Partial evaluation of shaped programs:  
experience with FISh

C.B. Jay  
School of Computing Sciences,  
University of Technology, Sydney  
P.O. Box 123, Broadway NSW 2007, Australia;  
email: cbj@soc.scs.uts.edu.au

Abstract  

FISh is an array-based programming language that combines imperative and functional programming styles. Static shape analysis uses partial evaluation to convert higher-order polymorphic programs into simple, efficient imperative programs. This paper explains how to compute shapes statically, and uses concrete examples to illustrate its several effects on performance.

1 Introduction

Partial evaluation uses limited information about inputs to optimise a program. Common instances are datum values, e.g., integers and booleans, and the shapes of data structures, e.g., the length of a list or the number of rows and columns of a matrix. Datum values can be used to unwind a recursion or evaluate a conditional, while shape information can be used to simplify data layout and memory management, e.g., by unboxing data. Shape information may be provided explicitly in a program, e.g., using types such as int[2][3], but this approach severely limits program reuse. Conversely, polymorphic programming languages, such as ML and Haskell, tend to focus on inductive types, e.g., lists and trees, but do not provide any support for inferring shapes. Shape theory [Jay93] provides a formal account of data types as shaped entities, which supports programming with shapes. It has been used to guide the types, terms and compilation strategy of the FISh programming language [FISh] [JS98].

The explicit use of shapes in FISh supports several advantages not currently possible in other languages, namely: new forms of polymorphism, especially polydimensionality (the ability to apply a program to an array with an arbitrary number of dimensions) [Jay98b]; static detection of shape errors e.g., many array-bound errors [Sek98]; and, program optimisations. All of these advantages are achieved using (static) shape analysis of programs during off-line partial evaluation. This paper will focus on its use in optimisation.

The most obvious benefit of shape information is in improved memory management. This is of crucial importance, in parallel and distributed programming, but is also a significant issue in sequential implementations. For example, unboxing eliminates a level of indirection in accessing data, e.g., replacing an array of pointers to floats by an array of floats, but then access to entries requires that their size be known in order to compute offsets. When the entries are of datum type then this can be inferred from the type [HM95, Ler97] but in general, if the entries are themselves structured, e.g., arrays, then type inference is insufficient, and a proper shape analysis is required. FISh is already able to handle polydimensional arrays, and is being extended to cope with inductive types, such as lists.

A more subtle benefit of shapes arises from improved separation of denotational and operational issues. This can be seen most clearly by comparing lists and vectors (one-dimensional arrays). It is common to distinguish these operationally: a vector typically indicates some combination of a named block of storage, constant access time to all entries and in-place update; while a list typically indicates a pointer to the heap, linear access time and referential transparency. Shape theory distinguishes vectors and lists denotationally: a vector is a list whose entries all have the same shape. For example, the entries in a vector of vectors must all have the same length, so that the whole corresponds to a matrix rather than an arbitrary list of lists. This regularity of vectors (and arrays generally) supports the operational features mentioned above, but they are not inherent.

FISh exploits this by allowing both assignable arrays of type var α and array expressions of type exp α. The former support assignment, and hence in-place update, while the latter can only be used once, and so may be re-used for other purposes. Conversely, one can envisage assignable lists, where each entry has different, but fixed, memory requirements. This distinction between var and exp types is inherited from Reynolds' Algol-like languages [Rey81] but the use of shape analysis means that it can be applied to structured data types as well as datum types. The relationship can be captured by the following slogan, from which the name “FISh” is derived:

\[
\text{Functional} \Rightarrow \text{Imperative + Shape}
\]

That is, higher-order, referentially transparent, functional programs can be constructed from efficient imperative procedures combined with shape information. The latter is used to control creation of local variables to which the procedure can be applied. Partial evaluation computes all of the shape information, reducing the higher-order functions to imperative procedures. Without further effort, this approach generates too many duplicate data structures, and pointless copying. Further optimisations, based on shape and free-variable analysis, eliminate most unnecessary structures.

A third source of efficiency is that shapes can be used by the programmer to optimise some algorithms. We will use folding (or reduction) over arrays as our example.
These benefits are augmented by an aggressive approach to function in-lining, which is the default choice for all (non-recursive) functions. This works well with the data-centric approach, and its support for while- and for-loops, where code copying is not a problem. Future versions are likely to pass some control over in-lining back to the programmer.

Aspects of these techniques are already familiar in partial evaluation. Shape theory provides a unified framework which selects these techniques from the range currently available, and presents them in a more general form than was previously possible. In combination they allow higher-order, polymorphic programs to be converted into simple, efficient imperative programs. A variety of small-to-medium sized programs have been written in FISH. Typical performance of polymorphic FISH programs is many times faster than equivalent programs in other polymorphic functional languages, and comparable to corresponding monomorphic programs in C (the target language of the current implementation). Even where C is polymorphic, FISH is typically faster. For example, polymorphic quicksort (quicksort in C) is twice as fast in FISH on large arrays of floats.

The sections of the paper address the following topics: introduction; review of the FISH language; partial evaluation in FISH; examples of optimisation; and, conclusions.

2 The FISH language

This section reviews the types and terms of the FISH language. A large amount of background material can be found at the FISH web-site [FISH] including an introduction to shape theory [Jay95], introductory articles on FISH [J998, J998a] a formal definition of the language, including partial evaluation and execution rules, a tutorial, sample programs and benchmarks.

2.1 Types

The raw syntax for the FISH types is given by

\[
\delta : D ::= \text{int} | \text{bool} | \text{float} | \text{char} | \ldots
\]

\[
\alpha : A ::= X | \delta | [\alpha]
\]

\[
\sigma : \text{Sh} ::= \cdots
\]

\[
\tau : T ::= \alpha | \sigma
\]

\[
\theta : P ::= U | \#U | \text{comm} | \text{var} \alpha | \exp \tau | \theta \rightarrow \theta
\]

\[
\phi : S ::= \theta | \forall \alpha : A, \phi | \forall U : P, \phi
\]

Following after Reynolds's account of Algol-like languages [Rey81, OT97] FISH distinguishes the data types (meta-variable \(\sigma\)), which represent storable values, from the phrase types (meta-variable \(\delta\)), which represent meaningful program segments. The data types are further divided into the array types (meta-variable \(\alpha\)) which are used to store arrays of data, and the shape types or static types (meta-variable \(\sigma\)) whose values are computed during compilation. These are used for static constants, and for describing the shape or structure of arrays, e.g. how many atoms of data an array will hold.

The array types are either array types variables (meta-variables \(X, Y : A\)), datum types (meta-variable \(\delta\)) or arrays \([\alpha]\) of \(\alpha\). Datum types represent atoms of data; currently, FISH supports datum types for integers int, booleans bool, reals or floats, float, and characters, char. The array type \([\alpha]\) represents regular arrays of \(\alpha\)'s. Here regular means that the arrays are finite-dimensional, rectangular, and their entries all have the same shape. For example the entries in an array of type \([[\text{int}\]]\) must all be arrays that have the same number of dimensions, i.e. the same rank, and size in each dimension. This means that array programs are able to act on arrays of arbitrary rank and size, i.e. are polydimensional programs.

Every datum type \(\delta\) has a corresponding shape type, called \(\delta\), whose values are computed statically, as compile-time constants. This distinction is similar to that in two-level languages, as in [NN92, BM97]. Here are some typical uses. The type size = \(\text{int}\) of sizes is used to represent the length, or size, of an array in a given dimension. The type fact = \(\text{bool}\) is used for static booleans, or facts, which are useful for expressing properties of shapes required during compilation. The type cost = \(\text{float}\) is used for static floats, useful for static cost analysis. The type mark = \(\text{char}\) may be used for labels.

The other shape types are of the form \#\(\alpha\) which represents the shapes of arrays of type \(\alpha\). The values of such a type correspond to lists of sizes (one for each dimension, outermost first) paired with the common shape of the array entries. These types of array shapes are a new idea. Partial evaluation of an array of type \(\alpha\) will include the complete evaluation of its shape of type \#\(\alpha\) without any explicit separation of inputs into static and dynamic parts.

Take care not to confuse \(\delta\) and \#\(\delta\). The former type has many values, one for each value of type \(\delta\), representing sizes, facts, etc. The latter has only one value, representing the common shape of all \(\delta\)-values. For example, \(3 : \text{size}\) compared to int shape : \#(int).

Many of the type distinctions above originate in the semantics of arrays introduced in [Jay94] and further developed in [Jay99]. However, their motivation from a programming perspective is not so compelling. Future versions of FISH may simplify the type system, and hence the term structure, but this will produce fresh semantic challenges.

Now let us consider the phrase types. Phrase type variables (meta-variable \(U : P\)) are used to express polymorphism. Each such has a shape \#\(U\) (see below).

The type \(\text{comm}\) of commands represents operations that modify the store, such as assignments.

Data types are used to construct phrase types in two distinct ways. Each array type \(\alpha\) yields a type var \(\alpha\) of array variables of type \(\alpha\). Terms of this type have mutable values. Each data type \(\tau\) yields a type exp \(\tau\) of expressions of type \(\tau\) whose values are immutable. Array variables represent stored quantities, much as reference types do in ML.

Unlike earlier Algol-like languages, which could only store atomic data, FISH also supports storable arrays. Consequently, one is able to define polymorphic array operations, such as mapping and reducing, which work for arrays of arbitrary shape, without having to fix the shape in advance. This appears to be in conflict with the well-known incompatibility of references and polymorphism ([To88] and also [Ok93]) but in FISH all polymorphism is instantiated statically, before execution.

The function type \(\theta_1 \rightarrow \theta_2\) represents functions from \(\theta_1\) to \(\theta_2\). When \(\theta = \text{comm}\) the result is a procedure. A ground type is a phrase type which is not a function type.

Note that although FISH supports functions of arbitrarily high type, and that functions are first-class citizens as phrases (i.e. they can be passed as arguments to functions, and returned as results) they are not storable values because their shape, and hence their storage requirements, are un-
known. In particular, the shape of a function is a function (of shapes) for which no equality test is available. Hence the
regularity condition for array entries cannot be established.
Every phrase type \( \theta \) has an associated phrase type \( \text{shp} \) \( \theta \) (or \( \#\theta \)) which is its shape (see the language definition for
details). The key point for our discussion is that the shape of
a function is a function of shapes, i.e.,

\[
\#(\theta_0 \to \theta_1) = \#\theta_0 \to \#\theta_1
\]

This property of the types reflects the idea that the shapes
of all data structures can be computed statically, e.g. if \( f : \exp \text{[int]} \to \exp [a] \) is a function on arrays of integers and \( a \) is such an array then the shape of \( f \) \( a \) is \( \#(f \ a) = \#f \ #a \)
which can be computed from the knowledge of \( f \) and the
shape of \( a \).

Also, commands are not allowed to change the shape of
the store, and hence all well-shaped commands have the
same shape which is, by convention, the true fact \( \text{true} \).

\textit{FISH} supports Hindley-Milner style polymorphism using
type schemes (meta-variable \( \phi \)) obtained by quantifying
over array and phrase type variables. The scheme \( \forall X : A : \phi \)
represents quantification by an array type variable \( X \) and \( \forall U : P : \phi \)
represents quantification by a phrase type variable \( U \).

\[
\begin{align*}
\text{Commands} & : \text{skip, abort, assign, seq, cond, forall, whiletrue, fix, newvar, output, get, subexp, condexp, newexp, dyn} \in \phi \\
\text{Array variables} & : \text{get, sub, var} \in \phi \\
\text{Essential datum constants} & : n \in \phi \\
\text{Array expressions} & : \text{var2exp, d, getexp, subexp, condexp, newexp} \in \phi \\
\text{Shape expressions} & : \text{d, zero, succ, undim, lead, pred, numdim, equal} \in \phi \\
\text{Phrase polymorphic constants} & : \text{condsh, primrec, error, shape} \in \phi
\end{align*}
\]

\[
\begin{align*}
\text{Figure 1: FISH Constants}
\end{align*}
\]

\[
\begin{align*}
\text{Commands} & : \text{skip : comm, abort : comm, assign : } \forall X : A : \text{var } X \to \exp X \to \text{comm} \\
& \text{seq : comm } \to \text{comm, cond : } \exp \text{bool } \to \text{comm } \to \text{comm, forall : } \exp \text{int } \to \exp \text{int } \to \text{comm } \to \text{comm, whiletrue : } \exp \text{bool } \to \text{comm, fix : } (\text{comm } \to \text{comm} ) \to \text{comm, newvar : } \forall X : A : \exp \text{#X } \to \text{(var X } \to \text{comm} ) \to \text{comm, output : } \forall X : A : \exp X \to \text{comm}
\end{align*}
\]

\[
\begin{align*}
\text{Array variables} & : \text{get : } \forall X : A : \text{var } X \to \exp X \\
& \text{sub : } \forall X : A : \text{var } X \to \exp \text{int } \to \var X
\end{align*}
\]

\[
\begin{align*}
\text{Essential datums constants} & : n \in \text{int} \\
& + \{ \text{int, int} \}, \text{exp int } \to \text{exp int } \to \text{exp int, } = \{ \text{int, int} \}, \text{exp int } \to \text{exp int } \to \text{exp bool, true } \{ \text{bool} \}, \text{false } \{ \text{bool} \} \to \text{exp bool}
\end{align*}
\]

\[
\begin{align*}
\text{Array expressions} & : \text{var2exp : } \forall X : A : \text{var } X \to \exp X \\
& \text{d : } \{ \delta_0, \ldots, \delta_k \} : \exp \delta_0 \to \ldots \to \exp \delta_{k+1} \to \exp \delta_k \\
& \text{getexp : } \forall X : A : \exp X \to \exp X \\
& \text{subexp : } \forall X : A : \exp int \to \exp X \to \exp X \\
& \text{condexp : } \forall X : A : \exp bool \to \exp X \to \exp X \to \exp X \\
& \text{newexp : } \forall X : A : \exp \text{#X } \to \exp \text{var X } \to \text{comm} \to \exp X \\
& \text{dyn : } \exp \delta \to \exp \delta
\end{align*}
\]

\[
\begin{align*}
\text{Shape expressions} & : \{ \delta_0, \ldots, \delta_k \} : \exp \delta_0 \to \ldots \to \exp \delta_{k+1} \to \exp \delta_k \\
& \text{d : } \exp \#\delta \to \exp \#\delta \\
& \text{zro : } \forall X : A : \#X \to \#X \\
& \text{suc : } \forall X : A : \exp \text{size } \to \#\{X \} \to \#\{X \} \\
& \text{undim : } \forall X : A : \#\{X \} \to \#\{X \} \\
& \text{lead : } \forall X : A : \#\{X \} \to \exp \text{size} \\
& \text{pred : } \forall X : A : \#\{X \} \\
& \text{numdim : } \forall X : A : \#\{X \} \to \exp \text{size} \\
& \text{equal : } \forall X : A : \#\{X \} \to \#\{X \} \to \exp \text{fact}
\end{align*}
\]

\[
\begin{align*}
\text{Phrase polymorphic constants} & : \text{condsh : } \forall U : P : \exp \text{fact } \to U \to U \to U \\
& \text{primrec : } \forall U : P : \exp \text{size } \to U \to U \to \exp \text{size } \to U \\
& \text{error : } \forall U : P : U \\
& \text{shape : } \forall U : P : U \to \#U
\end{align*}
\]
example, if $x$ is a matrix then the variable $y$ given by $\text{sub } x \ i$ is a vector, and $y \ j$ is a zero-dimensional array, whose unique entry is named by applying get. We write

$$A[i_0, i_1, \ldots, i_n]$$

for get (sub $(\ldots \ (\text{sub } A\ i_0) \ldots)\ i_n$).

The *primitive variable arrays* are those whose construction only uses primitive expressions of integer type (see next paragraph) as indices. All others are *civilised array variables*.

Let $x$ be an occurrence of an array variable in a term. It is *assigned* if its immediate context is assign $x \ t$ and is *evaluated* if its immediate context is var2exp $x$ (see next paragraph).

**Datum constants** Datum constants are expressions $d(\delta) : \exp\ \delta$ and datum operations $d(\delta_0, \ldots, \delta_n : \exp\ \delta_0 \rightarrow \ldots \rightarrow \exp\ \delta_{n-1} \rightarrow \exp\ \delta_n$ used to represent datum values and operations. We will often write $d$ for $d(\delta_0, \ldots, \delta_n)$ when the choice of datum types is clear. Also binary operations may be written infix, e.g. $t_1 + t_2$ for $+ \ t_1 \ t_2$. The precise choice of operations does not materially affect the language design. Only those specifically required below are included in the figure.

**Array expressions** Each array variable $x$ has a value given by the expression var2exp $x$ also written as $\text{var} x$. Datum constants may be used to construct array expressions. getexp and subexp are expression analogues of get and sub. The conditional expression condexp $b_1 \ t_1 \ t_2$ or

$$\text{if}\ b \ \text{then}\ t_1 \ \text{else}\ t_2$$

evaluates $t_1$ if $b$ is true, and $t_2$ if $b$ is false. The needs of shape analysis impose a side-condition on the formation of such terms: both $t_1$ must have the same shape, which is then the inferred shape of the whole expression. The expression block newexp $sh \ f$ or

$$\text{new } #x = sh \ \text{in} \ f \ x \ \text{return} \ x$$

is like a command block except that the value of the local variable is returned before $x$ is deallocated.

The constants var2exp and datum constants (both expressions and functions) are called *primitive data constants*. Expressions built from these terms, term variables of type $\exp\ \delta$ and primitive array variables are called *primitive expressions*. The constants getexp, subexp, newexp and condexp are the *civilised expression constants*.

For each datum type $\delta$ there is a coercion from static to dynamic values:

$$\text{dyn}() : \exp\ \delta \rightarrow \exp\ \delta.$$

**Shape expressions** Every datum constant $d$ has a corresponding shape constant $\text{sh } d$. For example $\text{sh } +$ is addition on sizes. Every value of datum type $\delta$ has the same shape, given by $\text{shape } \exp\ #d$. For example, every integer $n$ has shape intsh. Note that, by contrast, the shape of $n$ is $\text{sh } n$. That is, values of shape type are their own shape.

There are six constants which manipulate array shapes. zero dim $sh$ is a constructor that takes an array shape $sh$ as argument, and returns the shape of a 0-dimensional array whose sole entry has shape $sh$. succdim is a constructor that takes a size $\text{sh } n$ and an array shape $sh$, and returns another array shape, of one higher dimension, whose size in that dimension is $n$ and whose subarrays all have shape $sh$. For example, succdim $\text{sh } 3$ intsh is the shape of a vector of integers of length three. A more convenient syntax for array shapes represents zero dim by a colon and succdim by a comma separated list of integers, enclosed in braces. For example, $\{2, 3 : \text{intsh}\}$ denotes

$$\text{succdim } 2 \ (\text{succdim } 3 \ (\text{zerodim intsh}))$$

which is the shape of a $2 \times 3$ matrix of integers.

undim is a selector corresponding to zerodim. Similarly, lendim and preddim are selectors corresponding to succdim. Finally, the selector numdim determines the number of dimensions of an array, e.g. numdim $\{\text{~2, ~3 : intsh}\}$ reduces to $\text{~2}$. The remaining constant in this group is equal which checks for equality of shapes, returning a fact. We may write equal $x y$ as $x = y$.

It will be useful below to distinguish the shape constructors $\{\text{zero dim}, \text{zerodim}, \text{succdim}\}$ from the shape destructors, $\{\text{sh zero dim}, \text{sh zerodim}, \text{sh succdim}\}$ undim, lendim, preddim and equal. Terms constructed solely from shape constructors are called *shape values*.

**Phrase polymorphic terms** The shape conditional condsh $b\ t_0 \ t_1$ or

$$\text{if}\ b \ \text{then} \ t_0 \ \text{else} \ t_1$$

branches according to the value of the fact $b$. As the value of $b$ will be known during shape analysis, there is no requirement for the branches to have the same shape, as occurs in a shape conditional. The syntactic sugar

$$\text{check}\ b\ t$$

stands for condsh $b\ t\ \text{error}$. It is used extensively during shape analysis to check for errors.

**Primrec $f\ x\ t$** represents primitive recursion. If $t$ is $\text{~n}$ then this primitive recursion unwinds to

$$f\ n\ (f\ (\text{~}(\text{~}(n-1))(\ldots\ (f\ 0\ x)\ldots)))$$

The term $\text{~n}$ represents shape errors, which result from, say, attempting to multiply matrices whose shapes don't match. The constant shape or $#$ returns the shape of its argument.

We will abuse notation and allow a data type to stand for the corresponding expression type whenever the context makes this clear. Thus, we have $3 : \text{int}$ for $3 : \text{exp}\ \text{int}$. Also, references to polymorphic constants will always refer to phrase polymorphic constants rather than data polymorphic ones.

### 3 Partial Evaluation

A $\text{FISh}$ program is a closed term of type comm. (Array expressions can be converted to commands by applying output : exp $\rightarrow$ comm.) Static shape analysis reduces $\text{FISh}$ programs to programs constructed in a simple sub-language, called $\text{Turbot}$, whose raw syntax of terms is given
by

where term variables \( x \) must be of type \( \exp \int \exp \var \alpha \lor \text{comm} \). Turbot evaluation is given by a structured, or big-

Note that Turbot does not support functions of higher

typedef v variables / Turbot evaluation is given by a structured, or big-
appearance and further optimisations in which commands are treated as store-transformers.

The key goal is to compute all shapes, which necessarily

The rules for eliminating phrase polymorphic constants

shows that an assignment is well-shaped if both sides have

This rule reflects the requirement that both branches of an

expression conditional must have the same shape. This constraint

By unwinding all primitive recursions, we run the risk of

code explosion \cite{JW96,AS97}. FISH avoids most of the
disadvantages by promoting the use of for-and while-loops,
in which code only appears once, the use of local variables

whose initialisation is eager, and optimisations which eliminate

unnecessary copying of data structures.

The rules for eliminating phrase polymorphic constants

are given in Figure 6. This includes all explicit shape computa-

tions, resolving all shape conditionals and unwinding all

primitive recursion. There is no space here to go discuss all

of the explicit shape computations but let us consider two of

the most interesting. The reduction

This is of datum type, e.g., is an integer, then the returned
value can be stored in a register, but if it is a proper array
then it is not clear where to put its value. The solution is
to expand the scope of \( y \) to contain the assignment, as in

Note that there is no return value now, as indicated by the
keyword end. Often there is a more efficient solution, as shown in Section 4.2.

After partial evaluation of a FISH program, the shape of
resulting Turbot program is computed to check for shape
errors, e.g., assigning an array of the wrong shape.

Shape analysis has some novel characteristics compared to
standard partial evaluation techniques, e.g., \cite{JG93}, as its
techniques all derive from a single semantic insight. In this it is more like the parametrized partial evaluation described by Consel and Khou \cite{CK93} but requires even less intervention by the programmer. A fortiori, it can also be seen as a form of staged evaluation \cite{ST97}. In FISH, however, the distinction between static and dynamic is based on the division between shape and data rather than an analysis of the properties of the particular program at hand. Also, it is able to work with partial information about a single input, e.g., the length of a vector, as well specialising with respect to total information about some inputs. Thus, shape analysis can be fully automatic, without requiring selection of variables to be handled statically, or code re-organisation. Nevertheless, a significant fraction of variables suitable for static treatment are either of datum type, or describe shapes of data structures, and so can be handled in FISH.

4 Examples

Now let us consider the impact of partial evaluation on pro-
gram performance. The examples will illustrate the three
effects listed in the introduction, namely, unboxing, array
expressions, and explicit use of shapes.

4.1 Unboxed data: quicksort

Polymorphism is usually handled by boxing the data, i.e., by
using pointers. Shape analysis determines the shape of the arguments statically, so that all data can be unboxed. Let us consider quicksort, as it is one of the few standard C library functions that is polymorphic, so that comparison becomes possible. A FISH program for polymorphic quicksort, of type

\[
\text{quicksort} : (a \rightarrow a \rightarrow \text{bool}) \rightarrow [a] \rightarrow [a]
\]
The array type \( a \) can be instantiated to be any datum type, or nested array type. Nevertheless, comparisons are always made directly using the array entries.

By contrast, C’s standard polymorphic quicksort function \( qsort \) uses pointers and typecasts to control polymorphism. An example comparison function for floats is

\[
\text{int \ cmp(const void \ *i, \ const void \ *j) \{}
\text{int \ res;}
\text{if \ (*double*)i - (*double*)j > 0.0 \}
\text{res = 1;}
\text{else \{res = -1;\}}
\text{return \ res; \}}
\]

Figure 2 shows user times for quicksort on a random array of 200,000 FISH floats (C doubles). Two kinds of program are tested. Monomorphic programs are specialised to handle floats, while polymorphic programs must be able to work with arbitrary data types and comparison functions. For C, the standard \( qsort \) function was used in the polymorphic case. This function achieves polymorphism by using pointers to locate array entries, and then de-referencing them to make the comparison. All of this creates longer, more complex programs, and also slows down execution by a factor of three. Similar problems are likely with the OCAML polymorphic program. FISH avoids pointer manipulations through shape analysis (and performs function inlining) so that the polymorphic program is as fast as its monomorphic one, twice as fast as \( qsort \), and over six times faster than OCAML. This, in turn, is significantly faster than a corresponding Haskell program. Details of the experimental technique are given in Section 5.

<table>
<thead>
<tr>
<th></th>
<th>OCAML</th>
<th>C</th>
<th>FISH</th>
</tr>
</thead>
<tbody>
<tr>
<td>poly</td>
<td>9.04</td>
<td>3.59</td>
<td>1.69</td>
</tr>
<tr>
<td>mono</td>
<td>2.22</td>
<td>1.29</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 2: User times (seconds) for quicksort (polymorphic and monomorphic) on a random array of 200,000 floats (doubles)

4.2 Array expressions: mapping

FISH supports both array variables (which can be assigned) and array values. This is counter to the approach in most programming languages, where all arrays are assignable, this being their raison d’être. This added flexibility allows us to introduce additional optimisations on array expressions.

Consider an assignment to an array variable \( x \)

\[
x := e
\]

If \( e \) is the value \( y \) of some other array variable \( y \) then a bulk copy of memory (e.g. memcpy in C) is the simplest approach. This is safe because shape analysis guarantees that \( x \) and \( y \) have the same shape. A more frequent occurrence is that \( e \) is given by an expression block \( \text{new \#y = sh in } C \) return \ y in which \( x \) does not appear free in \( C \). Then there is no need to create the local variable \( y \) at all, merely to copy its result to \( x \). Rather we can use \( x \) directly. The resulting optimisation

\[
\text{let quicksort_pr (cmp: exp a -> exp a -> bool) (array: var [a]) =}
\text{let rec qs m n =}
\text{if m=n then skip}
\text{else}
\text{new pivot = array[(m + n) div 2]}
\text{and i = m}
\text{and j = n in}
\text{while cmp array[i] pivot do incr i done;}
\text{while cmp pivot array[j] do decr j done;}
\text{while i < j do}
\text{new aux = array[i] in}
\text{array[i] := array[j];}
\text{array[j] := aux}
\text{end;}
\text{incr i; decr j;}
\text{while cmp array[i] pivot do incr i done;}
\text{while cmp pivot array[j] do decr j done done;}
\text{(if j=i then incr i; decr j else skip);}
\text{qs m (!i); qs (!i) n end}
\text{in qs 0 (lendim \#array -1) ;;}
\text{let quicksort cmp arg =}
\text{new aux = arg in}
\text{quicksort_pr cmp aux return aux ;;}
\]

is thus

\[
x := \text{newexp \ sh f > check (#x # = sh) (f x)}
\]

\[
\text{if } f(x) \cap f(v) = \emptyset.
\]

In words, if \( x \) and the expression block have the same shape, and \( x \) is not free in the body of the block then use it as the local variable.

Although this optimisation looks fairly trivial, its correctness is dependent on a number of design features that are unique to FISH. (Previous Algol-like languages have not supported array data types.) First, the ability to manipulate whole arrays in this way, without using pointers into a heap, depends on shape analysis to ensure that copying occurs between structures of equal size and shape. Second, the check that \( x \) is not free in \( C \) would be inadequate if aliasing were allowed [Rev78, Rey89].

This optimisation eliminates many of the space leaks that confront implementers of functional languages, while maintaining a high degree of referential transparency in the source code (using \text{newexp}). The effect can be illustrated by looking at the action of polymorphic mapping

\[
\text{map : (a -> b) -> [a] -> [b]}
\]

on an expression block.

\[
\text{map is defined in the standard Prelude for FISH and was explained in detail in [JS98] as a canonical application of the FISH slogan. It is defined as}
\]

\[
\text{proc2fun map_pr map_sh}
\]

When applied to a function \( f \) and an array expression \( e \) a local variable of shape \( \text{map_sh \#f \#e} \) is created and then the
procedure map_pr f is used to assign appropriate values to its entries. Rather than review the details of the construction let us consider an example, and see the effect of the optimisation on the resulting C code.

Here is a short FISh session. The fill ... with ... syntax allows one to build an array from its shape and a list of its entries.

Figure 4: Unoptimised C code generated for mapping

```c
#define include <math.h>
#define include <stdio.h>
#define include <stdlib.h>
#define include <string.h>
#define include <sys/stat.h>
#include "fish.h"
#include "fish.h"

int main(int argc, char * argv[])
{
    argv = argv[0];
    argc = argc[0];
    int A[2][3];
    int B[2][3];
    int C[2][3];
    C[0][0] = 0; C[0][1] = 1; C[0][2] = 2;
    C[1][1] = 4; C[1][2] = 5;
    memcpy(B,C,sizeof(B));
}
int i;
for (i = 0; i < 2; i++)
{
    for (j = 0; j < 3; j++)
    {
        A[i][j] = (2*B[i][j]);
    }
}
}

let mat = fill (2,3:int_shape) with [0,1,2,3,4,5];
let f x = 2*x;
%show - assign_opt;;
let mat2 = map f mat;;
%show + assign_opt;;
let mat3 = map f mat;;
let mat4 = selfmap f mat;;
%run mat2;;
%run mat3;;
%run mat4;;

In each case the output is the same, namely

```c
fill { 2,3 : int_shape }
with [
    0,2,4,
    6,8,10 ]
```

However, the first program has the assignment optimisation switched off, and so uses three local variables. The C program generated by the FISh compiler for mat2 is given in Figure 4. The variable C is used to construct mat which is then copied to B. B holds the input to the mapping, whose result is stored in the variable A representing mat2. Note that the program for map has been written to ensure that the computation of mat is only performed once, outside the for-loops. Note, too that memcpy is used to copy C to B. This is perfectly safe as the shape analyser has already checked that the two variables have the same shape.

Of course, this copying is unnecessary, and is eliminated by the optimisation applied to the program for mat3. Its C code only has two local variables A and B representing mat3 and mat respectively, with the central assignment being

```c
A[i][j] = (2*B[i][j]);
```

Of course, one can object that a single variable should suffice, since the shapes of mat and mat3 are the same. This can be achieved by using

```c
selfmap : (α → α) → [α] → [α]
```

If the result has the same shape as its input then it may store the result in the same location as the argument. This is the case in our example, where selfmap is used to define mat4 whose central assignment is

```c
A[i][j] = (2*B[i][j]);
```

Unfortunately, the current version of FISh does not allow the type of selfmap to be generalised to that of map (whose function argument may produce a result of different type) as the test for shape equality requires arguments of the same type. This should be generalised in future.

4.3 Shape-based optimisation: reduction versus folding

Shape analysis allows us to customise algorithms during compilation according to the shapes that arise even though the source code is fully polymorphic. For example, operations such as summing or taking the product of a list or array of numbers can be defined as a reduction using a primitive binary operations, e.g. addition or multiplication. An efficient algorithm uses a single auxiliary variable to hold all of the intermediate values. This is safe because all of the intermediate values have the same shape. Reduction is often identified with the polymorphic operation of folding of type

```c
(a → b → a) → a → [b] → a
```

However, for general data types the intermediate values of type α may have different shapes, e.g. be arrays of different lengths, so that one is forced to create fresh storage for each intermediate value. The FISh standard prelude supports both reduce and fold on arrays. The latter is implemented as reduce if all of the intermediate values have the same shape, but will create multiple storage locations on those rare occasions when it is necessary to do so. Here is a fragment of the code for fold taken from the FISh standard prelude.

```c
let fold f x y =
    if #f #x (zeroShape #y) #= #x
    then reduce f x y
    else ...
```
The shape conditional tests whether the shape of $f$ applied to the shape of the auxiliary variable and the common shape of the array entries is the same as that of the auxiliary.

If the array types involved are actually datum types, e.g., int, then the type determines the shape; and so reduction (or quicksort) can be specialised without recourse to shape analysis, as in TIL [HM95]. However, the approach given here works for all data types, not just the datum types. For example, to add the columns of a matrix may be given as fold (zipop plus). Type analysis would not allow any simplification, but shape analysis allows this to become a reduction.

5 Benchmarks

This section compares the run-times speed of compiled FISH programs with a number of other polymorphic languages for several array-based problems, especially OCAML, which is one of the best of such other languages. 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 -0 flag and all floating-point variables of type double (64 bits). For OCAML code, we used ocamlc, the native-code compiler, from the 1.07 distribution, using the flag - unsafe (eliminating array bounds checks), and also -inline 100, to enable any in-lining opportunities. OCAML also uses 64 bit floats.

As in [JS98] the times for FISH are often faster than OCAML, usually at least twice as fast, and sometimes significantly better than that. The results are summarised in Figure 5. Note, however, that OCAML requires all arrays to be initialised, while FISH does not.

We timed four 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, and quicksort of a floating-point array. None of the benchmarks includes I/O, in order to focus comparison on array computation.

Matrix multiplication used two different algorithms, here called “loops” and “semi-combinatory” (code omitted). The loops algorithm uses an assignment within three nested for-loops. This algorithm is the usual one written in an imperative language. The semi-combinatory algorithm closely follows the usual definition of matrix multiplication, with a double-nested for-loop containing an inner-product.

6 Conclusions

This paper has shown how knowledge of shapes supports a combination of higher-order polymorphic programming with efficient, imperative implementations. In particular, knowledge of shapes during compilation supports a wide range of program optimisations, such as unboxing of data, re-use of local variables and explicit uses of shape. These techniques all constitute a form of partial evaluation, but they emerge out of a single semantic approach, rather than being adapted to individual programs.

In particular, it is not necessary for the user to determine which inputs should be static and which dynamic, as this is determined from general principles. Where user intuition can yield further benefits, this can often be captured within the programming constructs of the language itself, as occurs in the conversion of fold into reduce, rather than by annotations.

All of the work described here has been implemented, with the source code made publically available, and is supported by a formal definition.

Current work is proceeding in two directions. One is to combine the ideas of FISH with those of Functional ML [JBM98] to create a language that supports both array types and inductive data types. In developing this, many of the idiosyncrasies of the FISH language appear to be falling away, leaving a simpler programming language but a more complicated semantics. If successful, this program may also reduce the distance between FISH and other, better known, programming languages, so that shape ideas could be incorporated within them.

The other development is that of a portable parallel version of FISH called Goldfish [JCSS97, Jay8a]. It will use shape analysis to guide data distribution and support a static cost model.

There are also many opportunities for further partial evaluation and optimisation based on shape information, e.g. the further elimination of dynamic array bound checks.

Overall, the FISH language demonstrates in concrete terms the benefits that can be extracted by incorporating shape ideas into the computational framework.

Acknowledgements

I would like to thank Neil Jones for his constructive criticism and support, and the anonymous referees and Gabriele Keller for helping me clarify these ideas.

References


\textbf{Figure 6: FISH reductions: phrase polymorphic constants}

\begin{verbatim}
condsh `true > \lambda x, y. x
condsh `false > \lambda x, y. y
primrec f \equiv 0 > x
primrec f x
\neg (n + 1) > f \equiv n (\text{primrec } f x \equiv n)
error t > error
c t_0 \ldots t_{i-1} error > error for any combinator \(c\) except
  \text{condsh.} \(k \neq 0\) or \text{primrec.} \(j \neq 2\)
#x > sh if \(x = (\mathit{sh}, \theta)\)
#(t t_1) > #t #t_1
#(\lambda x.t_2) > \lambda y.t_3 if \#t_2 = \neg s, t_3 where \(x \notin \text{fv}(t_3)\)
  and \(\Gamma' = \Gamma, y : \#U, x : (y, U)\)
#skip > `true
#abort > `true
#assign > `equal
#seq > `=
#cond > \lambda x, y, z. \text{check} (equal \(x\) \(y\)) \text{check} \(y\) \(z\)
#forall > \lambda x, y, z. \text{check} (equal \(x\) \(y\)) \(z\) \text{int\_shape}
#whiletrue > \lambda x, y. \text{check} (equal \(x\) \(y\)) \(y\)
#fix > \lambda x. x \equiv true
#newvar > \lambda x, y, y x
#output > \lambda x. \text{check} (equal \(x\) \(y\)) \neg true
#get > \text{undim}
#sub > \lambda x, y. \text{check} (equal \(y\) \(x\)) \text{predim} \(x\)
#d{\delta_0, \ldots , \delta_k} > \lambda x_0. \text{check} (equal x_0 \ x_0) \ldots \lambda x_{k-1}. \text{check} (equal x_{k-1} \ x_{k-1}) \delta_1\text\_shape
#getexp > \text{undim}
#subexp > \lambda x, y. \text{check} (equal \(y\) \(x\)) \text{predim} \(x\)
#condexp > \lambda x, y, z. \text{check} (equal \(x\) \(y\)) \text{check} (equal \(y\) \(z\)) \(y\)
#newexp > \lambda x, y, z. \text{check} (equal \(x\) \(y\)) \(z\)
#dyn\{\delta\} > \lambda x. \text{check} (equal \(x\) \(\delta\text\_shape\)) \(\delta\text\_shape\)
#var2exp > \lambda x. x
#shape > \lambda x. x
#succdim > \lambda x. \text{check} (\neg x \equiv `0) \text{succdim} \(x\)
#c > c otherwise
\end{verbatim}

\textbf{Figure 7: FISH reductions: Beta and where}

\begin{verbatim}
(\lambda x.t) a > t[a/x]
t where x = a > t[a/x]
\end{verbatim}

\textbf{Figure 8: FISH reductions: shape expressions}

\begin{verbatim}
dyn\{\delta\} \neg d\{\delta\} > d\{\delta\}
\neg d\{\delta_0, \ldots , \delta_k\} \neg n_0 \ldots \neg n_{k-1} > \neg p\{\delta\} where \(p = d \equiv n_0 \ldots n_{k-1}\)
undim (zero dim t) > t
undim (succdim s t) > error
lendim (zero dim t) > error
lendim (succdim s t) > s
predim (zero dim t) > error
predim (succdim s t) > t
numdim (zero dim t) > `0
numdim (succdim s t) > numdim t \equiv `0
equal \(\delta\text\_shape\) \(\delta\text\_shape\) > `true
equal (zero dim t_0) (zero dim t_1) > equal t_0 t_1
equal (zero dim t_0) (succdim s_1 t_1) > \neg false
equal (succdim s_0 t_0) (zero dim t_1) > \neg false
equal (succdim s_0 t_0) (succdim s_1 t_1) > check \(s_0 \equiv s_1\) equal t_0 t_1
\end{verbatim}

\textbf{Figure 9: FISH reductions: shape contexts}

\begin{verbatim}
newvar sh \lambda x. \text{error} > \text{error}
forall t_2 t_3 \lambda x. \text{error} > \text{error}
fix \lambda x. \text{error} > \text{error}
newexp sh \lambda x. \text{error} > \text{error}
\end{verbatim}

Let \(\Gamma' = \Gamma, x : (\mathit{sh}, \theta)\) and \(t_0 \rightarrow_{GT} t_1\).

\begin{verbatim}
newvar sh \lambda x. t_0 >_{GT} newvar sh \lambda x. t_1
\end{verbatim}

When \((\mathit{sh}, \theta) = (\neg true, \text{comm})\)

\begin{verbatim}
forall t_2 t_3 \lambda x. t_0 >_{GT} forall t_2 t_3 \lambda x. t_1
fix \lambda x. t_0 >_{GT} fix \lambda x. t_1
newexp sh \lambda x. t_0 >_{GT} newexp sh \lambda x. t_1
\end{verbatim}

\begin{verbatim}
When \((\mathit{sh}, \theta) = (\neg true, \text{comm})\)
newvar sh \lambda x. t_0 >_{GT} newvar sh \lambda x. t_1
\forall t_2 t_3 \lambda x. t_0 >_{GT} forall t_2 t_3 \lambda x. t_1
fix \lambda x. t_0 >_{GT} fix \lambda x. t_1
newexp sh \lambda x. t_0 >_{GT} newexp sh \lambda x. t_1
\end{verbatim}
Figure 10: \textbf{FISH} reductions: data reduction

\begin{align*}
\text{assign } t e & \quad \rightarrow \quad \text{vtc } (\lambda x. \text{assign } x e) \ t \\
\text{if } & \quad \rightarrow \quad \text{vtc } (\lambda x.\text{!}x) \ t \quad \text{if } \text{newexp} \text{ or } \text{condexp} \text{ in } t \\
\text{!t} & \quad \rightarrow \quad \text{vtc } (\lambda y. \text{!}y) \ t \\
\text{vtc } f y & \quad \rightarrow \quad f y \quad \text{if } y \text{ is a term variable} \\
\text{vtc } f (\text{get } t) & \quad \rightarrow \quad \text{vtc } (\lambda y. \text{f } \text{get } y) \ t \\
\text{vtc } f (\text{sub } t i) & \quad \rightarrow \quad \text{vtc } (\lambda y. \text{newvar int\_shape } \lambda j.
\quad j := i; f (\text{sub } y j)) \ t \\
\text{vte } f y & \quad \rightarrow \quad f y \quad \text{if } y \text{ is a term variable} \\
\text{vte } f (\text{get } t) & \quad \rightarrow \quad \text{vte } (\lambda y. \text{f } \text{get } y) \ t \\
\text{vte } f (\text{sub } t i) & \quad \rightarrow \quad \text{vte } (\lambda y. \text{newexp } (\# f \text{ (preddim } \# t))
\quad \lambda z. \text{newvar int\_shape } \lambda j.
\quad j := i; z := f (\text{sub } y j)) \ t \\
\text{getexp } !t & \quad \rightarrow \quad !(\text{get } t) \\
\text{subexp } !t_{1} t_{2} & \quad \rightarrow \quad !(\text{sub } t_{1} t_{2})
\end{align*}

Let \( g \) be a term and \( n \) be a natural number. If \((g, n)\) is one of \((\text{assign } t, 0)\), \((\text{cond } 2)\), \((\text{forall } 2)\), \((\text{forall } t, 1)\), \((\text{whiletrue } 1)\) or \((\text{output } 0)\) then

\begin{align*}
g \ (\text{newexp } sh \ f) t_{1} & \ldots t_{n} > \text{newvar } sh \ x_{0}.
\quad f x_{0}; g \ x_{0} t_{1} \ldots t_{n} \\
g \ (\text{condexp } s_{0} s_{1} s_{2}) t_{1} & \ldots t_{n} > \text{cond } s_{0} (g s_{1} t_{1} \ldots t_{n})
\quad (g s_{2} t_{1} \ldots t_{n})
\end{align*}

Let \( h \) be a term and \( n \) be a natural number. If \((h, n)\) is one of \( (d \{k_{0} \ldots k_{i} \} s_{0} \ldots s_{j}, k - 1 - j)\), \((\text{getexp } 0)\), \((\text{subexp } 1)\) or \((\text{subexp } s, 0)\) then

\begin{align*}
h \ (\text{newexp } sh \ f) t_{1} & \ldots t_{n} > \text{newexp } (\# h \ sh \ # t_{1} \ldots # t_{n})
\quad \lambda x. \text{newvar } sh \ x_{0}.
\quad f x_{0};
\quad x := h \ x_{0} t_{1} \ldots t_{n} \\
h \ (\text{condexp } s_{0} s_{1} s_{2}) t_{1} & \ldots t_{n} > \text{condexp } s_{0} (h s_{1} t_{1} \ldots t_{n})
\quad (h s_{2} t_{1} \ldots t_{n})
\end{align*}
Figure 5: Benchmark results.