DSLs:
To Embed or not to Embed

Ryan Newton

5/2014

INDIANA UNIVERSITY

Musings and rants pertaining to adoption

- Not a research talk
- But first some context…

INDIANA UNIVERSITY

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu

INDIANA UNIVERSITY

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu

Messaging (MPI)

INDIANA UNIVERSITY

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu

Messaging (MPI)

Threading (TBB,OMP..)

INDIANA UNIVERSITY

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu

Messaging (MPI)

Threading (TBB,OMP..)

Vectorization (SIMD)

INDIANA UNIVERSITY

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu

Messaging (MPI)

Threading (TBB,OMP..)

Vectorization (SIMD)

Cloud Haskell, HdpH, meta-par

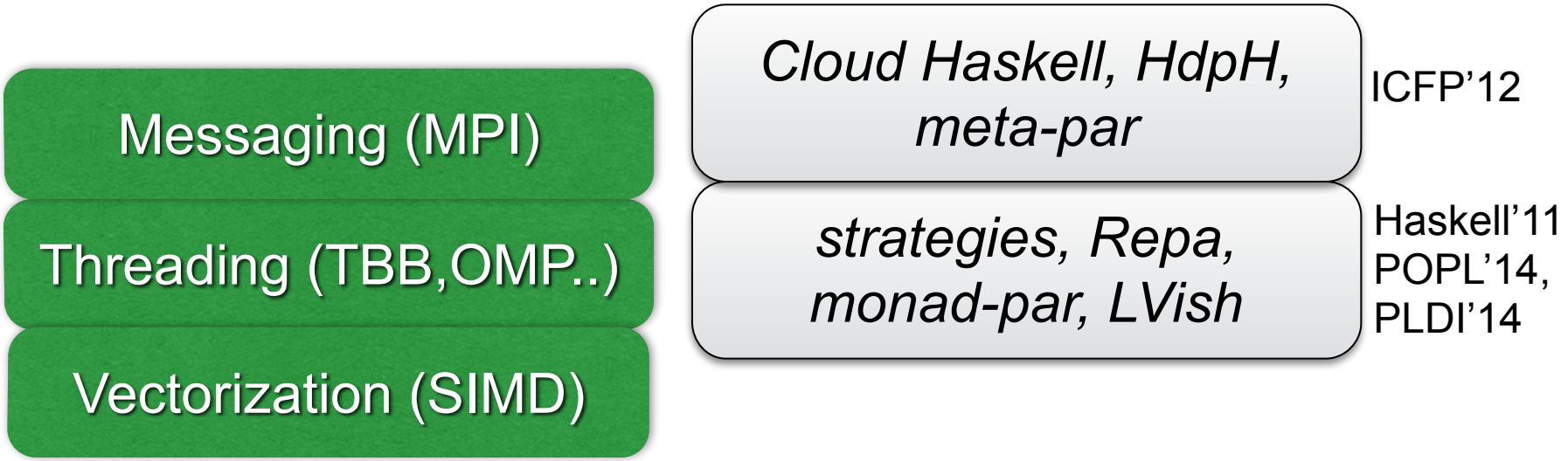
ICFP'12

A Haskell Cake

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu



A Haskell Cake

INDIANA UNIVERSITY

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu

Messaging (MPI)

Threading (TBB,OMP..)

Vectorization (SIMD)

Cloud Haskell, HdpH, meta-par — ICFP'12

strategies, Repa, monad-par, LVish — Haskell'11 POPL'14, PLDI'14

GHC SIMD primops, Intel HRC — Mainland, ICFP'13

A Haskell Cake

SafeHaskell Determinism

- Pure function
  - (Architecture,Prog) → Answer
- Still allows parallelism!
- Some relaxation helpful:
  - (Word size, OS name, GPU model, Prog) → Answer

HLL's (e.g. Haskell) for HPC?

- Not yet.
- Feature bag for workload:
  - {Flops, read/write array elem, funcall/jmp}
  - {alloc/gc, array copy, indirect jmp}
- Orthogonal: compiler quality
  - aggressive compiler techniques can perform alchemy & transform feature bag
  - Haskell can sometimes get down to blue only (how often? see Leaf Petersen et al. 2013)
  - What remains is bkend compiler quality

INDIANA UNIVERSITY

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu

Cloud Haskell, HdpH, meta-par

strategies, Repa, monad-par, LVish

GHC SIMD primops, Intel HRC

A Haskell Cake

INDIANA UNIVERSITY

Three Layer Cake for Shared-Memory Programming

Arch D. Robison
Intel Corporation
1906 Fox Drive
Champaign IL 61821
arch.robison@intel.com

Ralph E. Johnson
University of Illinois at Urbana-Champaign
Department of Computer Science
201 N. Goodwin Ave
Urbana IL 61801
rjohnson@illinois.edu

Cloud Haskell, HdpH, meta-par

Accelerate, StreamIt ….

strategies, Repa, monad-par, LVish

GHC SIMD primops, Intel HRC

A Haskell Cake

DSLs @ IU

- Standalone
  - Harlan - closures on the GPU + region based mem mgmt *(OOPSLA'14)*
- Haskell-embedded
  - Accelerate - cpu bkends + multi-device
  - StreamItHS (one-off experiment)
  - Obsidian - lower-lvl GPU kernel construction *(CACM 06/2014)*
  - *[new]* Probabilistic programming (Ken Shan)
  - *[new]* FLOOD (new streaming DSL) + Flange (net. monitoring and in-net comp.)

Accelerate

Accelerate

- Meta: Haskell
  - replace Prelude things: (==*), (>*), IsNum

Accelerate

```
dotp :: IsNum n => Acc (Vector n)
                -> Acc (Vector n)
                -> Acc (Scalar n)
dotp x y = fold (+) 0 (zipWith (*) x y)
```

- Meta: Haskell
  - replace Prelude things: (==*), (>*), IsNum

Accelerate

```
dotp :: IsNum n => Acc (Vector n)
                -> Acc (Vector n)
                -> Acc (Scalar n)
dotp x y = fold (+) 0 (zipWith (*) x y)
```

- Meta: Haskell
  - replace Prelude things: (==*), (>*), IsNum
  - (Z :. x :. y) shapes and shape types
    - rank polymorphism *(e.g. fold inner of N dims)*

Accelerate

```
dotp :: IsNum n => Acc (Vector n)
                -> Acc (Vector n)
                -> Acc (Scalar n)
dotp x y = fold (+) 0 (zipWith (*) x y)
```

- Meta: Haskell
  - replace Prelude things: (==*), (>*), IsNum
  - (Z :. x :. y) shapes and shape types
    - rank polymorphism *(e.g. fold inner of N dims)*
  - works w/ custom product types
    - lift/unlift for tuples *(no overloaded pat. match)*

Accelerate

```
dotp :: IsNum n => Acc (Vector n)
                -> Acc (Vector n)
                -> Acc (Scalar n)
dotp x y = fold (+) 0 (zipWith (*) x y)
```

- Meta: Haskell
  - replace Prelude things: (==*), (>*), IsNum
  - (Z :. x :. y) shapes and shape types
    - rank polymorphism *(e.g. fold inner of N dims)*
  - works w/ custom product types
    - lift/unlift for tuples *(no overloaded pat. match)*
  - Implementation: observable sharing + type-level de-bruijn indices

Accelerate

```
dotp :: IsNum n => Acc (Vector n)
                -> Acc (Vector n)
                -> Acc (Scalar n)
dotp x y = fold (+) 0 (zipWith (*) x y)
```

- Meta: Haskell
  - replace Prelude things: (==*), (>*), IsNum
  - (Z :. x :. y) shapes and shape types
    - rank polymorphism *(e.g. fold inner of N dims)*
  - works w/ custom product types
    - lift/unlift for tuples *(no overloaded pat. match)*
  - Implementation: observable sharing + type-level de-bruijn indices
- Target:
  - multi-dim arrays, tuples, scalars,

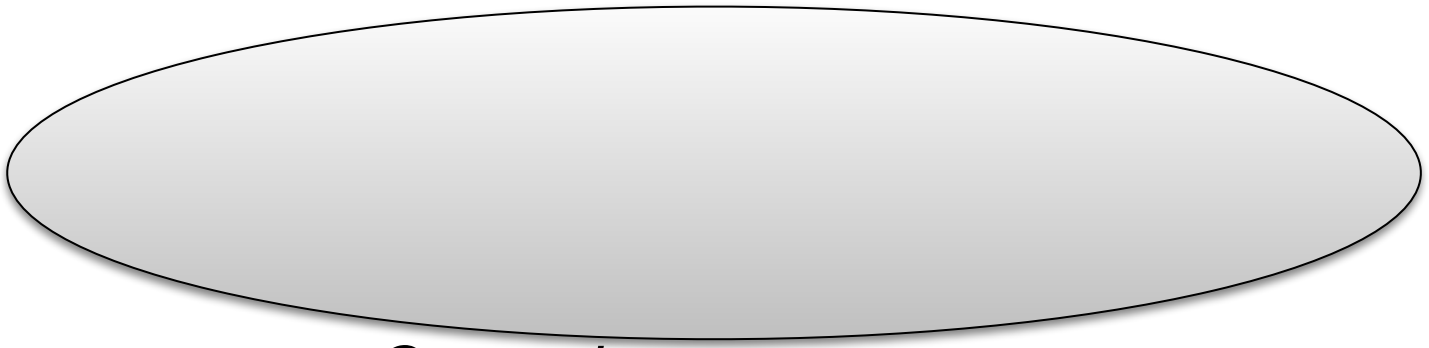
Accelerate

```
dotp :: IsNum n => Acc (Vector n)
                -> Acc (Vector n)
                -> Acc (Scalar n)
dotp x y = fold (+) 0 (zipWith (*) x y)
```

- Meta: Haskell
  - replace Prelude things: (==*), (>*), IsNum
  - (Z :. x :. y) shapes and shape types
    - rank polymorphism *(e.g. fold inner of N dims)*
  - works w/ custom product types
    - lift/unlift for tuples *(no overloaded pat. match)*
  - Implementation: observable sharing + type-level de-bruijn indices
- Target:
  - multi-dim arrays, tuples, scalars,
  - fixed communication combinators (fold, scan)

DSL compilers provide small targets

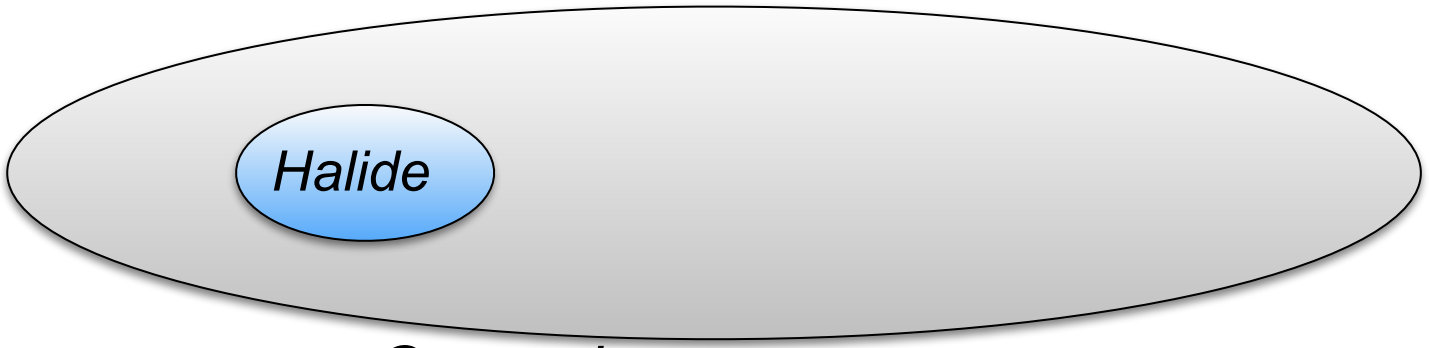
- Combination of
  - restricting lang features
  - impose structure (stream graph, stencil, etc)
- How best to hit them?



General purpose programs

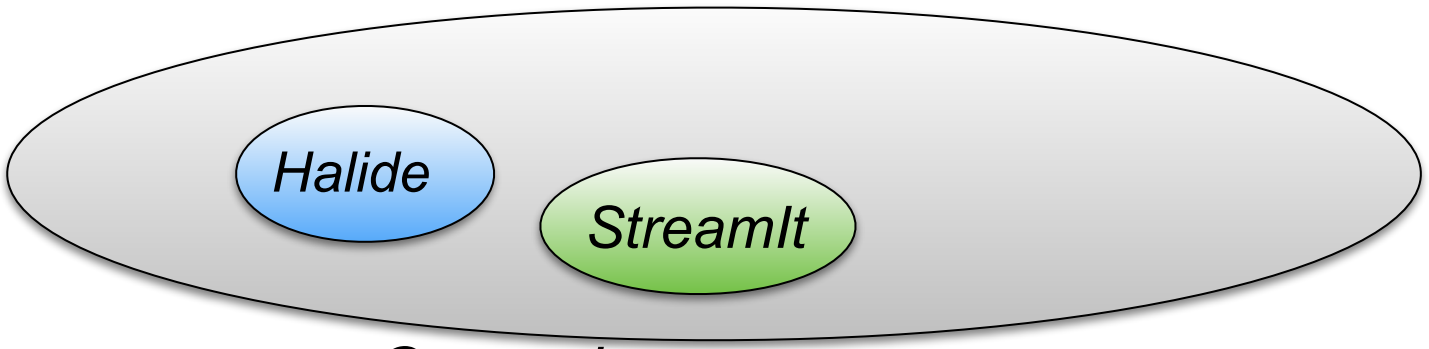
DSL compilers provide small targets

- Combination of
  - restricting lang features
  - impose structure (stream graph, stencil, etc)
- How best to hit them?



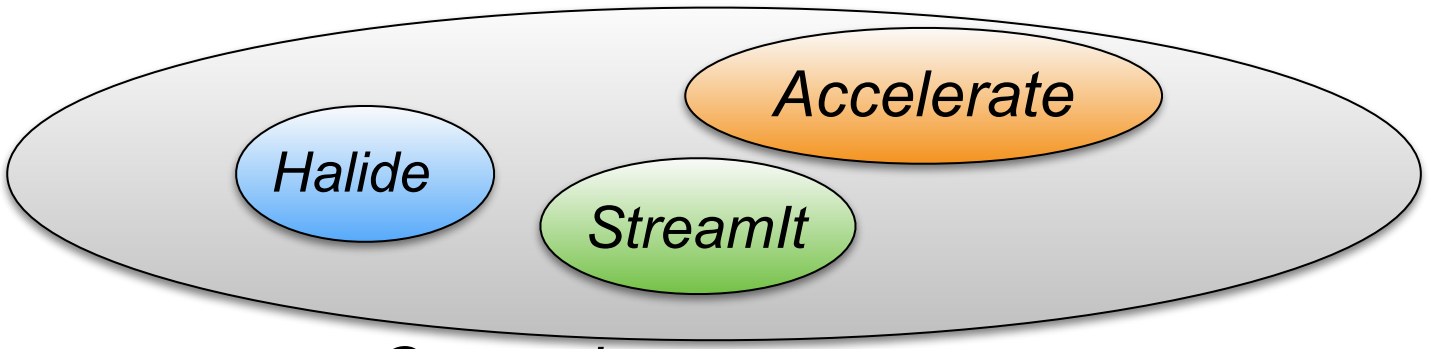
DSL compilers provide small targets

- Combination of
  - restricting lang features
  - impose structure (stream graph, stencil, etc)
- How best to hit them?



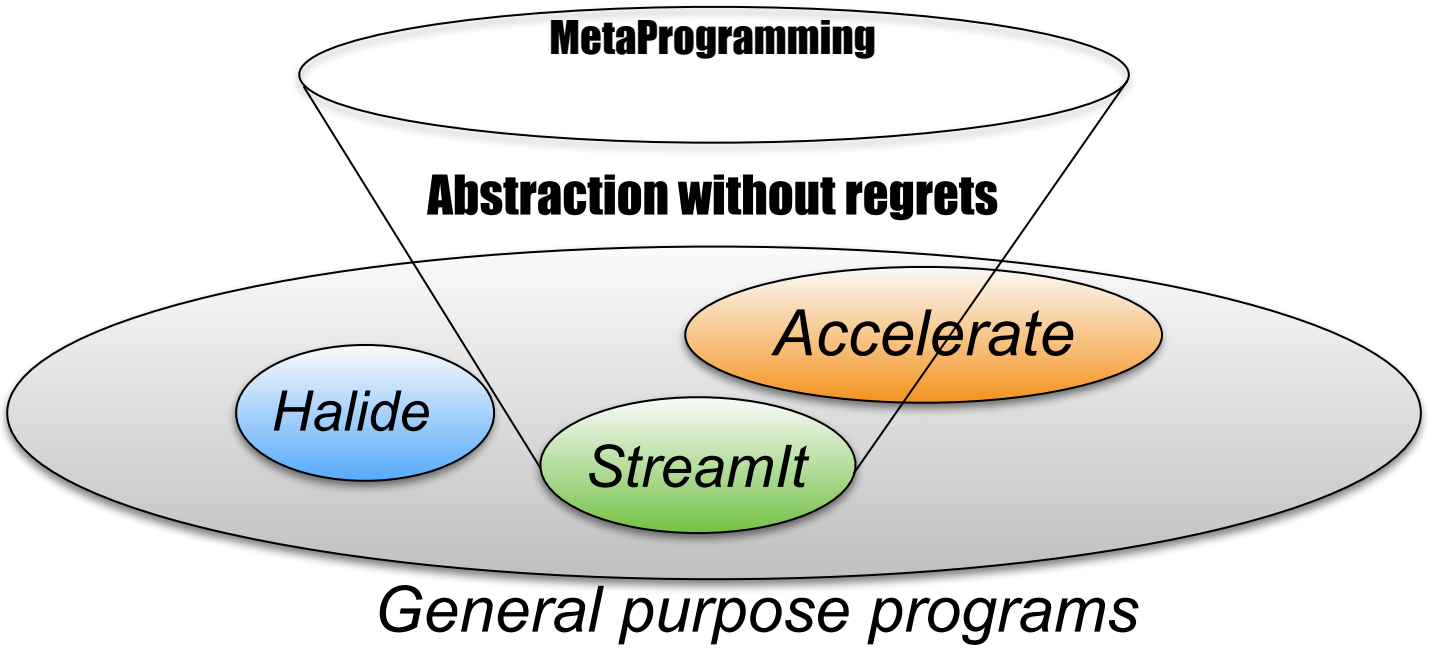
DSL compilers provide small targets

- Combination of
  - restricting lang features
  - impose structure (stream graph, stencil, etc)
- How best to hit them?



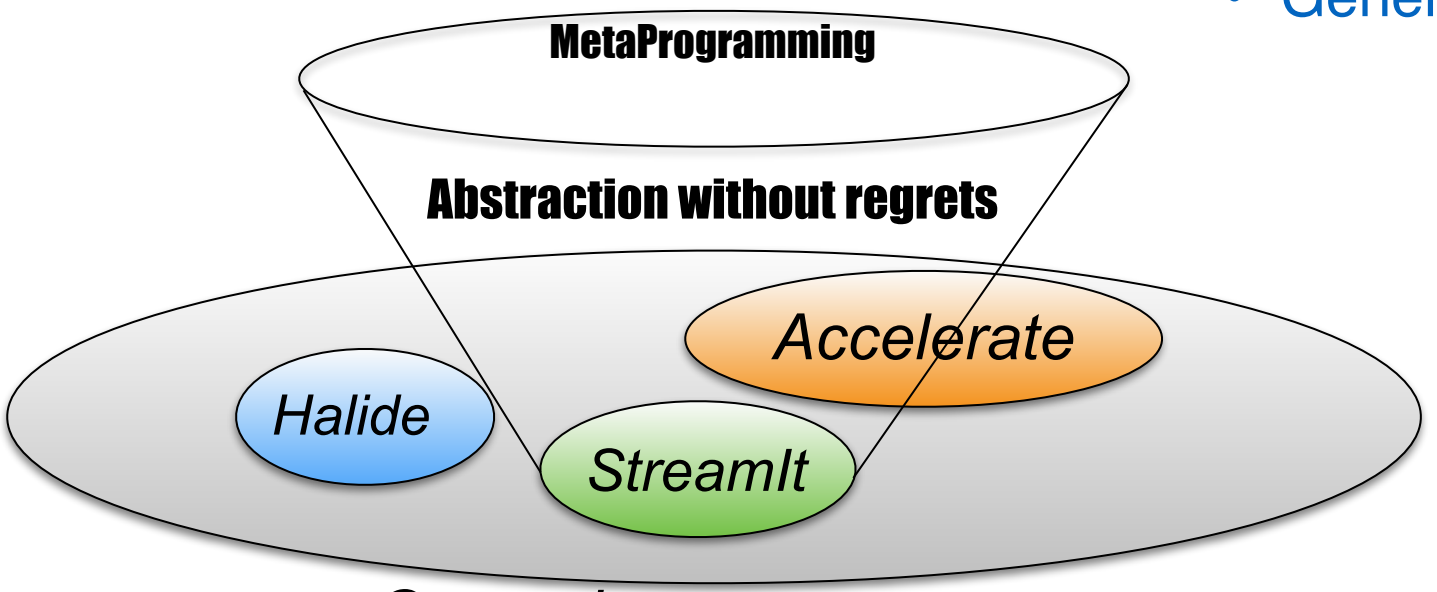
DSL compilers provide small targets

- Combination of
  - restricting lang features
  - impose structure (stream graph, stencil, etc)
- How best to hit them?



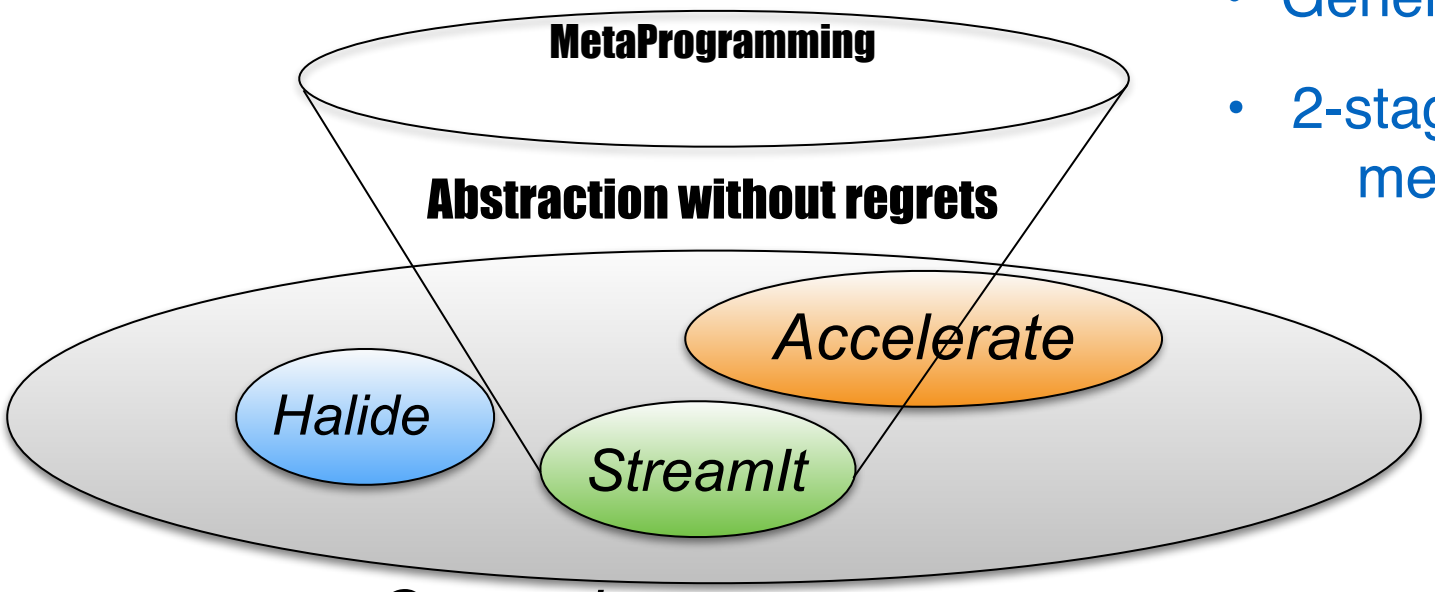
DSL compilers provide small targets

- Combination of
  - restricting lang features
  - impose structure (stream graph, stencil, etc)
- How best to hit them?
  - General staging



DSL compilers provide small targets

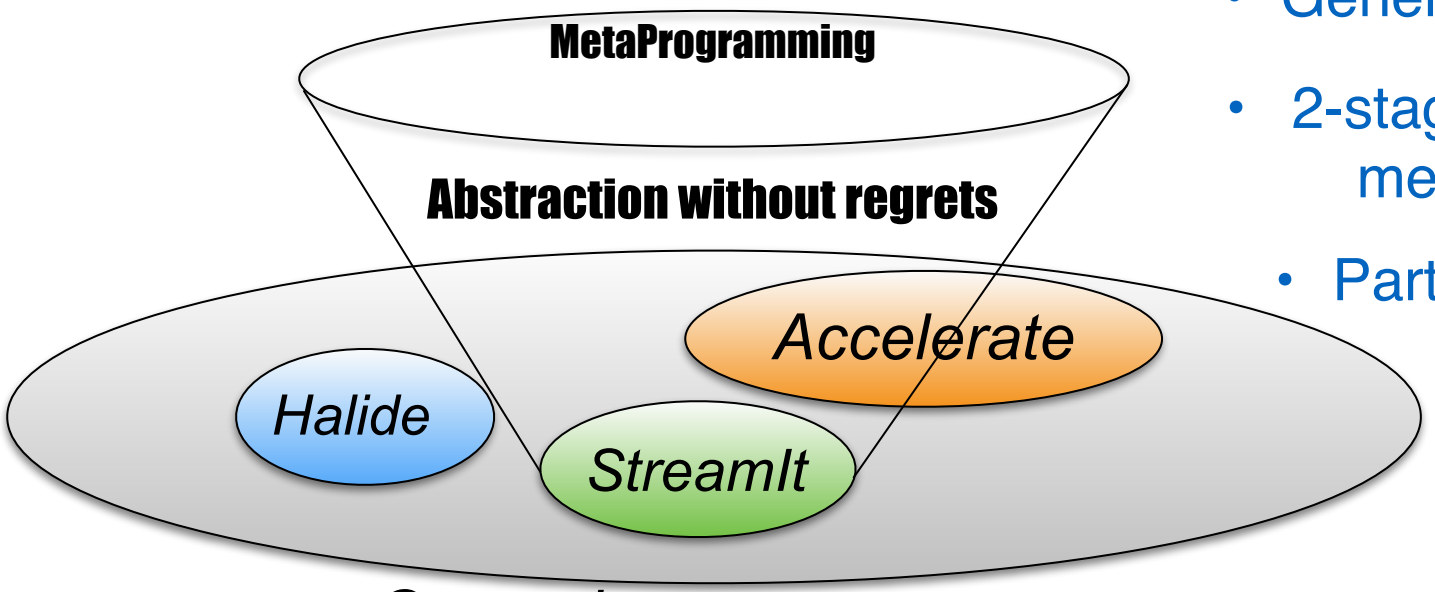
- Combination of
  - restricting lang features
  - impose structure (stream graph, stencil, etc)
- How best to hit them?



- General staging
- 2-stage hetero. metaprog.

DSL compilers provide small targets

- Combination of
  - restricting lang features
  - impose structure (stream graph, stencil, etc)
- How best to hit them?



- General staging
- 2-stage hetero. metaprog.
- Partial Eval.

Part 2: Ease of use?

INDIANA UNIVERSITY

Research langs:

- Copperhead, Sh, ArBB,
- Nikola, Obsidian, Accelerate,
- Harlan, NOVA

By practitioners, for practitioners:

- R, NumPy
- Theano, Torch

Research langs:

- Copperhead, Sh, ArBB,
- Nikola, Obsidian, Accelerate,
- Harlan, NOVA

By practitioners, for practitioners:

- R, NumPy
- Theano, Torch

What are we missing?
Are we being honest about state of lib support, compiler support, etc?
Do we provide the most important bits?

Ease of use? REPL time - NumPy

Ease of use? REPL time - NumPy

```
>>> from numpy import *
```

Ease of use? REPL time - NumPy

```
>>> from numpy import *
>>> arange(10)
```

Ease of use? REPL time - NumPy

```
>>> from numpy import *
>>> arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Ease of use? REPL time - NumPy

```
>>> from numpy import *
>>> arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> arange(10).reshape(2,5)
```

Ease of use? REPL time - NumPy

```
>>> from numpy import *
>>> arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> arange(10).reshape(2,5)
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

REPL time - Accelerate

REPL time - Accelerate

```
> import Data.Array.Accelerate
```

REPL time - Accelerate

```
> import Data.Array.Accelerate
> :t generate
```

REPL time - Accelerate

```
> import Data.Array.Accelerate
> :t generate
generate
  :: (Elt a, Shape ix) =>
     Exp ix ->
     (Exp ix -> Exp a) ->
     Acc (Array ix a)
```

REPL time - Accelerate

REPL time - Accelerate

```
> generate 10 id
```

REPL time - Accelerate

```
> generate 10 id
> generate 10 (\x -> x)
```

REPL time - Accelerate

```
> generate 10 id
> generate 10 (\x -> x)
```

```
No instance for (Elt a0) arising from a use of ‘print’
    The type variable ‘a0’ is ambiguous
    Note: there are several potential instances:
      instance Elt ()
        -- Defined in ‘Data.Array.Accelerate.Array.Sugar’
      instance (Elt a, Elt b) => Elt (a, b)
        -- Defined in ‘Data.Array.Accelerate.Array.Sugar’
      instance (Elt a, Elt b, Elt c) => Elt (a, b, c)
        -- Defined in ‘Data.Array.Accelerate.Array.Sugar’
      ...plus 38 others
```

REPL time - Accelerate

```
> generate 10 (\x -> x) :: Acc (Vector Int)

Couldn't match type ‘DIM0 :. Int’ with ‘Int’
    Expected type: Exp Int
      Actual type: Exp DIM1
    In the expression: x
```

REPL time - Accelerate

REPL time - Accelerate

```
> generate 10
    (\x -> let (Z :. i) = unlift x
           in i)
     :: Acc (Vector Int)
```

REPL time - Accelerate

```
> generate 10
    (\x -> let (Z :. i) = unlift x
           in i)
     :: Acc (Vector Int)
```

```
No instance for (IsNum (DIM0 :. Int))
arising from the literal ‘10’
In the first argument of ‘generate’, namely
‘10’
```

REPL time - Accelerate

REPL time - Accelerate

```
> generate (Z :. 10)
    (\x -> let (Z :. i) = unlift x
           in i)
     :: Acc (Vector Int)
```

REPL time - Accelerate

```
> generate (Z :. 10)
    (\x -> let (Z :. i) = unlift x
           in i)
     :: Acc (Vector Int)
```

```
Couldn't match
   expected type ‘Exp DIM1’
with actual type ‘Z :. head0’
In the first argument of ‘generate’, namely
‘(Z :. 10)’
```

REPL time - Accelerate

REPL time - Accelerate

```
> generate (constant (Z :. 10))
    (\x -> let (Z :. i) = unlift x
           in i)
     :: Acc (Vector Int)
```

REPL time - Accelerate

```
> generate (constant (Z :. 10))
    (\x -> let (Z :. i) = unlift x
           in i)
     :: Acc (Vector Int)
```

```
generate Z :. 10 (\x0 -> indexHead x0)
```

REPL time - Accelerate

REPL time - Accelerate

```
> import Data.Array.Accelerate.Interpreter
> run $ generate (constant (Z :. 10))
    (\x -> let (Z :. i) = unlift x
           in i)
     :: Vector Int
```

REPL time - Accelerate

```
> import Data.Array.Accelerate.Interpreter
> run $ generate (constant (Z :. 10))
    (\x -> let (Z :. i) = unlift x
           in i)
     :: Vector Int

Array (Z :. 10) [0,1,2,3,4,5,6,7,8,9]
```

REPL time - Accelerate

```
> import Data.Array.Accelerate.Interpreter
> run $ generate (constant (Z :. 10))
    (\x -> le
           in
     :: Vecto

Array (Z :. 10) [0,1,2,3,4,5,6,7,8,9]
```

Caveat: `(Z :. 10 :. 20)` is usually `(Z :. (10::Int) :. (20::Int))`

REPL time - Accelerate

```
> import Data.Array.Accelerate.Interpreter
> run $ generate (constant (Z :. 10))
    (\x -> le
           in
     :: Vecto

Array (Z :. 10) [0,1,2,3,4,5,6,7,8,9]
```

Caveat: `(Z :. 10 :. 20)` is usually `(Z :. (10::Int) :. (20::Int))`

Caveat: no 2D or higher printing

REPL time - Accelerate

```
> run $ generate (index1 10) unindex1
        :: Vector Int

Array (Z :. 10) [0,1,2,3,4,5,6,7,8,9]
```

REPL time - [Easy] Accelerate

REPL time - [Easy] Accelerate

```
> p0
```

REPL time - [Easy] Accelerate

```
> p0
|  1  2  3  4  5 |
|  6  7  8  9 10 |
```

REPL time - [Easy] Accelerate

```
> p0
|  1  2  3  4  5 |
|  6  7  8  9 10 |

> :t p0
p0 :: Matrix Int64
```

REPL time - [Easy] Accelerate

```
> p0
|  1  2  3  4  5 |
|  6  7  8  9 10 |

> :t p0
p0 :: Matrix Int64

> p0 + p0
|  2  4  6  8 10 |
| 12 14 16 18 20 |
```

REPL time - [Easy] Accelerate

```
> p0
|  1  2  3  4  5 |
|  6  7  8  9 10 |

> :t p0
p0 :: Matrix Int64

> p0 + p0
|  2  4  6  8 10 |
| 12 14 16 18 20 |
```

Part 3: Alternatives to embedding

INDIANA UNIVERSITY

Alternative 1: Quotation-free metaprog

- "*Quotation*" includes:
  - `(foo ,x)  #`(foo #,x) $[foo ..] .< 8 - .~y .>
  - ```
    let (x,y) = lift (f x)
     in unlift (x + constant z, y)
    ```
- Quotation-free ⟹ partial eval.
  - Phase ambiguity?
  - Not if
    - 2 stage only
    - feature exists in *only* meta or target/object lang.
    ```

Sans embedding tricks

```
let x :: Exp Int
    y :: Exp Float
    (x,y) = lift (f x)
 in unlift (x + constant z, y)
      :: Exp (Int,Float)
```

V.S.

```
let (x,y) = (f x)
 in (x + z, y)
```

Sans embedding tricks

```
let x :: Exp Int
    y :: Exp Float
    (x,y) = lift (f x)
 in unlift (x + constant z, y)
     :: Exp (Int,Float)
```

V.S.

```
let (x,y) = (f x)
 in (x + z, y)
```

- Use haskell-src-exts, etc to parse standalone files.
- Enforce language subsetting (syntactically).
- Reuse existing Haskell type checker.

But… still want some abstraction. Simplest proposal:

```
p := map (λv.e) p | fold (λv1 v2.e) e p
   | v | let v=p in p | …
e := e+e | e*e | if e e e | …

M := λv.M | M M | M+M | if M M M | …
```

But… still want some abstraction. Simplest proposal:

```
p := map (λv.e) p | fold (λv1 v2.e) e p
   | v | let v=p in p | …
e := e+e | e*e | if e e e | …

M := λv.M | M M | M+M | if M M M | …
```

Hypothesis: people don't really use that much functionality in their metaprogs.

But… still want some abstraction. Simplest proposal:

- Expand source file into
  - 1. real haskell code
  - 2. a subset-validated AST (phase polymorphism)
- When codegen invoked, partially evaluate; succeed or fail
- Optimize if desired.

Partial eval. error messages can be surprisingly comprehensible

Partial eval. error messages can be surprisingly comprehensible

- `map (if x y z) a`

Partial eval. error messages can be surprisingly comprehensible

- `map (if x y z) a`

- "Expected x to be compile-time computable"

Partial eval. error messages can be surprisingly comprehensible

- `map (if x y z) a`

- "Expected x to be compile-time computable"

Partial eval. error messages can be surprisingly comprehensible

- `map (if x y z) a`
- "Expected x to be compile-time computable"
- Meta-prog runtime is *still compile-time* in the Accelerate case.

Partial eval. error messages can be surprisingly comprehensible

- `map (if x y z) a`
- "Expected x to be compile-time computable"
- Meta-prog runtime is *still compile-time* in the Accelerate case.
  - No fundamental advantage of type errors over other compile-time errors.

Partial eval. error messages can be surprisingly comprehensible

- `map (if x y z) a`
- "Expected x to be compile-time computable"
- Meta-prog runtime is *still compile-time* in the Accelerate case.
  - No fundamental advantage of type errors over other compile-time errors.
  - Tracking source locations is always good

Some languages already do a (possibly sloppy) form of implicit staging

- StreamIt - except not with very good error messages
- WaveScript - my thesis, embodies approach on prev. slides

Experience with “fuzzy” staging

```
S2 = iterate x in S {
       emit x;
       if (even(x)) emit x;
     };
```

Experience with “fuzzy” staging

```
fun foo(n,S) {
  if (n==0) S
  else {
   S2 = iterate x in S {
          emit x;
          if (even(x)) emit x;
        };
   foo(n-1,S2)
  }
}
```

Experience with "fuzzy" staging

```
fun foo(n,S) {
  if (n==0) S
  else {
   S2 = iterate x in S {
          emit x;
          if (even(x)) emit x;
        };
   foo(n-1,S2)
  }
}
```

Also: staging *optional* (e.g. partial eval)

I don't believe common EDSL tropes

- "Easier because you don't have to implement a parser"
- "Easier because you reuse the host std lib"
- "Easier because you avoid retraining in terms of syntax, etc"

Alternative 2: Embed in a more customizable host language

- Racket is a good candidate:

- ```
  (acc (let [(x y) (ref a1 i)]
         (map (+ x) a2)))
  ```

- IDE support, errors, custom type checker.
  ```

Proposal: all of the above!

- DSL compilers should (usually) be front-end agnostic
  - really portable = one .h and one .so

- We're thinking about doing an online user study
  - compare DSL frontends, e.g.:
    - Acc. traditional embedded vs.
    - Acc. w/easy wrappers vs.
    - Acc. Racket embedded vs.
    - Acc. language subsetting + partial eval