An Image Processing Language: External and Shallow/Deep Embeddings

Robert Stewart
Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, Scotland
R.Stewart@hw.ac.uk

ABSTRACT

Programming language users demand high performance, easy to understand syntax, and tooling such as profilers, debuggers and integrated development environments. Historically they were standalone, each with its own lexer and parser to implement a syntax, and an interpreter or compiler to implement a semantics. This approach incurs substantial engineering costs, both for the initial construction and also for ongoing maintenance as a language and its standard library grows. Modern language technology enables a more cost effective approach, namely to embed new languages inside existing languages, inheriting a host language's tooling, and its community as potential users. This paper uses a small image processing language to compare shallow and deep language embeddings with the external language approach. It focuses on optimisation opportunities, performance, ease of use and engineering costs.

CCS Concepts

•**Software and its engineering** → **Domain specific languages;** *Specialized application languages;* Data types and structures;

Keywords

Domain specific languages; image processing

1. INTRODUCTION

The motivation for *using* a DSL is productivity. They should be easy to use and read for a domain expert, so should lift a concise collection of domain concepts into meaningful notation that abstract common computational patterns. They should also hide details of the machine such as memory management, system calls, pointers and concrete data structure details, thus stripping away details outside the domain. The motivation for *building* a DSL is often performance or to attract domain experts as users. To satisfy power users, a DSL compiler should embody domain knowledge to eliminate unnecessary computation with algorithmic simplification rules, and a DSL runtime system may exploit common computational compositions for parallelism and memory efficiency.

Historically the approach was to build an external language with its own lexer and parser to parse its syntax, and an interpreter or compiler to implement its semantics. Or alternatively they were compiled to general purpose languages like C for execution, meaning that the notation in the language on top could bear resemblance to particular domains. The programmer was not hindered by the programming model of the target language, and the language engineer did not have to compile all the way to native code.

Advances in language engineering techniques offer new approaches for DSL implementations, namely by embedding them into existing languages. They can be embedded shallowly, where language primitives are implemented as computations in the host language, as values to which they evaluate. Or they can be embedded deeply, where each primitive returns a structure, not a value. They preserve their arguments into an AST, to be turned into a program of its own. The engineering cost of shallow embedding is cheap. The costs of deep embedding is less cheap because a compiler must be written to an external language, but is cheaper than building an external language from scratch. Deep embeddings geared for performance are very often compiled to a foreign language, bypassing the host's compiler, because the host language compiler 1) cannot compile to the target hardware, 2) has limited or no optimisations suitable for the domain, or 3) a foreign language compiler can generate better code for the domain.

This paper uses a small image processing language to compare two shallow embeddings, three deep embeddings and one external language. It presents a broad comparison of these distinct language approaches, rather than a deep comparison of complex optimisation opportunities in the image processing domain. The image processing language is therefore constricted to four image processing operators, and two for reading and writing image files. The language is presented in Section 2, six implementations of it are described in Section 3 and runtime performance for five benchmarks is reported in Section 4. The paper ends with a discussion in Section 5 on the trade-offs for optimisation, performance, ease of use and engineering costs.

RWDSL '16, March 12 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2889420.2892270

2. THE END LANGUAGE

2.1 Primitives

Expressive Notation for the image processing Domain (END) is a small language, with four image operations and two IO operations. The term *language* in this context is a collection of primitives, rather than a specific implementation. The language is in Figure 1. Image files are loaded with *imread* and written to with *imwrite*. The *brightenBy* and *darkenBy* primitives increase and reduce the pixel values at each position in the image by a user specific integer value. The *blurX* and *blurY* primitives blur the appearance of an image by applying a one dimensional weighted convolution with the kernel $\begin{bmatrix}1 & 2 & 1\end{bmatrix}$ at each position and its adjacent row or column wise neighbours. The nearest pixel is mirrored at the edges.

```
imread     : String  → Image
imwrite    : Image → String  → ()
blurX      : Image → Image
blurY      : Image → Image
brightenBy : Int  → Image → Image
darkenBy   : Int  → Image → Image
```

Figure 1: END Language Primitives

2.2 Benchmarks & Optimisations

The END implementations are compared using the five benchmarks in Figure 2. *Program 1* will assess raw image traversal performance. *Program 2* presents an opportunity for the language implementations to fuse two *blurX* operations at each pixel point with only one image traversal. *Program 3* could be optimised by eliminating the composition of *brightenBy* and *darkenBy*, as their composition brightens then darkens an image to the same degree. *Program 4* could be optimised in the same way, with an added complication that the brightening then darkening values are known only at runtime. *Program 5* represents an optimisation challenge when traversal is first row wise, then column wise, then point wise.

3. LANGUAGE IMPLEMENTATIONS

When a programmer wants to write a new software program, should they use a domain specific *embedded* language or a domain specific *external* language? This question is unpacked in the discussion Section 5. For image processing needs, the programmer can choose to use image processing specific notation, such as Halide [12] or the widely used OpenCV [2]. They may instead choose an array programming language and represent images as 2D integer arrays, using embeddings such as Repa [9] or Accelerate [3], or an external language like SaC [13]. Or they may just use plain sequential arrays or vectors in a general purpose language like Haskell. Table 1 summarises these different approaches. There is a distinction made between *source* and *target* languages. The source language is the one used by the programmer to express their image processing application, and the target language is the one compiled to native code. This section uses the image processing benchmarks from Section 2.2 to compare the image processing library route with Halide, array programming with Repa and Accelerate, two toy embeddings *ShallowEND* and *DeepEND*, and external array

Program *1*
```
img1 ← imread("in.png")
img2 ← brightenBy( 20 , img1 )
imwrite(img2 , "out.png")
```
Program *2*
```
img1 ← imread("in.png")
img2 ← blurX( img1 )
img3 ← blurX( img2 )
imwrite(img3 , "out.png")
```
Program *3*
```
img1 ← imread("in.png")
m ← getLine
img2 ← brightenBy( m , img1 )
img3 ← darkenBy( m , img2 )
imwrite(img3 , "out.png")
```
Program *4*
```
img1 ← imread("in.png")
m ← getLine
n ← getLine
img2 ← brightenBy( m , img1 )
img3 ← darkenBy( n , img2 )
imwrite(img3 , "out.png")
```
Program *5*
```
img1 ← imread("in.png")
img2 ← blurX( img1 )
img3 ← blurY( img2 )
img4 ← brightenBy( 30 , img3 )
imwrite(img4 , "out.png")
```

Figure 2: Benchmark Programs

programming with SaC. All implementations are available online [14].

3.1 Shallow Embeddings

3.1.1 ShallowEND

The simplest implementation of the END language is *ShallowEND*. The concrete implementation of an image is an algebraic data type with three values: a vector of unboxed integer values representing pixels, and the width and height of the image:

```
data Image = Image { pixels :: Vector Int
                   , width  :: Int
                   , height :: Int }
```

The entire *ShallowEND* implementation is based on immutable images of type *Image*, and function calls to the *vectors* [1] library to create new images from existing images. Haskell has a number of features that make it a good language for implementing DSLs, including type classes, infix syntax, quasi quotation and operator overloading. In the spirit of the domain, *brightenBy* and *darkenBy* are exposed with overloadings of + and -. *Program 3* brightens an input image twice, using a user specified runtime value. The *ShallowEND* program for *Program 3* is:

[1] http://hackage.haskell.org/package/vector

	Domain			Optimisations		Embedding		Language	
	Image processing	Array processing	General purpose	Compile time	Run time	Shallow	Deep	Source	Compiled
ShallowEND	✓			✓		✓		Haskell	Haskell
OpenCV	✓			✓		✓		C++	C++
DeepEND	✓				✓		✓	Haskell	Haskell
Halide	✓				✓		✓	C++	LLVM
Repa		✓		✓		✓		Haskell	Haskell
Accelerate		✓			✓		✓	Haskell	LLVM
SaC		✓		✓		*external*		SaC	C
Haskell			✓	✓		*external*		Haskell	Haskell

Table 1: Summary of Image Processing Approaches

```
main = do
    img1 ← imread "in.png"
    m ←read <$> getLine
    let img2 = ((−) m ∘ (+) m) img1
    imwrite "out.png" img2
```

As *ShallowEND* is a shallow embedding, there are no *runtime* optimisations of this program. All domain specific optimisations must happen at compile time. Fortunately, the GHC compiler includes a rewrite rule system [8] that lets us define the identity function optimisation when *brightenBy* is composed with *darkenBy*:

```
{−# RULES "darkenBy.brightenBy" forall img n.
          darkenBy n (brightenBy n img) = img #−}
```

In order for this rule to fire, the *NOINLINE* pragma is attached to these functions, to ensure they are not inlined before the GHC compiler starts applying user defined rewrite rules. Strictly speaking, the composition of *brightenBy* with *darkenBy* is not the identity function, because *e.g.* brightening a pixel with value 200 by 100 would saturate it to the white value 255. Brightening pixels modifies their values to at most 255, darkening to at least 0 for black. This can therefore be considered a domain specific optimisation that trades accuracy for efficiency, and one that no compiler should infer, at least not without the user's permission.

A similar optimisation is one that calculates the difference between *brightenBy* and *darkenBy* values, and replaces the two calls with a single *brigtenBy* call that accepts negative values. This *does* preserve accuracy, and is an optimisation that image processing language and array language compilers are able to apply by using loop fusion.

As the *vector* library is neither designed for array or image processing, it does not include built in support for handling boundary conditions. The mirroring of image pixels over its edges in order to compute *blurX* and *blurY* is therefore the responsibility of the programmer, shown in Figure 3. This concern is abstracted away with Accelerate and Halide, both languages provide the functionality to impose boundary conditions when applying overlapping stencil computations.

3.1.2 Repa

Repa [9] is a shallowly embedded parallel array DSL. Unlike *ShallowEND*, the Repa library has been carefully crafted for parallel performance. It takes ideas from the deep embedding approach, whereby computations are built up rather than being directly executed, as a way of performing fusion on array computation pipelines. Unlike those deep em-

```
blurX :: Image → Image
blurX (Image pixels w h) = Image (imap blurPixel pixels) w h
 where
  normalise x = round (fromIntegral x / 4.0)
  blurPixel i p
    −− right end of a row
    | (i+1) `mod` w == 0 =
      normalise ((pixels ! (i−1)) + p∗2 + p)

    −− left start to a row
    | i `mod` w == 0 =
      normalise (p + p∗2 + (pixels ! (i+1)))

    −− somewhere in between
    | otherwise =
      normalise (pixels ! (i−1) + p∗2 + (pixels ! (i+1)))
```

Figure 3: Hand Written Vector Image Boundary Condition

bedding approaches shown later, Repa function calls do not build up AST structures, they return Haskell array values. Those values, however, still represent fusible array operations because the returned arrays have *delayed* representations. Delayed arrays are represented by functions from indices to array elements at each position, rather than there being an unboxed value at each position. Array fusion is achieved by building up these plain Haskell functions as compositions. Manifest arrays are represented as unboxed values, *i.e.* real data.

Repa exposes the choice of when to use delayed and manifest array representations using Haskell's data families. To maximise the optimisation opportunities of END programs, images in the Repa implementation are all delayed, indicated by the type index *D*:

```
type Image = Array D DIM2 Int
```

Each call to an END primitive adds to the $\lambda(x, y) \rightarrow pixel$ composition for each position in the delayed array. Repa's important function is *computeP*, which separates the delayed array into chunks for multi-threaded parallelism, and the $\lambda(x, y) \rightarrow pixel$ at each position is then evaluated to an unboxed value, with an array type index *U*. The call to *computeP* is hidden inside the END primitive *imwrite*, when we know the user demands that image.

For efficient code generation, Repa uses the LLVM backend of the GHC Haskell compiler. The Repa implementation uses GHC primitive calls to transform aliasing information in the LLVM optimiser to encourage the LLVM compiler to

generate native SIMD vector opcodes. Because Repa's host is Haskell, it can also make use of the GHC rewrite rule that *ShallowEND* benefits from, to eliminate all image processing computation for *Program 3*.

3.2 Deep Embeddings

Calling primitives in deeply embedded languages return structures, not values. Compilers for deep embeddings are often optimised for particular computational models, *e.g.* deep embeddings in Haskell to compile structures to the high performance Intel Array Building Blocks C++ library [15], to CUDA [3] and to LLVM [10]. To illustrate this point, shallow and deep executions of *Program 5* are shown in Figure 4. In the shallow embedding (Section 3.1), calls to the END primitives result in direct execution of Haskell code to return image values. Any optimisations are applied by the GHC compiler, *e.g.* compile time fusion of vector operations. Deep embeddings of END primitives instead return structures, not values. Optimisations can be applied on those structures, before evaluating the program structure to a value using the host language at runtime (Section 3.2.1). Alternatively, those structures can be compiled to another language before being compiled to native code — bypassing the host language's compiler, possibly targeting a processor architecture the host language's compiler cannot support (Section 3.2.2). This section compares deeps embeddings of a sequential vector based END implementation, and LLVM backends of deep C++ and Haskell embeddings.

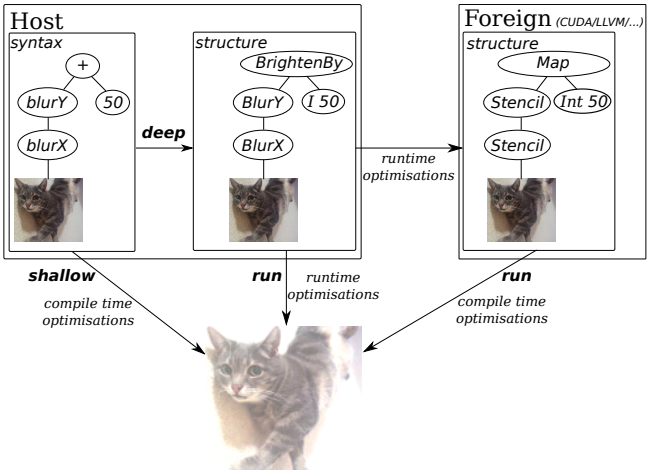

Figure 4: Shallow and Deep Embedding of Program 5

3.2.1 DeepEND

```
data DeepEND a where
  Img         :: Image → Exp Image
  I           :: Int  → Exp Int
  BlurX       :: Exp Image →Exp Image
  BlurY       :: Exp Image →Exp Image
  BrightenBy  :: Exp Int  → Exp Image →Exp Image
  DarkenBy    :: Exp Int  → Exp Image →Exp Image

brigtenBy = BrightenBy
...
```

Figure 5: *DeepEND* GADT

The *DeepEND* implementation uses generalised algebraic datatypes (GADTs), a type indexing feature in its host language Haskell. This deep embedding is shown in Figure 5. It has constructors *I* and *Img* for lifting integers and images into structures. The four *DeepEND* operations are exposed with data constructors, each with a smart constructors for convenience and do no computation, *e.g. brigtenBy*. *ShallowEND* and *DeepEND* programs look similar in Haskell, as illustrated in their respective expressions of *Program*5 in Figure 6. Smart constructors are used to lift the integer *30* and image *img1* into structures, and *run* interprets the deeply constructed AST.

```
-- | using ShallowEND
img1 ← imRead "in.png"
let  img2 = (brightenBy 30 ∘ blurY ∘ blurX) img1

-- | using DeepEND
img1 ← imRead "in.png"
let  img2 = run $ (brightenBy ( integer  30) ∘ blurY ∘ blurX)  (image img1)
```

Figure 6: DSL Notation for *Program 5*

The implementation of the *run* interpreter of *DeepEND* programs is shown in Figure 7. It uses the same image processing computations as the shallowly embedded *ShallowEND*. The difference is that the de-coupling between program construction and program evaluation offers the deep embedding the opportunity to optimise DSL programs before evaluation with *eval*. The *optimiseAST* function inspects ASTs at runtime to identify optimisations.

```
run  ::  Exp a → a
run ast  = eval (optimiseAST ast)

eval  ::  Exp a → a
eval  (I  i)       = i
eval  (Img img) = img
eval  (BrightenBy i  exp) = Shallow.brightenBy (eval  i)  (eval  exp)
eval  (DarkenBy  i  exp)  = Shallow.darkenBy   (eval  i)  (eval  exp)
eval  (BlurX exp)         = Shallow.blurX (eval  exp)
eval  (BlurY exp)         = Shallow.blurY (eval  exp)

optimiseAST :: Exp a → Exp a
optimiseAST (BrightenBy (DarkenBy subExp (I j)) (I  i))
  -- fire optimisation: (brightenBy m ∘ darkenBy n) = id
  | i == j    = optimiseAST subExp
  | otherwise =
      BrightenBy (DarkenBy (optimiseAST subExp) (I j)) (I  i)
optimiseAST (DarkenBy (BrightenBy subExp (I j)) (I  i))
  -- fire optimisation: (darkenBy m ∘ brightenBy n) = id
  | i == j    = optimiseAST subExp
  | otherwise =
      DarkenBy (BrightenBy (optimiseAST subExp) (I j)) (I  i)
optimiseAST (BlurX exp) = BlurX (optimiseAST exp)
optimiseAST exp@Img{} = exp
...
```

Figure 7: Optimising then Evaluating *DeepEND* ASTs

The *(darkenBy n . brightenBy n) = id* optimisation is implemented by pattern matching in the optimiser. One benefit to runtime, as opposed to compile time, optimisation is that values known only at runtime can be used in the deep embedding optimiser. In *Program 4*, two integers are acquired with *getLine* from the user and these values are used to brighten and then darken the image. If these runtime values are the same, the *optimiseAST* can eliminate the two

computations. This is not possible with *ShallowEND*, which relies on rewrite rules on values known at compile time.

3.2.2 Accelerate

Accelerate is a deeply embedded Haskell DSL with strict evaluation semantics for parallel array processing on multicore CPUs and GPUs. The library is a collection of array operations parameterised by types and scalar expressions. The programming model is quite similar to that of Repa, except that choices about when to delay and when to manifest array values is done automatically by the Accelerate compiler. So the *Array* type in Accelerate is not decorated with a D or U type index. The backend implementation of each array operation is tuned for each compute resource, *e.g.* fast on-chip shared memory for intra-block communication on GPUs in the CUDA backend [3].

As with the *DeepEND* implementation, the Accelerate functions construct an AST representation of programs. The functions in the Accelerate library are used to implement END primitives, *e.g. brightenBy* and its corresponding AST is shown in Figure 8.

```
-- | DSL primitive implemented with Accelerate
brightenBy :: Int → Acc Image →Acc Image
brightenBy i img = map (+(lift i)) img

-- | Constructed Accelerate AST for 'brightenBy'
Map add img
  where
    add = \x y → PrimAdd (<elided types>)
                 'PrimApp'
                 Tuple (NilTup 'SnocTup' x
                               'SnocTup' y)
```

Figure 8: From END to Accelerate ASTs

Each Accelerate backend is exposed with a *run* function, the user selects the desired backend by importing the correct module into their program. All array programming is done in the DSL, and each backend does the heavy lifting. There are numerous online code generators for compiling Accelerate ASTs to code at (Haskell) runtime, including CUDA [3] and LLVM [10] backends. The GPU backend uses Haskell's foreign function interface bound to a CUDA C API to transfer arrays to the GPU, generate GPU kernels for each collective array operation in the AST, and to transfer the resulting array back from the GPU to the host.

3.2.3 Halide

Halide is an image processing language that separates the expression of an algorithm from the parallel schedule to execute that algorithm. It is deeply embedded in C++, and takes a similar approach to Accelerate, by having online CUDA and LLVM code generators for parallel execution on multicore CPUs and GPUs. One difference is that the Halide compiler generates SIMD opcodes directly for each architecture (on ARM using NEON, on x86 using SSE and AVX), whereas Accelerate's LLVM backend relies on the LLVM compiler to vectorise the generated LLVM IR code.

Halide can be regarded as a small embedded functional programming DSL inside C++. Images that may otherwise be implemented as mutable 2D arrays in C++ are instead side-effect free functions from image indices to array elements. Expressions inside these pure image functions can be arithmetic operations, if-then-else expressions, references to function arguments or *let* expressions and calls to other functions.

```
Func imread(string filename);
Func imwrite(string fname, Func imgFun);
Func brightenBy(int i, Func imgFun);
Func darkenBy  (int i, Func imgFun);
Func blurY(Func imgFun);

/* Halide implementation of blurX */
Func blurX(Func imgFun)
{ Var x("x"), y("y"), c("c");
  Func input_16("input_16");
  input_16(x,y,c) = cast<uint16_t>(imgFun(x,y,c));
  Func blur_x("blur_x");
  blur_x(x, y, c) = (input_16(x-1, y, c) +
                     2 * input_16(x, y, c) +
                     input_16(x+1, y, c)) / 4;
  blur_x.vectorize(x,8).parallel(y);
  blur_x.compute_root();
  Func output("outputBlurX");
  output(x,y,c) = cast<uint8_t>(blur_x(x,y,c));
  return output; }
```

Figure 9: Halide Implementation of END

The type signatures for the Halide functions that implement END are shown in Figure 9, along with the body of the *blurX* function to show the Halide programming model. *Var* objects are names to use as variables in the language embedding. The user defined Halide function *blur_x* scope variables x and y on the left side, and on the right is an expression that uses those variables. In this case, x and y combined represent a pixel coordinate, indexing into the image column and row respectively. The third argument to *blur_x*, the variable c, represents the colour channel dimension. Operations on c are applied to every channel. In this case the image is a grayscale image so $+$ is applied to a single colour channel, though *blurX* can be used on images with multiple colour channels such as RGB images.

The Halide approach is slightly different to the languages described so far. The programming model separates algorithms from schedules, enabling the programmer to hand craft the parallelism for each stage of the pipeline. The traversal of the image in Figure 9 is done with an outer loop traversing columnwise, and an inner loop traversing row wise. When applying the 1D convolution kernel $\begin{bmatrix}1 & 2 & 1\end{bmatrix}$, the inner loop over rows is vectorised with *vector(x,8)*.

This schedule can be good for locality performance if the $\lambda(x, y) \rightarrow pixel$ function overlaps source coordinates to create target pixels. This is the case for the row wise 1D blur kernel, where the inner loop is likely to fit neatly into registers for SIMD SSE vector opcode generation. The *parallel(y)* call splits the source image function into chunks of rows for thread level parallelism with a parallel outer for loop.

The implementation of END *Program 5* is in Figure 10. The input image file is loaded and converted to a greyscale image with *imread*. The pipeline corresponding to the END *Program 5* is constructed by passing $\lambda(x, y) \rightarrow pixel$ functions around. The implementation of END's *imwrite* calls Halide's *save_image()* to save the image to a file.

Programs are constructed as feed forward pipelines of functions. A function builds on top of an existing image function, to define what value each pixel should have in an output image. Building an in-memory Halide program structure is done by passing around higher order functions. Pro-

```
Func img1Fun(x,y,c) = imread("in.png");
Func img2Fun = blurX(img1Fun);
Func img3Fun = blurY(img2Fun);
Func img4Fun = brightenBy(30, img3Fun);
imwrite("outpng", width, height, img4Fun);
```

Figure 10: END Program 5 with Halide

grams are evaluated by calling the Halide function *realize()* on a user defined pipeline of functions. It JIT compiles the pipeline, then runs it and returns the resulting buffer as an image.

3.3 External

3.3.1 SaC

Single Aassignment C (SaC) is an external functional array programming language, with syntax familiar to C programmers with notable exclusions of pointers and global variables. The language treats N-dimensional arrays as first class data structures. Like the Repa and Accelerate languages, SaC language primitives operate on arrays, *e.g.* to inspect the dimensionality, access array elements, and to generate new arrays from existing arrays. The compiler generates C code and relies on existing C compilers to optimise this further and to generate native code.

```
typedef int[.,.] image;
image imread(string fname)
void  imwrite(string fname, image img)
image blurX(image img)
image blurY(image img)
image darkenBy(int i, image img)

/* implementing brightenBy with SaC */
image brightenBy(int i, image img)
{ return min (img + i , 255); }
```

Figure 11: END Implementation with SaC

The type signatures for the SaC implementation of the END language is in Figure 11, along with the implementation of *brightenBy*. The *image* type is a typedef for *int[.,.]*, representing a 2D array of integers whose exact image shape is not known until runtime. SaC doesn't have a built in primitive for clamping arrays at their edges, so the implementation of *blurX* and *blurY* is more involved than Accelerate or Halide, as it involves shifting the image left and right to handle the image boundary conditions. All array arithemtic operations in the SaC standard library are implemented with SaC's *with* loop construct. The END implementation uses those array operations, so the optimisations to END programs are achieved by fusing the underlying *with* loops beneath the four END image operations. In contrast to Halide, all scheduling choices are automatically decided by the compiler.

4. BENCHMARK EVALUATION

All language implementations of the END image processing programs from Section 2.2 have been benchmarked on a four core Intel i5 3.20GHz CPU with 16GB of main memory running the 4.2.8 Linux kernel. GHC 7.10.2 was used to compile and run the embedded Haskell implementations, a git snapshot of the SaC compiler was used to compile SaC to C, and GCC 5.3.1 to compile the Halide C++ and SaC-generated C. A git snapshot of the *accelerate-llvm* Haskell was used in the Accelerate implementation, whilst Repa and vector library versions 3.4.0.2 and 0.11.0.0 respectively were used.

	Type	END Program 1	2	3	4	5
ShallowEND	*shallow*	80	2089	0	154	2132
Repa	*shallow*	886	7863	542	1412	8926
DeepEND	*deep*	78	2953	0	0	2988
Accelerlate	*deep*	77	623	81	88	629
Halide	*deep*	184	343	203	202	742
SaC	*external*	85	447	64	126	723

Table 2: Image processing Runtimes in Milliseconds

The benchmark results, reported in milliseconds, are in Table 2. They are the mean of five runs. To assess the image processing performance and to observe the effects of optimisations, the reported runtimes are for the image processing *computation*, the time taken to read and write image files is ignored. The same 7050×4525 8bit color RGB image is the input to every program. The value obtained with *getLine* for *Program 3* was 30, and the two *getLine* calls for *Program 4* both returned 30 at runtime.

As discussed in Section 3.1.1, the optimisation that eliminates image brightening used with image darkening, and vice versa, represents a domain specific trade-off between accuracy and efficiency. The GHC rewrite rule in Section 3.1.1 successfully substitutes the composition of *brightenBy n* and *darkenBy n* with the identity function in *Program 3* at compile time, resulting in a $0ms$ runtime. The rule was also used in the Repa implementation, so the $542ms$ is likely the time to setup the Repa scheduler with the *computeP* call. The runtime AST optimiser in *DeepEND* eliminated all computation in *Program 3* and *Program 4*, resulting in a $0ms$ runtime for both.

Of the parallel implementations, SaC and Accelerate achieve very similar fast runtimes, whilst Halide is marginally slower perhaps due to the runtime latency of JIT compiling and running the image processing pipeline. Repa was significantly slower for *Program 2* and *Program 5*. These results are measurements of very simple image processing, and should not be taken too seriously. For a more rigorous performance comparison of these parallel libraries and language, a much wider variety of real world and higher order image processing algorithms should be used as the benchmark.

5. DISCUSSION

5.1 Domain Specificity

What constitutes a DSL? Is it a language that constrains the programmer so that efficient native code can be generated, or is it a language that empowers the programmer to express algorithms more easily? Can a DSL be a control flow oriented language? It was argued [7] at the Nvidia technol-

ogy conference in 2009 that OpenGL is a DSL, one which captures code structure bracketed by *glBegin()* and *glEnd()* with sequences of instructions in between. Programmers indent their code as if these primitives are control structures of the host language.

In terms of image processing with END, the *ShallowEND* implementation is the embedding that is most directly connected to its host. Images pixels are stored as vectors in the host language, and the image operations are implemented using calls to functions in the *vectors* library. There are many Haskell libraries that also use that library, so it is trivial to use this END implementation in conjunction with other domains. The *Repa* implementation of END represents three layers of specialisation. The host language is general purpose, Repa at the next level is for array programming, and END image processing sits at the top. SaC, the only external language in this comparison, is more specialised than any of the embedded implementations because Haskell is general purpose and SaC is restricted to array programming.

Halide is the interesting case when testing the notion of domain specificity for image processing. To question the notion of domain specificity, the author has used Halide to implement an audio processing program, which is available online [14]. It uses a MIDI audio file parser library [2] for Halide's host language C++, and wraps the MIDI track data into buffers and then into Halide image data structures. A MIDI file contains tracks, and a track contains events. Event data takes the following form, one line per MIDI event:

```
1200  0xff  51  3  4  86  9c
1200  0x90  43  30
1440  0x90  48  31
```

The first column is the tick, indicating the duration of the note. The second column is the command byte, The *0x90* value represents a note to be on, whilst *0xff* indicates a meta message about a lyric, cue point, or set tempo. The third column is the note, and the fourth column is the note's volume. A segment of the program is shown in Figure 12. It extracts the note from each *0x90* MIDI event into an array, passes this into an Halide image as a buffer for processing, then reconstructs a new MIDI file using the modified notes.

The author has chosen to process Ludwig van Beethoven's Piano Sonata No. 8 in C minor, Op. 13. The program modifies $E\flat$ to E, $B\flat$ to B and $A\flat$ to A, such that the piece is transformed into the parallel key of C major, for a cheerier and more spirited sound. The processing of Piano Sonata No. 8 is mapped into a very wide image with one row, and that row is vectorised by a factor of 8 for faster processing. This leakage of another domain into Halide is primarily due to Halide's embedding in an otherwise general purpose language for which there are many libraries satisfying other domains.

Halide lacks library calls for reading non-image files, making it is less easy to process other file formats. Nevertheless, this MIDI experiment shows that domain specific does not always mean domain exclusive. A deeper investigation could assess, across different DSLs, how often DSL misuse is possible and whether misuse results in bad performance. For example, using the Halide image structure to host dense 2D arrays and measure the performance of applications from other domains, given that Halide's code generator has been tuned with image processing in mind.

It's never certain how languages end up being used. For example LISP, invented by John McCarthy in 1958, quickly became popular primarily for artificial intelligence research. Clojure however, a dialect of LISP, is today widely used in many other domains including web programming and distributed databases.

```
Func noFlats(Func music)
{ Var x, y, c;
  Expr value = music(x, y, c);
  // Eb to Es, Bb to B, Ab to A
  value = select(38==value, 39, value);
  value = select(50==value, 51, value);
  ...
  Func noFlats("noFlats");
  noFlats(x, y, c) = value;
  noFlats.vectorize(x, 8);
  noFlats.compute_root();
  return noFlats; }

int main()
{ // parse MIDI data
  MidiFile file;
  file.read("in.mid");
  int width = file.getEventCount(0);
  uint8_t pitches[width];
  for (int event=0; event<file[0].size(); event++)
    { uint8_t pitch = (uint8_t) file[0][event][1];
      pitches[event] = pitch; }

  // C minor to parallel C major with Halide
  Buffer in = Buffer(UInt(8),width,1,1,0,pitches);
  Image<uint8_t> image(in);
  songFun(x, y, c) = cast<uint8_t>(image(x, y, c));
  Func songWithNoFlats = noFlats(songFun);
  Image<uint8_t> newSong(width, 1, 1);
  songWithNoFlats.realize(newSong);

  // extract MIDI data from the image
  MidiFile outFile;
  outputfile.addTrack(1);
  buffer_t image_buffer =  *(newImg.raw_buffer());
  for (int event=0; event < events; event++)
    { MidiEvent event = midifile[0][event];
      event[1] = image_buffer.host[event];
      outFile.addEvent(1, event.tick, event); }
  outFile.write("out.mid");
}
```

Figure 12: Leaking Audio Processing into Halide

5.2 Ease of Use

Closely relating to language domain specificity is ease of use. How easy a programming language or library is to use is a notoriously difficult metric to measure. An embedded language like Repa or Accelerate is likely to be easy to learn if the host language is already known to the programmer, and potentially difficult to learn otherwise. For the image processing domain expert, an image processing specialised language like Halide or OpenCV are likely to be easier to read and write, because of their primitive types like *Image*, their built in support for common operations like stencil computations, support for handling boundary conditions and their standard libraries. Using Halide effectively is slightly more involved than using any of the other languages, because of its separation of algorithm to scheduling. An efficient Halide programmer needs to know the image processing domain and also have knowledge of parallel computing too. The

[2] https://github.com/craigsapp/midifile

scheduling choices include vectorisation, thread parallelism, unrolling, image tiling and nesting orders of loops. These choices are embedded into automatic compiler optimisations in other languages, including SaC.

There are several usability drawbacks when using embedded DSLs. DSL programs can be awkward to write if the host syntax is rigid. Moreover, domain specific error reporting of domain errors in an embedded setting can be difficult to implement in the language.

An important subtlety that can result in poor performance is the unclear boundary where the DSL ends and the host language begins. For example, an Halide programmer may start out by writing their functions as ones that take an image and return an image, just as you would when using SaC or a general purpose language. As described in Section 3.2.3, the "correct" programming model is instead to build up a pipeline of $\lambda(x, y) \rightarrow pixel$ functions bypassing higher order functions around. Not doing so eliminates all fusion optimisations in the Halide compiler. And yet a C++ compiler will compile $\lambda image \rightarrow image$ functions just as happily as higher order Halide functions, and this incorrect programming style was a mistake the author made when starting out with Halide.

A programmer's ability to use array based languages like Repa, Accelerate and SaC depends largely on their understanding of shapes, functional ranks and rank-N arrays – prevalent concepts in these languages. If the user is familiar with the programming model, these languages will likely yield high performance for their programs, because the compilers are rich in optimisable array metadata so can aggressively fuse and parallelise array computations. SaC has the ability to define a *typedef* for common structures in a domain, such a images, making programs easier for the domain expert to read. Nevertheless, an external language like SaC is yet another syntax and semantics the programmer must learn before they can become productive using it.

5.3 Software Scalability

Another consideration for language adoption is whether the language can be used *at scale*. Real world large scale software must satisfy the requirements of multiple domains. One framework may involve database querying, web services, and data processing.

It may be implemented as a collection of microservice written in different languages, glued together with language agnostic APIs. Alternatively if executable files, as opposed to language functions, each provide modularised functionality, then they can each be implemented in different languages then lifted into a scripting language and composed by pipeling *stdout* of one to *stdin* of another. A third approach is to use a hybrid language, which is the interspersion of multiple languages in one, *e.g.* JavaServer Pages programs consist of HTML with fragments of Java [5]. A fourth approach is to unify different problems into one language such as [4], which is a single web functional language for specifying what appears in the browser, what happens on the server and relational database queries.

The most common approach however, is to use a general purpose language and adopt domain specific libraries available for that language. In this case, Repa and Accelerate array programming can easily be integrated with other domains within Haskell, the same is true for Halide image processing amongst other domains with C++. External approaches are vulnerable to language cacophony, *i.e.* can it be used in conjunction with others? Despite being a specialised array programming language, SaC functions can be called from C programs thanks to SaC's foreign function interface, although this does require to the programmer to maintain code written in two different languages.

5.4 Engineering Costs

By far the cheapest engineering cost is the shallow embedding approach. The engineer implements each primitive using the host language, and relies entirely on the host language's compiler to generate native code. The expense here is the high probability of inferior performance when compared to deep embeddings and external languages. Programs in these latter approaches are able to be optimised holistically, and these optimisations are not constrained by the host's compiler's ability to generate good machine code. Deep embeddings overcome this problem, at the expense of needing to implement a compiler for the language, one for each targeted processor architecture. It comes then as little surprise that deep embedding efforts can bit rot, which is discussed in [10].

Functional programming languages are particularly well suited for hosting embedded DSLs, with features such as algebraic data types for hosting deep embeddings. Constructing and optimising ASTs in the absence of algebraic data types can become unworkable and tedious at scale [6]. An alternative to the GADT approach for separating an API from its implementation is type classes. A programmer defines a language API as a type class, and this allows multiple implementations (instances) of that class. Two syntactic features for DSL implementation is quasi quotation [11] and operator overloading, both can be used to depart from the host's syntax to present a more domain focused syntax.

External languages are the most expensive to develop and maintain. As the language evolves and as standard libraries grow, maintaining the compiler can become a tiresome and thankless task. Moreover, any tooling surround an external language, such as profilers, debuggers, and integrated development environments can easily bit rot if not kept in sync with the language as it changes. In contrast, embedded DSLs inherit all of the tooling from the host language for free.

5.5 Performance

The invention of *ShallowEND* and *DeepEND* were primarily motivated by the author's intention to demonstrate two forms of optimisation. The first, applied to *ShallowEND*, is to use domain specific rewrite rules to be fired at compile time (Section 3.1.1), which eliminates all computation for the END $Program3$. This approach means that there is no runtime overhead to judge whether optimisations can take place or not.

The author raised a question [1] with the Halide compiler developers on whether the Halide DSL enables the expression of image processing compile time rewrite rules. Their view is that such optimisations should happen in DSLs on top of Halide, in a very similar way to how the END embedding using Repa uses a rewrite rule to avoid generating Repa computations when possible.

The second optimisation approach, applied to *DeepEND*, uses AST optimisation at runtime. This AST transformation approach can optimise programs in ways that compile

time optimisations cannot. Because they fire at runtime, runtime values can be used to *e.g.* using two *30* values returned by *getLine* to eliminate the composition of *brightenBy* and *darkenBy* in *Program 4*. Another example might be a JIT compiled array programming language, by specialising functions for exact array shapes once these shapes are known, *e.g.* by inspecting at runtime the dimensions of an image returned with *imread*. However, the complexity of runtime AST transformation increases as the size of a deeply embedded language increases. As this complexity grows, so too does the time it takes to transform the AST, representing a runtime overhead. Balancing the runtime cost of optimisation, with the runtime saved by applying those optimisations, represents a delicate trade-off the language implementer needs to consider.

Of the six END implementations compared in this paper, four do image processing in parallel. One is shallowly embedded *(Repa)*, two are deeply embedded (*Accelerate* and *Halide*), and one is external (*SaC*). The three embedded implementations share one common programming trope: they all have the programmer build up compositions of image processing computations, and expose a primitive for optimising and evaluating that composition. In all three cases, that composition is pipelines of $\lambda(x, y) \rightarrow pixel$ functions. In Repa, the programmer builds builds up this function using Repa's delayed array representation, which is hidden inside the END abstraction on top of Repa. In Accelerate, the programmer uses the *Acc* embedded language to build up array computations. In Halide, the user passes around functions to build up these compositions directly. The three respective primitives to turn program structures into values is shown in Figure 13. These image processing compositions are fused at runtime when these primitives are called.

```
-- | Turns a delayed Repa array into a manifest array
-- holding unboxed values, (simplified types shown).
computeP :: Array D DIM2 Int →IO (Array U DIM2 Int)

-- | Compile and run Accelerate program on a CPU or GPU
run :: Arrays a ⇒ Acc a → a


/* Evaluate Halide function into existing buffer */
Func::realize (Image<T> dst)
```

Figure 13: Turning Program Runtime Structure into Values

The Halide approach of separating algorithms from schedules represents an interesting trade off between relying on an optimising compiler like the ones for SaC and Accelerate, versus letting a programmer making better optimisation choices or indeed bad choices. A more thorough benchmark is needed than the one in this paper, with a wider coverage of more complex image processing algorithms, to judge whether user guided scheduling yields better performance than optimising compilers for array languages.

6. ACKNOWLEDGMENTS

The author would like to thank Trevor McDonell for his insights into the performance of shallow and deep embeddings, Greg Michaelson for thought provoking discussions on the very notion of DSLs, and the developers of the SaC compiler for providing technical support. Thanks also to Patrick Maier and Blair Archibald for feedback on an earlier draft. This work was funded by the Engineering and Physical Science Research Council (EPSRC) under the *EP/K009931/1* Rathlin project grant.

7. REFERENCES

[1] A. Adams. "Rewrite rule system for user defined Halide pipeline transformations?". Halide-dev mailing list thread https://lists.csail.mit.edu/pipermail/halide-dev/2016-February/002209.html, February 2016.

[2] G. R. Bradski and A. Kaehler. *Learning OpenCV - Computer Vision with the OpenCV Library: Software That Sees*. O'Reilly, 2008.

[3] M. M. T. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming, DAMP 2011, Austin, TX, USA, January 23, 2011*, pages 3–14. ACM, 2011.

[4] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 266–296. Springer, 2007.

[5] M. Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011.

[6] J. Gibbons. Functional Programming for Domain-Specific Languages. In *Central European Functional Programming School - 5th Summer School, CEFP 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*, volume 8606 of *Lecture Notes in Computer Science*. Springer, 2015.

[7] P. Hanrahan. Domain-Specific Languages for Heterogeneous GPU Computing. Invited talk at the Nvidia Technology Conference, October 2009. http://www.graphics.stanford.edu/~hanrahan/talks/dsl/dsl1.pdf.

[8] S. P. Jones, A. Tolmach, and T. Hoare. Playing By The Rules: Rewriting as a Practical Optimisation Technique in GHC. In *Proceedings of the ACM SIGPLAN Haskell Workshop, Firenze, Italy, September 2, 2001*, pages 203–233. ACM, 2001.

[9] B. Lippmeier, M. M. T. Chakravarty, G. Keller, and S. P. Jones. Guiding Parallel Array Fusion with Indexed Types. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 25–36. ACM, 2012.

[10] T. L. McDonell, M. M. T. Chakravarty, V. Grover, and R. R. Newton. Type-Safe Runtime Code Generation: Accelerate to LLVM. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 201–212. ACM, 2015.

[11] S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything Old is New Again: Quoted Domain-Specific Languages. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016,*

St. Petersburg, FL, USA, January 20 - 22, 2016, pages 25–36. ACM, 2016.

[12] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 519–530. ACM, 2013.

[13] S. Scholz. Single Assignment C: Efficient Support for High-Level Array Operations in a Functional Setting. *J. Funct. Program.*, 13(6):1005–1059, 2003.

[14] R. Stewart. Sources for END DSL Implementations. GitHub repository https://github.com/robstewart57/END-dsl-implementations, February 2016.

[15] B. J. Svensson and M. Sheeran. Parallel Programming in Haskell Almost For Free: An Embedding of Intel's Array Building Blocks. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing, Copenhagen, Denmark. ICFP 2012, September 9-15, 2012*, pages 3–14. ACM, 2012.