Haskell EDSL Implementations

Scottish Programming Languages and Verification
Summer School 2019

Rob Stewart (R.Stewart@hw.ac.uk)
August 2019

Heriot-Watt University, Edinburgh
Haskell Take on DSLs
Hi all,

for people that have followed my posts on the DSL subject this question probably will seem strange, especially asking it now..

Because out there I see quite a lot of stuff that is labeled as DSL, I mean for example packages on hackage, quite useuful ones too, where I don't see the split of assembling an expression tree from evaluating it, to me that seems more like combinator libraries.

Thus:

What is a DSL?

Günther
<table>
<thead>
<tr>
<th>Date</th>
<th>Subject</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-10-28</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>S. Doaitse Swierstra</td>
</tr>
<tr>
<td>2009-10-28</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Nils Anders Daniel</td>
</tr>
<tr>
<td>2009-10-22</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Robert Atkey</td>
</tr>
<tr>
<td>2009-10-13</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Nils Anders Daniel</td>
</tr>
<tr>
<td>2009-10-12</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Robert Atkey</td>
</tr>
<tr>
<td>2009-10-12</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>S. Doaitse Swierstra</td>
</tr>
<tr>
<td>2009-10-12</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Sjoerd Visscher</td>
</tr>
<tr>
<td>2009-10-09</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Colin Paul Adams</td>
</tr>
<tr>
<td>2009-10-09</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Gregg Reynolds</td>
</tr>
<tr>
<td>2009-10-08</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Colin Paul Adams</td>
</tr>
<tr>
<td>2009-10-08</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>George Pollard</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>[Haskell-cafe] What <em>is</em> a DSL?</td>
<td>oleg</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Creighton Hogg</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Robert Atkey</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Minh Thu</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Gunther Schmidt</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Don Stewart</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Dan Piponi</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Robert Atkey</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Minh Thu</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Joe Fredette</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Gunther Schmidt</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Joe Fredette</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>Re: [Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Emil Axelsson</td>
</tr>
<tr>
<td>2009-10-07</td>
<td>[Haskell-cafe] What <em>is</em> a DSL?</td>
<td>Gunther Schmidt</td>
</tr>
</tbody>
</table>
A DSL is just a domain-specific language. It doesn’t imply any specific implementation technique. A shallow embedding of a DSL is when the ”evaluation” is done immediately by the functions and combinators of the DSL. I don’t think it’s possible to draw a line between a combinator library and a shallowly embedded DSL.

A deep embedding is when interpretation is done on an intermediate data structure.

– Emil Axelsson, Chalmers University.
I’ve argued that every monad gives a DSL. They all have the same syntax - do-notation, but each choice of monad gives quite different semantics for this notation.

– Dan Piponi
I’ve informally argued that a true DSL – separate from a good API – should have semantic characteristics of a language: binding forms, control structures, abstraction, composition. Some have type systems. Basic DSLs may only have a few characteristics of languages though – a (partial) grammar. That’s closer to a well-defined API in my books.

– Don Stewart
Parsec, like most other parser combinator libraries, is a shallowly embedded DSL... a Haskell function that does parsing, i.e. a function of type

```
String -> Maybe (String, a)
```

You can’t analyse it further— you can’t transform it into another grammar to optimise it or print it out— because the information about what things it accepts has been locked up into a non-analysable Haskell function. The only thing you can do with it is feed it input and see what happens.

– Bob Atkey
Embeddings in Haskell
• GHC gives us
  • frontend: syntax & type checking
  • interpreter: test components and small programs
• Haskell EDSL often rely on
  • higher order functions
  • type class overloading
  • monads
• Choices
  1. functions directly capture semantics of language (shallow)
  2. based on the abstract syntax of EDSL program (deep)
     • multiple interpretations e.g. acceleration, visualisation..
Shallow Embeddings

- DSL
  - libraries
  - language
  - assembly
  - GHC RTS
  - executable

 frontend
 backend
Compile Time Metaprogramming
Three case studies

1. Repa: array processing
2. Accelerate: array processing
   - strict evaluation semantics (host language is lazy)
3. Lava: circuit description
Array Processing: Repa
Haskell Embeddings
Parallel Shallow Embedding
Repanda Language

```
data family Array rep sh e
data instance Array D sh e = ADelayed sh (sh -> e)
data instance Array U sh e = AUnboxed sh (Vector e)

-- types for array representations
data D    -- Delayed
data U    -- Manifest, unboxed

computeP :: (Load rs sh e, Target rt e)
    => Array rs sh e
    -> Array rt sh e
```

type Image a = Array U DIM2 a

gradientX :: Image Float -> IO (Image Float)
gradientX img = computeP
  $ forStencil2 BoundClamp img
  [stencil2| -1  0  1
              -2  0  2
              -1  0  1 |]

gradientY :: Image Float -> IO (Image Float)
gradientY img = computeP
  $ forStencil2 BoundClamp img
  [stencil2|  1  2  1
            0  0  0
            -1 -2 -1 |]

gradMagnitude :: Float -> Image Float -> Image Float
    -> IO (Image (Float, Word8))
gradMagnitude threshLow dX dY = computeP $ R.zipWith mag dX dY
  where mag = ...
Repa Example

```haskell
readImage :: String -> IO Image
saveImage :: Image -> String -> IO ()

main = do
    image1 <- readImage "input.png"
    image2 <- gradientX image1
    image3 <- gradientY image1
    image4 <- gradMagnitude thresh image2 image3
    saveImage image4 "output.png"
```

- Each `computeP` call uses static scheduler
  - assumes well balanced regular parallelism
- Monadic interface sequences parallel "gang" schedulers
  - avoid: cache contention, overloading OS scheduler

Lippmeier et al., “Guiding parallel array fusion with indexed types”.
-- 'n' is number of threads to use
forkGang :: Int -> IO Gang
forkGang n =

... zipWithM_ forkOn [0..] -- create worker threads
$ zipWith3 gangWorker
[0 .. n-1] mvsRequest mvsDone

gangWorker :: Int -> MVar Req -> MVar () -> IO ()
gangWorker threadId varRequest varDone
= do -- Wait for a request
    req <- takeMVar varRequest
    case req of
        ReqDo action
        -> do -- Run the action we were given.
            action threadId
            ...


Array Processing: Accelerate
Haskell Embeddings
Deep Embeddings with Haskell

Material from

- Trevor McDonell’s PhD thesis
- Email exchanges with Trevor

User programs generate CUDA/LLVM programs at runtime

```haskell
dotp :: Num a -> Vector a -> Vector a -> Acc (Scalar a)
dotp xs ys =
  let
    xs' = use xs
    ys' = use ys
  in
    fold (+) 0 ( zipWith (*) xs' ys' )
```

Acc is an Accelerate program, will produce value of type a

run function generates code, compiles it, executes it

```haskell
run :: Arrays a -> Acc a -> a
```
McDonell, “Optimising Purely Functional GPU Programs”.
map :: (Shape sh, Elt a, Elt b)
     => (Exp a -> Exp b)
     -> Acc (Array sh a)
     -> Acc (Array sh b)

zipWith :: (Shape sh, Elt a, Elt b, Elt c)
     => (Exp a -> Exp b -> Exp c)
     -> Acc (Array sh a)
     -> Acc (Array sh b)
     -> Acc (Array sh c)

stencil :: (Stencil sh a stencil, Elt b)
      => (stencil -> Exp b)
      -> Boundary (Array sh a)
      -> Acc (Array sh a)
      -> Acc (Array sh b)

-- slice, fold, backpermute, ...
• Skeletons build trees to represent array computations
• GADTs preserve embedded program’s type info in term tree
• Smart constructors

\[
\text{Data.Array.Accelerate.Language}
\]

map = Acc $$\text{Map}$$
zipWith = Acc $$\text{ZipWith}$$
fold = Acc $$\text{Fold}$$
...

• Internal conversion from HOAS to de Bruijn representation enables program transformations and recovers sharing

-- convert array expression to de Bruijn form
-- incorporating sharing information
convertAcc :: Arrays arrs => Acc arrs -> AST.Acc arrs
The generated IR is optimised (e.g. fusion) then compiled to object code, which is linked at runtime and executed.
[cunit |
  $esc:("#include <accelerate_cuda.h>")
  $edecls:texIn
  extern "C" __global__ void map
  ( // types of the elements of the input/output arrays
    $params:argIn,
    $params:argOut
  ){
    const int shapeSize = size(shOut);
    const int gridSize = $exp:(gridSize dev);
    int ix;
    for ( ix = $exp:(threadIdx dev); ix < shapeSize; ix += gridSize )
    {
        // gets input array element from index
        $items:(dce x .=. get ix)

        // scalar operation per element
        $items:(setOut "ix" .=. f x)
    }
  }
]

Listing 4.1 from McDonell’s PhD thesis.
Skeleton Code Templates

- Accelerate now LLVM based (not CUDA)
- But same template skeleton idea
- Parallel code structure defined by skeleton templates
- Types & user defined functions added to template during code gen
- Doesn’t use TemplateHaskell’s quasiquotation
- Instead uses Haskell LLVM library API

Skeleton Code Templates: Map (LLVM)

```haskell
mkMap aenv apply = let
    (arrOut, paramOut) = mutableArray @sh "out"
    (arrIn, paramIn) = mutableArray @sh "in"
    paramEnv = envParam aenv

    in
    makeOpenAcc "map" (paramOut ++ paramIn ++ paramEnv) $ do
        start <- return (lift 0)
        end <- shapeSize (irArrayShape arrIn)
        imapFromTo start end $ \i -> do
            xs <- readArray arrIn i
            ys <- app1 apply xs
            writeArray arrOut i ys
        return_

    -- from 'accelerate-llvm' package
    imapFromTo :: IR Int -> IR Int
          -> (IR Int -> CodeGen Native ()) -> CodeGen Native ()
```
Comparing Accelerate and Repa
Comparing Accelerate and Repa

Same goals:

- Collective operations on regular multidimensional arrays
- Non-nested, flat data-parallelism
- Embed in Haskell

Achieve these goals in very different ways:

- Repa uses type indexed array representations to help GHC generate better code
- Accelerate avoids GHC’s code generation altogether
Performance

McDonell et al., “Type-safe runtime code generation: accelerate to LLVM”.
Things you can do many with an Accelerate program:

1. Pretty print it
2. Interpret it
3. Generate & execute CUDA for GPUs
4. Generate & execute LLVM for CPUs/GPUs
5. Visualise program graph with GraphViz
Accelerate Arrays and Functions

\[
\begin{align*}
\text{arr1} & :: \text{Acc (Array DIM2 Int)} \\
\text{arr1} & = \text{A.use } $ \text{A.fromList (Z :: 3 :: 3)} [1..9] \\
\text{arr2} & :: \text{Acc (Array DIM2 Int)} \\
\text{arr2} & = \text{A.use } $ \text{A.fromList (Z :: 3 :: 3)} [10..19] \\
\text{f} & :: \text{Acc (Array DIM2 Int) -> Acc (Array DIM2 Int)} \\
\text{f} & = \text{A.map (+2) . A.map (+1)} \\
\text{g} & :: \text{Acc (Array DIM2 Int) -> Acc (Array DIM2 Int)} \\
\text{g} & = \text{A.transpose}
\end{align*}
\]
let program = A.zip (f arr1) (g arr2)
print program -- show it

let a0 = use (Array (Z :: 3 :: 3) [1,2,3,4,5,6,7,8,9]) in
let a1 = use (Array (Z :: 3 :: 3) [10,11,12,13,14,15,16,17,18]) in generate
  (intersect
    (shape a0)
    (let x0 = shape a1
      in Z :: indexHead x0 :: indexHead (indexTail x0)))
  (\x0 -> (2 + (1 + (a0!x0))
    , a1!Z :: indexHead x0 :: indexHead (indexTail x0)))
let program = A.zip (f arr1) (g arr2)
print print (A.run program) -- run it

Matrix (Z :: 3 :: 3)

[ (4,10), (5,13), (6,16),
  (7,11), (8,14), (9,17),
  (10,12), (11,15), (12,18)]
Comparison of Profiling Tooling
• Repa uses GHC runtime system
• Threadscape for profiling GHC generated parallel code
• Hence: Repa can inherit Threadscope profiling tool
Accelerate Profiling

- Accelerate doesn’t generate parallel code via GHC
- Doesn’t have access to GHC tools e.g. Threadscape
- Use NVidia profiler GPU profiling tooling instead

Figure 4.2 from McDonell’s thesis.

McDonell, “Optimising Purely Functional GPU Programs”. 36
Implementation Considerations
Repa Implementation Considerations

- Good: GHC has good multicore/concurrency support
- Good: less engineering reuse GHC code generation
- Questionable: at mercy of GHC code generation

Question:

Can GHC Core be relied on for producing efficient high performance numerical code? E.g inlining and constant propagation for aggressive array fusion?

GHC Core is a SystemF language, not an array processing IR.
Accelerate Implementation Considerations

- Generate **simple LLVM IR** for the LLVM compiler
- Hope LLVM optimisations fire e.g. loop vectorisation
- LLVM/CUDA compilers assume human-written code
- Accelerate should mimic what a human would write
- Obscure LLVM code might rule out LLVM optimisations
- Don’t generate SIMD instructions
  - Rely on LLVM auto-vectorisation
  - Accelerate produces code it *knows* LLVM can vectorise well
  - Accelerate tells LLVM exactly which CPU is being used
  - Ask LLVM to vectorise for this CPU
Another Domain: Circuit Description
Deep EDSL: Lava

- Strongly typed EDSL for describing hardware circuits
- Deeply embedded
  - **test** circuit designs with GHCi (host language interpreter)
  - **generate** VHDL to synthesise circuits to hardware

Example from Andy Gill’s *ACM Communications* paper.

---

Gill, “Domain-specific languages and code synthesis using Haskell”.

39
counter
:: (Rep a, Num a) => Signal Bool -> Signal Bool -> Signal a

counter restart inc = loop

where
reg = register 0 loop
reg' = mux2 restart (0, reg)
loop = mux2 inc (reg' + 1, reg')
Counting Pulses

Simulate with GHCi:

GHCi> counter low (toSeq (cycle [True,False,False]))
1 : 1 : 1 : 2 : 2 : 2 : 3 : 3 : 3 : ...

Reify deep embedding:

GHCi> reify (counter (Var "restart") (Var "inc"))
[(0,MUX2 1 (2,3)),
 (1,VAR "inc"),
 (2,ADD 3 4),
 (3,MUX2 5 (6,7)),
 (4,LIT 1),
 (5,VAR "restart"),
 (6,LIT 0),
 (7,REGISTER 0 0)]
architecture str of counter is
    signal sig_2_o0 : std_logic_vector(3 downto 0);
    ...
begin
    sig_2_o0 <= sig_4_o0 when (inc = '1') else sig_6_o0;
sig_5_o0 <= std_logic_vector(...);
sig_6_o0 <= "0000" when (restart = '1') else sig_10_o0;
sig_10_o0_next <= sig_2_o0;
proc14 : process(rst,clk) is
begin
    if rst = '1' then
        sig_10_o0 <= "0000";
    elseif rising_edge(clk) then
        if (clk_en = '1') then
            sig_10_o0 <= sig_10_o0_next;
        ...
end process;
end architecture;
Summary
## Summary

*Approach domain specific opts host opts language examples*  

<table>
<thead>
<tr>
<th>Approach</th>
<th>domain specific opts</th>
<th>host opts</th>
<th>language</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>shallow</td>
<td>yes (rewrite rules)</td>
<td>yes</td>
<td>host</td>
<td>repa, HdpH-RS</td>
</tr>
<tr>
<td>deep</td>
<td>yes (runtime)</td>
<td>no</td>
<td>host</td>
<td>Accelerate, Lava</td>
</tr>
<tr>
<td>MP</td>
<td>yes (compile time)</td>
<td>yes</td>
<td>quasiquotes</td>
<td>PanTHeon</td>
</tr>
</tbody>
</table>

(MP = metaprogramming)