact}]$ |
| ap2e | : | $\forall X_\alpha.\ (\mathsf{var}[X] \to \mathsf{comm}) \to (\mathsf{exp}[X] \to \mathsf{comm})$ |

Figure 5: Additional FISH terms.

```
comp:  ((y -> x) -> ((z -> y) -> (z -> x)))
identity:  (x -> x)
check:  (fact -> (x -> x))
proc2fun:  ((#a -> #b) -> ((var[b] -> (var[a] -> comm)) ->
                                                  (a -> b)))
len:  (vec[a] -> size)
map:  ((b -> a) -> (vec[b] -> vec[a]))
zipop:  ((c -> (b -> a)) -> (vec[c] -> (vec[b] -> vec[a])))
reduce:  ((a -> (b -> a)) -> (a -> (vec[b] -> a)))
fold:  ((a -> (x -> x)) -> (x -> (vec[a] -> x)))
copy:  (size -> (a -> vec[a]))
sum_float:  (vec[float] -> float)
transpose:  (vec[vec[a]] -> vec[vec[a]])
mat_mult_float:  (vec[vec[float]] -> (vec[vec[float]] ->
                                        vec[vec[float]]))
```

Figure 6: Some GOLDFISH combinators.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#include "fish.h"

int main(int argc,char *argv[]) {

  {
    double *A;
    A = xmalloc(1000000 * sizeof(A[0]));

    {
      double *B;
      B = xmalloc(1000000 * sizeof(B[0]));

      {
        int i;
        for (i = 0; i < 1000000; i++) {
          B[i] = (double)(i);
        }
      }

      {
        int i;
        for (i = 0; i < 1000000; i++) {
          A[i] = B[i] / 7.000000;
        }
      }

      free(B);
    }

    free(A);
  }

  return 0;
}
```

Figure 7: Generated C code for mapping division over a float array.

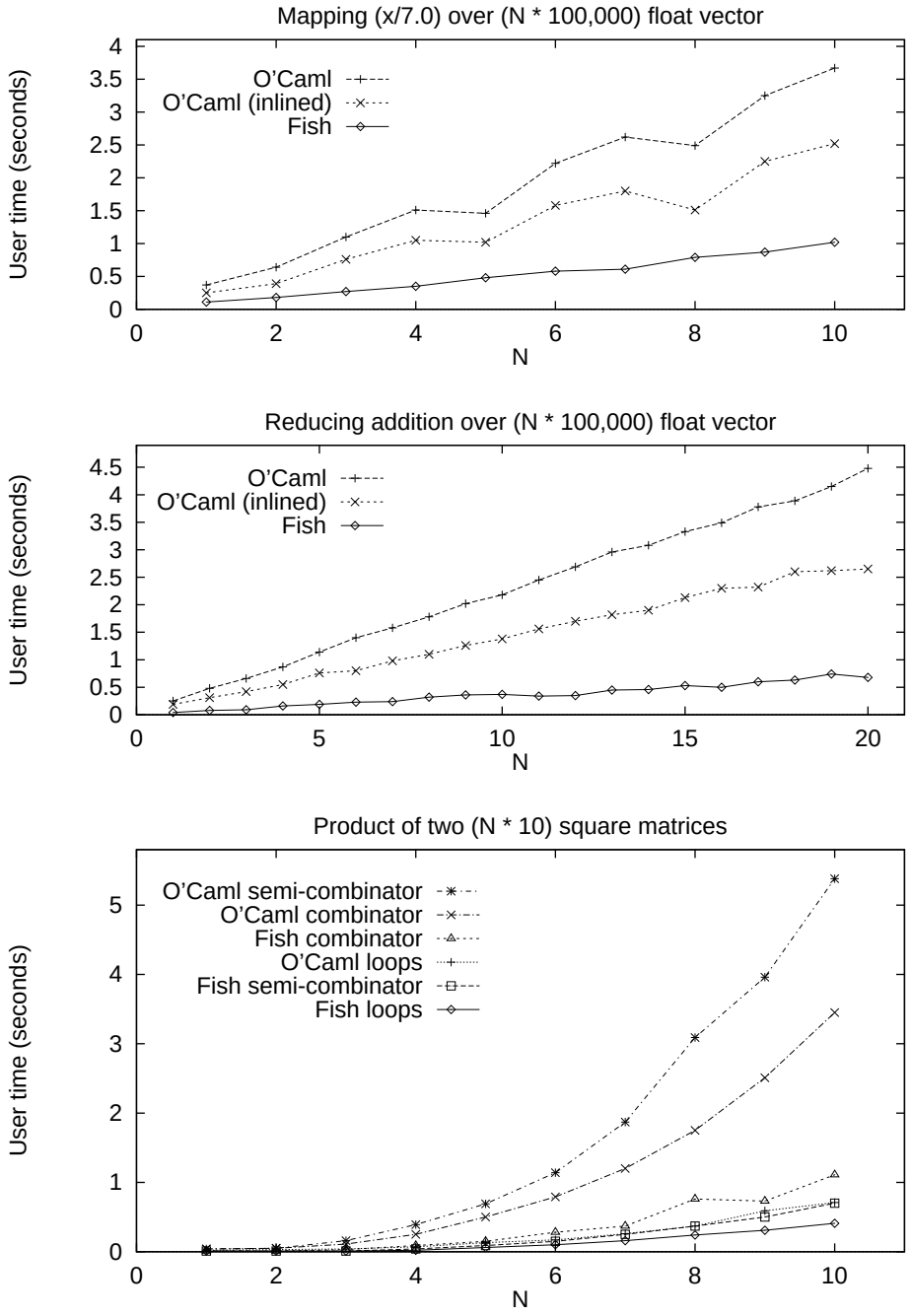


Figure 8: Benchmark results.