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@socs.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 [Jay95] 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:

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

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 (`qsort` 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** [JS98, Jay98b] 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

$$\begin{array}{rcl}
\delta : \mathsf{D} & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{float} \mid \mathsf{char} \mid \ldots \\
\alpha : \mathsf{A} & ::= & X \mid \delta \mid [\alpha] \\
\sigma : \mathsf{Sh} & ::= & \tilde{}\delta \mid \#\alpha \\
\tau : \mathsf{T} & ::= & \alpha \mid \sigma \\
\theta : \mathsf{P} & ::= & U \mid \#U \mid \mathsf{comm} \mid \mathsf{var}\ \alpha \mid \mathsf{exp}\ \tau \mid \theta \to \theta \\
\phi : \mathsf{S} & ::= & \theta \mid \forall X : \mathsf{A}.\, \phi \mid \forall U : \mathsf{P}.\, \phi
\end{array}$$

Following after Reynold's account of Algol-like languages [Rey81, OT97] **FISh** distinguishes the *data types* (meta-variable τ), which represent storable values, from the *phrase types* (meta-variable θ), which represent meaningful program fragments. The data types are further divided into the *array types* (meta-variable α) which are used to store arrays of data, and the *shape types* or *static types* (meta-variable σ) 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 $X, Y : \mathsf{A}$), *datum types* (meta-variable δ) or *arrays* $[\alpha]$ of α. 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 α'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 [[float]] 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 δ has a corresponding shape type, called $\tilde{}\delta$, 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 = ˜int of sizes is used to represent the length, or size, of an array in a given dimension. The type fact = ˜bool is used for static booleans, or *facts*, which are useful for expressing properties of shapes required during compilation. The type cost = ˜float is used for static floats, useful for static cost analysis. The type mark = ˜char may be used for labels.

The other shape types are of the form $\#\alpha$ which represents the shapes of arrays of type α. 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 α 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 $\tilde{}\delta$ and $\#\delta$. The former type has many values, one for each value of type δ, representing sizes, facts, etc. The latter has only one value, representing the common shape of all δ-values. For example, $\tilde{}3 : \mathsf{size}$ compared to $\mathsf{int\_shape} : \#[\mathsf{int}]$.

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 $U : \mathsf{P}$) are used to express polymorphism. Each such has a *shape* $\#U$ (see below).

The type 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 α yields a type var α of *array variables* of type α. Terms of this type have mutable values. Each data type τ yields a type exp τ of *expressions* of type τ whose values are immutable. Array variables represent stored quantities, much as reference types do in ML.

Unlike earlier Algol-like languages, **FISh** supports storable arrays, as well as storable atoms. 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 ([Tof88] and also [OK93]) but in **FISh** all polymorphism is instantiated statically, before execution.

The *function type* $\theta_1 \to \theta_2$ represents functions from θ_1 to θ_2. When $\theta_2 = \mathsf{comm}$ 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 unknown. 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 θ has an associated phrase type shp θ (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 [int] $\to$ exp $[\alpha]$ 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 ~true.

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

2.2 Terms

The raw syntax for **FISh** terms is the same as that for the Hindley-Milner type system:

$$t ::= x \mid c \mid \lambda x.t \mid t\ t \mid t \text{ where } x = t$$

except that where-expressions are preferred over let-expressions as evaluation will be call-by-name, not by value. x and y range over term variables. c ranges over constants. Type inference follows a modified version of the standard algorithm W.

The **FISh** constants are given in Figure 1. They are arranged in the families, according to the kind of their result type. Binary datum operations are written infix.

Commands skip is the command that does nothing. abort terminates computation. assign $x\ t$ or $x := t$ updates the value of the array variable x to be t. The command seq $C_0\ C_1$ or $C_0; C_1$ performs the command C_0 and then C_1. The command cond $b\ C_0\ C_1$ or

$$\text{if } b \text{ then } C_0 \text{ else } C_1$$

is a conditional, branching according to the value of the boolean expression b. The for-loop forall $m\ n\ f$ or

$$\text{for } m \le i < n \text{ do } f\ i \text{ done}$$

iterates the command $f\ i$ as i ranges over the integers from m to $n-1$. Similarly, whiletrue $b\ C$ or

$$\text{while } b \text{ do } C \text{ done}$$

is a while loop. fix is a fixpoint constructor for the command type. The *command block* newvar $sh\ f$ or

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

introduces a local variable x of shape sh, executes the command $f\ x$ and then de-allocates x. Note that while it is necessary to supply the shape of a newly declared variable, it is not necessary to initialise its entries. output takes an array expression and returns a command. Its intended action is to compute the value of the expression, and output the result as a side effect.

Figure 1: **FISh** Constants

Commands

skip	:	comm
abort	:	comm
assign	:	$\forall X : \mathsf{A}.$ var $X \to$ exp $X \to$ comm
seq	:	comm $\to$ comm $\to$ comm
cond	:	exp bool $\to$ comm $\to$ comm $\to$ comm
forall	:	exp int $\to$ exp int $\to$ (exp int $\to$ comm) $\to$ comm
whiletrue	:	exp bool $\to$ comm $\to$ comm
fix	:	(comm $\to$ comm) $\to$ comm
newvar	:	$\forall X : \mathsf{A}.$ exp $\#X \to$ (var $X \to$ comm) $\to$ comm
output	:	$\forall X : \mathsf{A}.$ exp $X \to$ comm

Array variables

get	:	$\forall X : \mathsf{A}.$ var $[X] \to$ var X
sub	:	$\forall X : \mathsf{A}.$ var $[X] \to$ exp int $\to$ var $[X]$

Essential datum constants

n\{int\}	:	exp int for every integer n
$+$\{int, int, int\}	:	exp int $\to$ exp int $\to$ exp int
$=$\{int, int, int\}	:	exp int $\to$ exp int $\to$ exp bool
true\{bool\}, false\{bool\}	:	exp bool

Array expressions

var2exp	:	$\forall X : \mathsf{A}.$ var $X \to$ exp X
$d\{\delta_0, \ldots, \delta_k\}$	:	exp $\delta_0 \to \ldots \to$ exp $\delta_{k-1} \to$ exp δ_k
getexp	:	$\forall X : \mathsf{A}.$ exp $[X] \to$ exp X
subexp	:	$\forall X : \mathsf{A}.$ exp int $\to$ exp $[X] \to$ exp $[X]$
condexp	:	$\forall X : \mathsf{A}.$ exp bool $\to$ exp $X \to$ exp $X \to$ exp X
newexp	:	$\forall X : \mathsf{A}.$ exp $\#X \to$ (var $X \to$ comm) $\to$ exp X
dyn$\{\delta\}$	:	exp $\tilde{}\delta \to$ exp δ

Shape expressions

$\tilde{}d\{\delta_0, \ldots, \delta_k\}$	:	exp $\tilde{}\delta_0 \to \ldots \to$ exp $\tilde{}\delta_{k-1} \to$ exp $\tilde{}\delta_k$
δ_shape	:	exp $\#\delta$
zerodim	:	$\forall X : \mathsf{A}.\ \#X \to \#[X]$
succdim	:	$\forall X : \mathsf{A}.$ exp size $\to \#[X] \to \#[X]$
undim	:	$\forall X : \mathsf{A}.\ \#[X] \to \#X$
lendim	:	$\forall X : \mathsf{A}.\ \#[X] \to$ exp size
preddim	:	$\forall X : \mathsf{A}.\ \#[X] \to \#[X]$
numdim	:	$\forall X : \mathsf{A}.\ \#[X] \to$ exp size
equal	:	$\forall X : \mathsf{A}.\ \#X \to \#X \to$ exp fact

Phrase polymorphic constants

condsh	:	$\forall U : \mathsf{P}.$ exp fact $\to U \to U \to U$
primrec	:	$\forall U : \mathsf{P}.$ (exp size $\to U \to U) \to U \to$ exp size $\to U$
error	:	$\forall U : \mathsf{P}.\ U$
shape	:	$\forall U : \mathsf{P}.\ U \to \#U$

Array variables The unique entry of a zero-dimensional array is named by get. Similarly, sub $x\ i$ names the variable which is the ith subarray of x (i is the *index*). For example, if x is a matrix then the variable y given by $\mathsf{sub}\ x\ i$ is a vector, and $\mathsf{sub}\ y\ j$ is a zero-dimensional array, whose unique entry is named by applying get. We write

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

for $\mathsf{get}\ (\mathsf{sub}\ (\ldots(\mathsf{sub}\ A\ i_0)\ldots i_k))$.

The *primitive array variables* 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 $\mathsf{assign}\ x\ t$ and is *evaluated* if its immediate context is $\mathsf{var2exp}\ x$ (see next paragraph).

Datum constants Datum constants are expressions $d\{\delta\} : \mathsf{exp}\ \delta$ and datum operations $d\{\delta_0, \ldots, \delta_k\} : \mathsf{exp}\ \delta_0 \to \ldots \to \mathsf{exp}\ \delta_{k-1} \to \mathsf{exp}\ \delta_k$ used to represent datum values and operations . We will often write d for $d\{\delta_0, \ldots, \delta_k\}$ 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 $\mathsf{var2exp}\ x$ also written as $!x$. Datum constants may be used to construct array expressions. getexp and subexp are expression analogues of get and sub. The conditional expression $\mathsf{condexp}\ b\ t_1\ t_2$ or

$$\mathsf{ife}\ b\ \ \mathsf{then}\ \ t_1\ \ \mathsf{else}\ \ t_2$$

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_i must have the same shape, which is then the inferred shape of the whole expression. The *expression block* $\mathsf{newexp}\ sh\ f$ or

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

is like a command block except that the value of the local variable x is returned before x is de-allocated.

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

For each datum type δ there is a coercion from static to dynamic values:

$$\mathsf{dyn}\{\delta\} : \mathsf{exp}\ \tilde{}\delta \to \mathsf{exp}\ \delta.$$

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

There are six constants which manipulate array shapes. $\mathsf{zerodim}\ sh$ 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 $\tilde{}n$ and an array shape sh, and returns another array shape, of one higher dimension, whose size in that dimension is $\tilde{}n$ and whose subarrays all have shape sh. For example, $\mathsf{succdim}\ \tilde{}3\ \mathsf{int\_shape}$ is the shape of a vector of integers of length three. A more convenient syntax for array shapes represents zerodim by a colon and succdim's by a comma separated list of integers, enclosed in braces. For example, $\{2, 3 : \mathsf{int\_shape}\}$ denotes

$$\mathsf{succdim}\ \tilde{}2\ (\mathsf{succdim}\ \tilde{}3\ (\mathsf{zerodim}\ \mathsf{int\_shape}))$$

which is the shape of a 2×3 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. $\mathsf{numdim}\ \{\tilde{}2, \tilde{}3 : \mathsf{int\_shape}\}$ reduces to $\tilde{}2$. The remaining constant in this group is equal which checks for equality of shapes, returning a fact. We may write $\mathsf{equal}\ x\ y$ as $x \# = y$.

It will be useful below to distinguish the *shape constructors* $\tilde{}d\{\delta\}$, zerodim and succdim from the *shape destructors*, $\tilde{}d\{\delta_0, \ldots, \delta_k\}$ undim, lendim, preddim and equal. Terms constructed solely from shape constructors are called *shape values.*

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

$$\mathsf{ifsh}\ b\ \ \mathsf{then}\ \ t_0\ \ \mathsf{else}\ \ t_1$$

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

$$\mathsf{check}\ b\ t$$

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

$$\mathsf{primrec}\ f\ x\ t$$

represents primitive recursion. If t is $\tilde{}n$ then this primitive recursion unwinds to

$$f\ \tilde{}n\ (f\ \tilde{}(n-1)(\ \ldots\ (f\ \tilde{}0\ x)\ldots))\ .$$

The term error 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 $3 : \mathsf{int}$ for $3 : \mathsf{exp\ int}$. Also, references to polymorphic constants will always refer to phrase polymorphic constants rather than data polymorphic ones.

3 Partial Evaluation

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

$$
\begin{array}{rcl}
t & ::= & x \mid \mathsf{skip} \mid \mathsf{abort} \mid \mathsf{assign}\ t_0\ t_1 \mid \mathsf{seq}\ t_0\ t_1 \mid \\
 & & \mathsf{cond}\ t_0\ t_1\ t_2 \mid \mathsf{forall}\ t_0\ t_1\ \lambda x.t_2 \mid \mathsf{whiletrue}\ t_0\ t_1 \mid \\
 & & \mathsf{fix}\ \lambda x.t \mid \mathsf{newvar}\ t_0\ \lambda x.t_1 \mid \mathsf{output}\ t \mid \\
 & & \mathsf{get}\ t \mid \mathsf{sub}\ t_0\ t_1 \mid !t \mid d\{\delta_0 \ldots\ \delta_k\}\ t_0\ t_1\ \ldots\ t_{k-1} \mid \\
 & & \tilde{\ } d\{\delta\} \mid \delta\_\mathsf{shape} \mid \mathsf{zerodim}\ t \mid \mathsf{succdim}\ t_0\ t_1
\end{array}
$$

where term variables x must be of type $\mathsf{exp\ int}, \mathsf{var}\ \alpha$ or comm. Turbot evaluation is given by a structured, or big-step operational semantics in which commands are treated as store-transformers.

Note that **Turbot** does not support functions of higher type or phrase polymorphic constants, and expressions are limited to shape constructors and primitive data constants. The other functions and constants must be eliminated by partial evaluation. This is achieved by the reduction rules given in Figures 6 – 10. This section will survey the rules with examples of their application and further optimisations in the following section. A more detailed account can be found in the language definition.

The key goal is to compute all shapes, which necessarily involves evaluation of shape functions, i.e. beta-reduction. Rather than try to separate out the shape functions for special treatment, **FISh** in-lines all non-recursive function calls statically (Figure 7). Usually, in-lining is a mixed blessing with the benefits of closure elimination offset by the cost of code explosion [JW96, Ash97]. **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 computations, resolving all shape conditionals and unwinding all primitive recursion. There is not space here to go discuss all of the explicit shape computations but let us consider two of the most interesting. The reduction

$$\#(x := t) \to^* \#x\ \#=\ \#t$$

shows that an assignment is well-shaped if both sides have the same shape. Many array-bound errors are caused by failures of this condition.

$$
\begin{gathered}
\#(\mathsf{ife}\ b\ \mathsf{then}\ x\ \ \mathsf{else}\ \ y) \to^* \\
\mathsf{check}\ (\#b\ \#=\ \#b)\ \mathsf{check}\ (\#x\ \#=\ \#y)\ \#x
\end{gathered}
$$

This rule reflects the requirement that both branches of an expression conditional must have the same shape. This constraint on conditionals is necessary for effective static shape analysis. Where the branches have different shapes the programmer is required to supply a condition that can be evaluated statically, using a shape conditional.

By unwinding all primitive recursions, we run the risk of code explosion, but its typical use is for supporting polydimensional programming, where recursion over the number of array dimensions is required, so that the number of iterations is usually no more than three.

Figure 8 gives rules for computing static quantities of datum type, and elimination rules for array shape destructors.

Figure 9 addresses the issue of shaping local variables. Their shape is known at creation, and these rules allow this information to be used when simplifying the body of the block, even though it contains free variables. That is, the context is allowed to carry shape information as well as type information about local variables.

Figure 10 describes reductions for simplifying array expressions. The first eight rules involve auxiliary functions vtc and vte which are used to handle local variables that appear within array indices. They are included here for completeness but will not affect the further discussion. The final four rules are used to convert expression conditionals and blocks to their command forms. Typical is the following assignment to an array variable x

```
x := new #y = sh in C return y
```

If `y` 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

```
new #y = sh in C ; x:= !y end
```

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. [JGS93], as its techniques all derive from a single semantic insight. In this it is more like the parametrized partial evaluation described by Consel and Khoo [CK93] but requires even less intervention by the programmer. A fortiori, it can also be seen as a form of staged evaluation [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 program performance. We will consider examples that illustrate the first 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

```
quicksort : (a -> a -> bool) -> [a] -> [a]
```

is given in Figure 3 (the `let rec` syntax is a sugared form of fix). 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

```
int cmp(const void *i, const void *j) {
  int res ;
  if (*(double*)i - *(double*)j > 0.0 )
  {res = 1 ;}
  else {res = -1 ;}
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.

	OCAML	C	**FISh**
poly	9.04	3.59	1.69
mono	2.22	1.29	–

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'etre. 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 $\mathsf{new}\ \#y = sh\ \mathsf{in}\ C\ \mathsf{return}\ \ y$ 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

Figure 3: quicksort.fsh

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

is thus

$$x := \mathsf{newexp}\ sh\ f \qquad > \qquad \mathsf{check}\ (\#x\ \# = sh)\ (f\ x) \quad \text{if } fv(x) \cap fv(f) = \{\}.$$

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 [Rey78, 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 `newexp`). The effect can be illustrated by looking at the action of polymorphic mapping

```
map : (a -> b) -> [a] -> [b]
```

on an expression block.

`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

```
proc2fun map_pr map_sh
```

When applied to a function `f` and an array expression `e` a local variable of shape `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

```
   /* translated by fish */

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

int _argc;
char **_argv;

int main(int argc,char *argv[]) {
   _argc = argc;
   _argv = argv;

   { 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][0] = 3;
         C[1][1] = 4; C[1][2] = 5;
         memcpy(B,C,sizeof(B));
       }
       { int i;
         for (i = 0; i < 2; i++) {
           { int j;
             for (j = 0; j < 3; j++) {
               A[i][j] = (2*B[i][j]);
   }}}}}
   fish_print(_argc,_argv,INT_SHAPE,2,3,
   ARRAY_BOUNDARY,END_OF_SHAPE,(char *)A);
   }
   return 0;
}
```

```
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

```
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

```
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

$$\texttt{selfmap} : (\alpha \to \alpha) \to [\alpha] \to [\alpha]$$

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 out example, where `selfmap` is used to define `mat4` whose central assignment is

```
A[i][j] = (2*A[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

$$(a \to b \to a) \to a \to [b] \to a$$

However, for general data types the intermediate values of type a 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.

```
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-time 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 `-O` flag and all floating-point variables of type `double` (64 bits). For OCAML code, we used ocamlopt, the native-code compiler, from the 1.07 distribution, using the flag `-unsafe` (eliminating arrays 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 is able to support 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 supported by a formal definition.

Current work is proceeding in two directions. One is to combine the ideas of **FISh** with those of Functorial ML [JBM98] to create a language that supports both array types and inductive data types. In developing this, many of the idiosyncracies 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, Jay98a]. 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 Gabi Keller for helping me clarify these ideas.

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.

[BM97] G. Belle and E. Moggi. Typed intermediate languages for shape analysis. In P. de Groote and J. R. Hindley, editors, *Typed Lambda Calculi and Applications. Proceedings*, volume 1210 of *Lecture Notes in Computer Science*, pages 1–10. Springer Verlag, 1997.

[CK93] C. Consel and S.C. Khoo. Parametrized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.

[FISh] Fish web-site. http://www-staff.socs.uts.edu.au /~cbj/FISh.

[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.

[Jay94] C.B. Jay. Matrices, monads and the fast Fourier transform. In *Proceedings of the Massey Functional Programming Workshop 1994*, pages 71–80, 1994.

[Jay95] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[Jay98a] C.B. Jay. Costing parallel programs as a function of shapes. invited submission to Science of Computer Programming, September 1998.

[Jay98b] C.B. Jay. Poly-dimensional array programming. http://www-staff.socs.uts.edu.au/~cbj /Publications/polydimensional2.ps.gz, August 1998.

[Jay99] C.B. Jay. Denotational semantics of shape: Past, present and future. In A. Scedrov and A. Jung, editors, *Fifteenth Conference on the Mathematical Foundations of Programming Semantics (MFPS XV) Tulane University New Orleans, LA USA April 28 - May 1, 1999: Proceedings*, Electronic Notes in Computer Science. Elsevier, 1999.

[JBM98] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, to appear.

[JCSS97] C.B. Jay, M.I. Cole, M. Sekanina, and P.A. 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 International, 1993.

[JS98] C.B. Jay and P.A. Steckler. The functional imperative: shape! In Chris Hankin, editor, *Programming languages and systems: 7th European Symposium on Programming, ESOP'98 Held as part of the joint european conferences on theory and practice of software, ETAPS'98 Lisbon, Portugal, March/April 1998*, volume 1381 of *Lecture Notes in Computer Science*, pages 139–53. Springer Verlag, 1998.

[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.

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

[OK93] A. Ohori and K Kato. Semantics for communication primitives in a polymorphic language. In *POPL '93: The 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 99–112. ACM Press, 1993.

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

[Rey78] John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 39–46, 1978.

[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.

[Rey89] John C. Reynolds. Syntactic control of interference, part 2. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. Della Rocca, editors, *Automata, Languages and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 704–722, Berlin, 1989. Springer-Verlag.

[Sek98] M. Sekanina. *Shape Analysis.* PhD thesis, University of Technology, Sydney, 1998.

[ST97] T. Sheard and W. Taha. Multi-stage programming with explicit annotations. In *Proceedings of PEPM '97*, 1997.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1988. available as CST-52-88.

Figure 6: **FISh** reductions: phrase polymorphic constants

$$\begin{array}{rcl}
\mathsf{condsh}\ \tilde{}\mathsf{true} & > & \lambda x, y.\ x \\
\mathsf{condsh}\ \tilde{}\mathsf{false} & > & \lambda x, y.\ y \\
\mathsf{primrec}\ f\ x\ \tilde{}0 & > & x \\
\mathsf{primrec}\ f\ x & & \\
\tilde{}(n+1) & > & f\ \tilde{}n\ (\mathsf{primrec}\ f\ x\ \tilde{}n) \\
\mathsf{error}\ t & > & \mathsf{error} \\
c\ t_0\ ..\ t_{k-1}\ \mathsf{error} & > & \mathsf{error}\quad \text{for any combinator } c \text{ except} \\
 & & \mathsf{condsh}, k \neq 0 \text{ or } \mathsf{primrec}, j \neq 2 \\
\#x & > & sh \quad \text{if } \Gamma(x) = (sh, \theta) \\
\#(t\ t_1) & > & \#t\ \#t_1 \\
\#\ (\lambda x.t_2) & > & \lambda y.t_3 \text{ if } \#t_2 \rightarrow^*_{\Gamma'} t_3 \text{ where } x \notin fv(t_3) \\
 & & \text{and } \Gamma' = \Gamma, y : \#U, x : (y, U) \\
\#\mathsf{skip} & > & \tilde{}\mathsf{true} \\
\#\mathsf{abort} & > & \tilde{}\mathsf{true} \\
\#\mathsf{assign} & > & \mathsf{equal} \\
\#\mathsf{seq} & > & \tilde{}= \\
\#\mathsf{cond} & > & \lambda x, y, z.\ \mathsf{check}\ (\mathsf{equal}\ x\ x)\ \mathsf{check}\ y\ z \\
\#\mathsf{forall} & > & \lambda x, y, z.\ \mathsf{check}\ (\mathsf{equal}\ x\ y)\ z\ \mathsf{int_shape} \\
\#\mathsf{whiletrue} & > & \lambda x, y.\ \mathsf{check}\ (\mathsf{equal}\ x\ x)\ y \\
\#\mathsf{fix} & > & \lambda x.\ x\ \tilde{}\mathsf{true} \\
\#\mathsf{newvar} & > & \lambda x, y.\ y\ x \\
\#\mathsf{output} & > & \lambda x.\ \mathsf{check}\ (\mathsf{equal}\ x\ x)\ \tilde{}\mathsf{true} \\
\#\mathsf{get} & > & \mathsf{undim} \\
\#\mathsf{sub} & > & \lambda x, y.\ \mathsf{check}\ (\mathsf{equal}\ y\ y)\ \mathsf{preddim}\ x \\
\#d\{\delta_0, \ldots, \delta_k\} & > & \lambda x_0.\ \mathsf{check}\ (\mathsf{equal}\ x_0\ x_0)\ \ldots \\
 & & \lambda x_{k-1}.\ \mathsf{check}\ (\mathsf{equal}\ x_{k-1}\ x_{k-1}) \\
 & & \delta_k\mathsf{_shape} \\
\#\mathsf{getexp} & > & \mathsf{undim} \\
\#\mathsf{subexp} & > & \lambda x, y.\ \mathsf{check}\ (\mathsf{equal}\ y\ y)\ \mathsf{preddim}\ x \\
\#\mathsf{condexp} & > & \lambda x, y, z.\ \mathsf{check}\ (\mathsf{equal}\ x\ x) \\
 & & \mathsf{check}\ (\mathsf{equal}\ y\ z)\ y \\
\#\mathsf{newexp} & > & \lambda x, y.\ \mathsf{check}\ (y\ x)\ x \\
\#\mathsf{dyn}\{\delta\} & > & \lambda x.\ \mathsf{check}\ (\mathsf{equal}\ x\ x)\ \delta\mathsf{_shape} \\
\#\mathsf{var2exp} & > & \lambda x.x \\
\#\mathsf{shape} & > & \lambda x.x \\
\#\mathsf{succdim} & > & \lambda x.\ \mathsf{check}\ (x\tilde{}\geq\tilde{}0)\ \mathsf{succdim}\ x \\
\#c & > & c \text{ otherwise}
\end{array}$$

Figure 7: **FISh** reductions: Beta and where

$$\begin{array}{rcl}
(\lambda x.t)\ a & > & t[a/x] \\
t\ \mathsf{where}\ x = a & > & t[a/x]
\end{array}$$

Figure 8: **FISh** reductions: shape expressions

$$\begin{array}{rcl}
\mathsf{dyn}\{\delta\}\ \tilde{}d\{\delta\} & > & d\{\delta\} \\
\tilde{}d\{\delta_0, \ldots, \delta_k\}\ \tilde{}n_0\ \ldots\ \tilde{}n_{k-1} & > & \tilde{}p\{\delta_k\} \text{ where} \\
 & & \ p = d\ n_0..n_{k-1} \\
\mathsf{undim}\ (\mathsf{zerodim}\ t) & > & t \\
\mathsf{undim}\ (\mathsf{succdim}\ s\ t) & > & \mathsf{error} \\
\mathsf{lendim}\ (\mathsf{zerodim}\ t) & > & \mathsf{error} \\
\mathsf{lendim}\ (\mathsf{succdim}\ s\ t) & > & s \\
\mathsf{preddim}\ (\mathsf{zerodim}\ t) & > & \mathsf{error} \\
\mathsf{preddim}\ (\mathsf{succdim}\ s\ t) & > & t \\
\mathsf{numdim}\ (\mathsf{zerodim}\ t) & > & \tilde{}0 \\
\mathsf{numdim}\ (\mathsf{succdim}\ s\ t) & > & \mathsf{numdim}\ t\ \tilde{}+\ \tilde{}1 \\
\mathsf{equal}\ \delta\mathsf{_shape}\ \delta\mathsf{_shape} & > & \tilde{}\mathsf{true} \\
\mathsf{equal}\ (\mathsf{zerodim}\ t_0)\ (\mathsf{zerodim}\ t_1) & > & \mathsf{equal}\ t_0\ t_1 \\
\mathsf{equal}\ (\mathsf{zerodim}\ t_0)\ (\mathsf{succdim}\ s_1\ t_1) & > & \tilde{}\mathsf{false} \\
\mathsf{equal}\ (\mathsf{succdim}\ s_0\ t_0)(\mathsf{zerodim}\ t_1) & > & \tilde{}\mathsf{false} \\
\mathsf{equal}\ (\mathsf{succdim}\ s_0\ t_0)\ (\mathsf{succdim}\ s_1\ t_1) & > & \mathsf{check}\ (s_0\tilde{}= s_1) \\
 & & \mathsf{equal}\ t_0\ t_1
\end{array}$$

Figure 9: **FISh** reductions: shape contexts

$$\begin{array}{rcl}
\mathsf{newvar}\ sh\ \lambda x.\mathsf{error} & > & \mathsf{error} \\
\mathsf{forall}\ t_2\ t_3\ \lambda x.\mathsf{error} & > & \mathsf{error} \\
\mathsf{fix}\ \lambda x.\mathsf{error} & > & \mathsf{error} \\
\mathsf{newexp}\ sh\ \lambda x.\mathsf{error} & > & \mathsf{error}
\end{array}$$

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

$$\mathsf{newvar}\ sh\ \lambda x.t_0 \quad >_\Gamma \quad \mathsf{newvar}\ sh\ \lambda x.t_1$$

When $(sh, \theta) = (\mathsf{int\_shape}, \mathsf{exp\ int})$

$$\mathsf{forall}\ t_2\ t_3\ \lambda x.t_0 \quad >_\Gamma \quad \mathsf{forall}\ t_2\ t_3\ \lambda x.t_1$$

When $(sh, \theta) = (\tilde{}\mathsf{true}, \mathsf{comm})$

$$\begin{array}{rcl}
\mathsf{fix}\ \lambda x.t_0 & >_\Gamma & \mathsf{fix}\ (\lambda x.t_1) \\
\mathsf{newexp}\ sh\ \lambda x.t_0 & >_\Gamma & \mathsf{newexp}\ sh\ \lambda x.t_1
\end{array}$$

Figure 10: **FISh** reductions: data reduction

$$
\begin{array}{rcl}
\mathsf{assign}\ t\ e & > & \mathsf{vtc}\ (\lambda x.\ \mathsf{assign}\ x\ e)\ t \\
 & & \quad \text{if } \mathsf{newexp} \text{ or } \mathsf{condexp} \text{ in } t \\
!t & > & \mathsf{vte}\ (\lambda x.!x)\ t \quad \text{if } \mathsf{newexp} \text{ or } \mathsf{condexp} \text{ in } t \\
\mathsf{vtc}\ f\ y & > & f\ y \quad \text{if } y \text{ is a term variable} \\
\mathsf{vtc}\ f\ (\mathsf{get}\ t) & > & \mathsf{vtc}\ (\lambda y.\ f\ (\mathsf{get}\ y))\ t \\
\mathsf{vtc}\ f\ (\mathsf{sub}\ t\ i) & > & \mathsf{vtc}\ (\lambda y.\ \mathsf{newvar}\ \mathsf{int\_shape}\ \lambda j. \\
 & & \qquad j := i; f\ (\mathsf{sub}\ y\ j))\ t \\
\mathsf{vte}\ f\ y & > & f\ y \quad \text{if } y \text{ is a term variable} \\
\mathsf{vte}\ f\ (\mathsf{get}\ t) & > & \mathsf{vte}\ (\lambda y.\ f\ (\mathsf{get}\ y))\ t \\
\mathsf{vte}\ f\ (\mathsf{sub}\ t\ i) & > & \mathsf{vte}\ (\lambda y.\ \mathsf{newexp}\ (\#f\ (\mathsf{preddim}\ \#t)) \\
 & & \qquad\qquad \lambda z.\ \mathsf{newvar}\ \mathsf{int\_shape}\ \lambda j. \\
 & & \qquad\qquad j := i; z := f\ (\mathsf{sub}\ y\ j))\ t \\
\mathsf{getexp}\ !t & > & !(\mathsf{get}\ t) \\
\mathsf{subexp}\ !t_1\ t_2 & > & !(\mathsf{sub}\ t_1\ t_2)
\end{array}
$$

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

$$
\begin{array}{rcl}
g\ (\mathsf{newexp}\ sh\ f)\ t_1\ \ldots\ t_n & > & \mathsf{newvar}\ sh\ \lambda x_0. \\
 & & f\ x_0; g\ !x_0\ t_1\ \ldots\ t_n \\
g\ (\mathsf{condexp}\ s_0\ s_1\ s_2)\ t_1\ \ldots\ t_n & > & \mathsf{cond}\ s_0\ (g\ s_1\ t_1\ \ldots t_n) \\
 & & (g\ s_2\ t_1\ \ldots t_n)
\end{array}
$$

Let h be a term and n be a natural number. If (h, n) is one of $(d\{\delta_0, \ldots, \delta_k\}\ s_0\ \ldots s_j, k-1-j)$, $(\mathsf{getexp}, 0)$, $(\mathsf{subexp}, 1)$ or $(\mathsf{subexp}\ s, 0)$ then

$$
\begin{array}{rcl}
h\ (\mathsf{newexp}\ sh\ f) & & \\
t_1\ \ldots\ t_n & > & \mathsf{newexp}\ (\#h\ sh\ \#t_1\ \ldots \#t_n) \\
 & & \lambda x.\ \mathsf{newvar}\ sh\ \lambda x_0. \\
 & & f\ x_0; \\
 & & x := h\ !x_0\ t_1\ \ldots\ t_n \\
h\ (\mathsf{condexp}\ s_0\ s_1\ s_2) & & \\
t_1\ \ldots\ t_n & > & \mathsf{condexp}\ s_0\ (h\ s_1\ t_1\ \ldots t_n) \\
 & & (h\ s_2\ t_1\ \ldots t_n)
\end{array}
$$

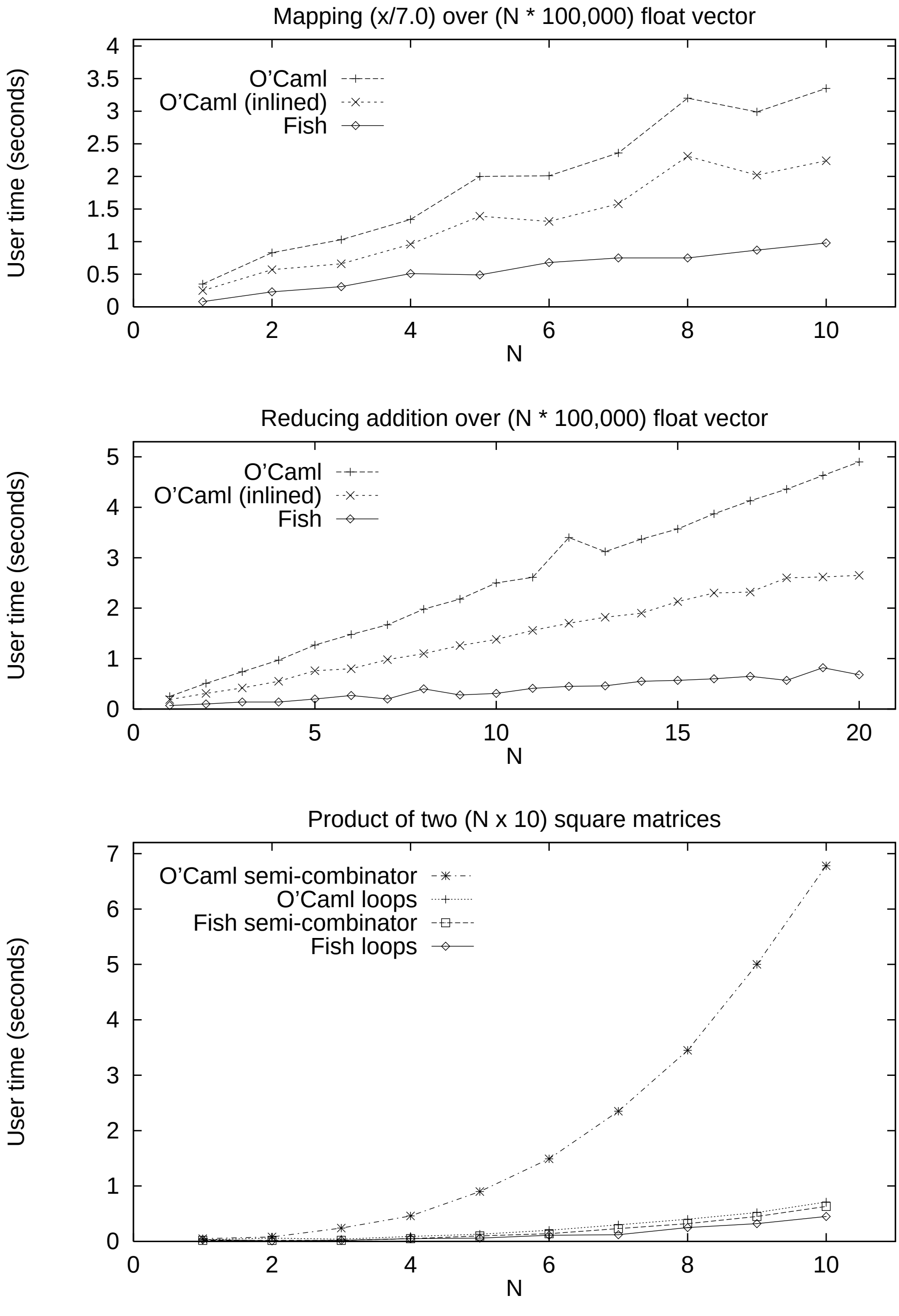

Figure 5: Benchmark results.