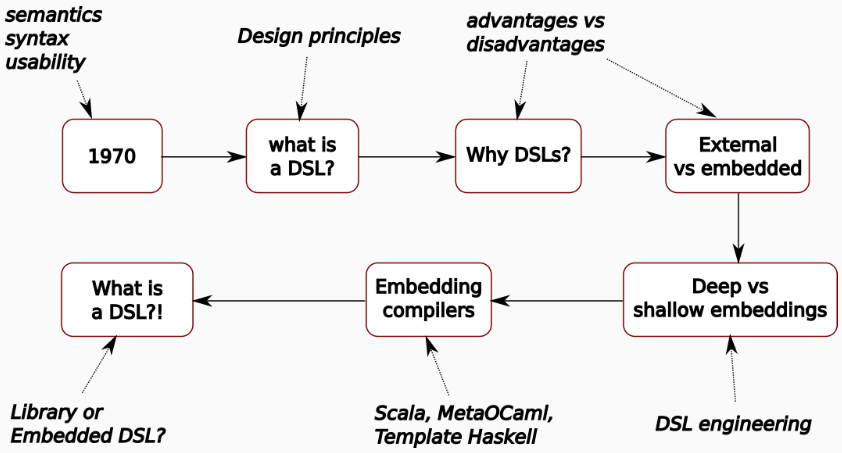
(Embedded) Domain Specific Languages

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

Heriot-Watt University, Edinburgh

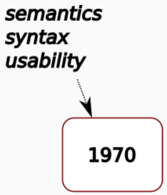

semantics
syntax
usability
Design principles
advantages vs disadvantages
1970
what is a DSL?
Why DSLs?
External vs embedded
Deep vs shallow embeddings
Embedding compilers
What is a DSL?!
Library or Embedded DSL?
Scala, MetaOCaml, Template Haskell
DSL engineering

Rob Stewart

Assistant Prof. at Heriot-Watt University in Edinburgh

1. Applications
    - vision, deep learning, symbolic computing
2. Languages
    - Functional programmer
    - Develop high level parallel languages
    - Formalisms with semantics & verification techniques
3. Heterogeneous architectures
    - Multicore CPUs, GPUs, FPGAs, HPC

DSLs connect these naturally

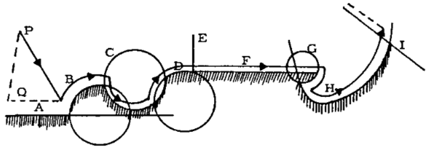

semantics
syntax
usability
1970

Automatically Programmed Tool (APT)

For numerically controlled cutting machine tools

Predecessor to Computer Aided Manufacturing (CAM) systems

H. EXAMPLE



```
        FED RT = +80. $$
          FROM / P $$
       DNT CT, GO TO / Q $$
TL LFT, DNT CT, GO LFT, NEAR / A,B $$
         GO LFT / B,C $$
         GO RGT / C,D $$
         GO LFT / D,E $$
       FAR, CROSS / F,G, $$
      NEAR,GO CLW / G,H $$
         GO CCW / H,I $$
     TL RGT,TERM,GO LFT / I $$
            STOP $$
            END $$
```

NOTE: ANY INSTRUCTION HERE CAN HAVE FEEDRATE GIVEN BEFORE "$$"

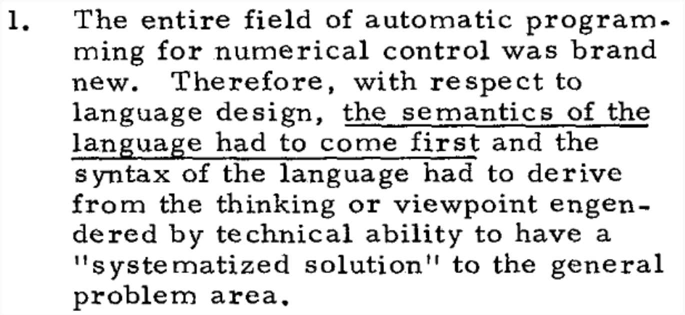
IF ONLY ONE SYMBOL IS USED IT IS THE DESTINATION CURVE (EXCEPT FOR "TERM") I. E. COULD HAVE

```
"- - - - - - - - - - - - - - - - -
GO LFT / B,C $$
GO RGT / D $$
- - - - - - - - - - - - - - - - - "
```

INSTEAD OF

```
"- - - - - - - - - - - - - - - - -
GO LFT / B,C $$
GO RGT / C,D $$
- - - - - - - - - - - - - - - - - "
```

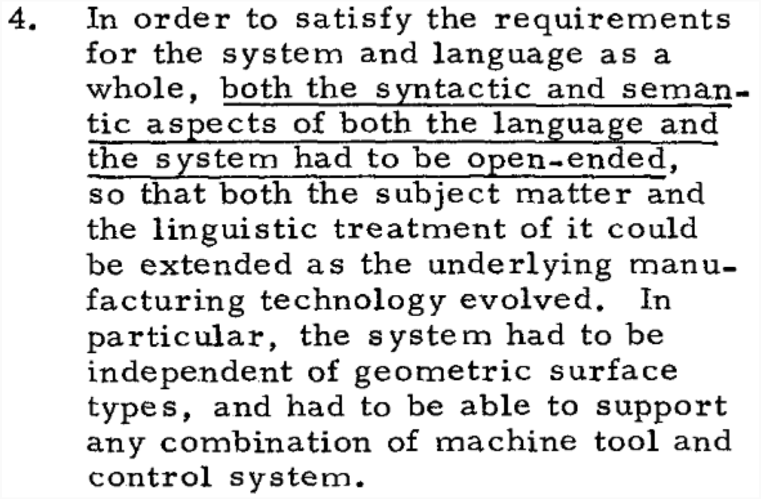
Semantics



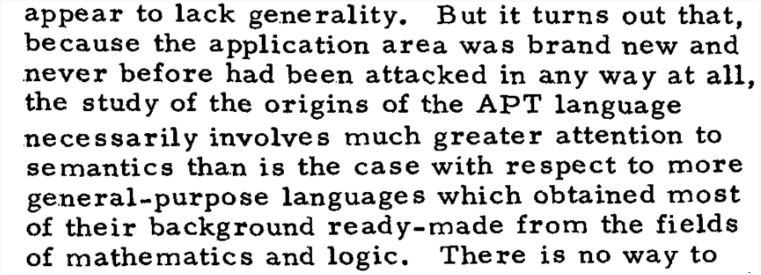
1. The entire field of automatic programming for numerical control was brand new. Therefore, with respect to language design, <u>the semantics of the language had to come first</u> and the syntax of the language had to derive from the thinking or viewpoint engendered by technical ability to have a "systematized solution" to the general problem area.

Douglas T. Ross. "Origins of the APT Language for Automatically Programmed Tools". In: *SIGPLAN Not.* 13.8 (Aug. 1978), pp. 61–99.

Future Proofing

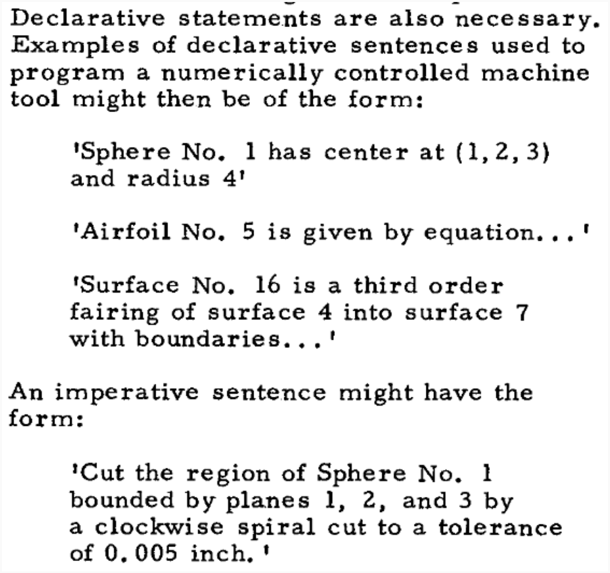

4. In order to satisfy the requirements for the system and language as a whole, both the syntactic and semantic aspects of both the language and the system had to be open-ended, so that both the subject matter and the linguistic treatment of it could be extended as the underlying manufacturing technology evolved. In particular, the system had to be independent of geometric surface types, and had to be able to support any combination of machine tool and control system.

Domain specificity



appear to lack generality. But it turns out that, because the application area was brand new and never before had been attacked in any way at all, the study of the origins of the APT language necessarily involves much greater attention to semantics than is the case with respect to more general-purpose languages which obtained most of their background ready-made from the fields of mathematics and logic. There is no way to

Declarative or Imperative?

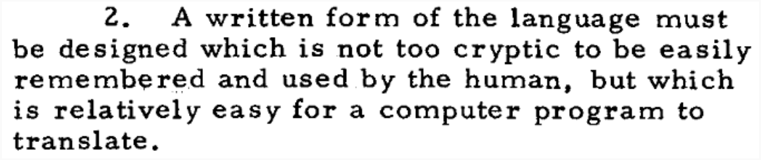

Declarative statements are also necessary. Examples of declarative sentences used to program a numerically controlled machine tool might then be of the form:

> 'Sphere No. 1 has center at (1, 2, 3) and radius 4'
>
> 'Airfoil No. 5 is given by equation...'
>
> 'Surface No. 16 is a third order fairing of surface 4 into surface 7 with boundaries...'

An imperative sentence might have the form:

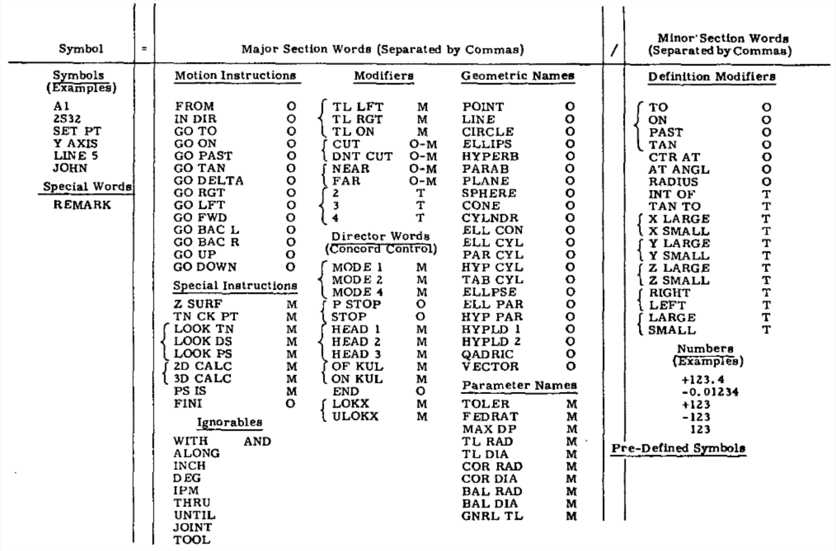
> 'Cut the region of Sphere No. 1 bounded by planes 1, 2, and 3 by a clockwise spiral cut to a tolerance of 0.005 inch.'

Nomenclature and program layout



2. A written form of the language must be designed which is not too cryptic to be easily remembered and used by the human, but which is relatively easy for a computer program to translate.

APT Vocabulary



Symbol	=	Major Section Words (Separated by Commas)	/	Minor Section Words (Separated by Commas)

Symbols (Examples)

A1
2S32
SET PT
Y AXIS
LINE 5
JOHN

Special Words

REMARK

Motion Instructions

FROM	O
IN DIR	O
GO TO	O
GO ON	O
GO PAST	O
GO TAN	O
GO DELTA	O
GO RGT	O
GO LFT	O
GO FWD	O
GO BAC L	O
GO BAC R	O
GO UP	O
GO DOWN	O

Special Instructions

Z SURF	M
TN CK PT	M
LOOK TN	M
LOOK DS	M
LOOK PS	M
2D CALC	M
3D CALC	M
PS IS	M
FINI	O

Ignorables

WITH AND
ALONG
INCH
DEG
IPM
THRU
UNTIL
JOINT
TOOL

Modifiers

TL LFT	M
TL RGT	M
TL ON	M
CUT	O-M
DNT CUT	O-M
NEAR	O-M
FAR	O-M
2	T
3	T
4	T

Director Words (Concord Control)

MODE 1	M
MODE 2	M
MODE 4	M
P STOP	O
STOP	O
HEAD 1	M
HEAD 2	M
HEAD 3	M
OF KUL	M
ON KUL	M
END	O
LOKX	M
ULOKX	M

Geometric Names

POINT	O
LINE	O
CIRCLE	O
ELLIPS	O
HYPERB	O
PARAB	O
PLANE	O
SPHERE	O
CONE	O
CYLNDR	O
ELL CON	O
ELL CYL	O
PAR CYL	O
HYP CYL	O
TAB CYL	O
ELLPSE	O
ELL PAR	O
HYP PAR	O
HYPLD 1	O
HYPLD 2	O
QADRIC	O
VECTOR	O

Parameter Names

TOLER	M
FEDRAT	M
MAX DP	M
TL RAD	M
TL DIA	M
COR RAD	M
COR DIA	M
BAL RAD	M
BAL DIA	M
GNRL TL	M

Definition Modifiers

TO	O
ON	O
PAST	O
TAN	O
CTR AT	O
AT ANGL	O
RADIUS	O
INT OF	T
TAN TO	T
X LARGE	T
X SMALL	T
Y LARGE	T
Y SMALL	T
Z LARGE	T
Z SMALL	T
RIGHT	T
LEFT	T
LARGE	T
SMALL	T

Numbers (Examples)

+123.4
-0.01234
+123
-123
123

Pre-Defined Symbols

DSL design

Language design historically hard!

- Semantics
- Future proofing
- Domain specificity
- Declarative or imperative
- Ease of use

Hard in 1978 … still hard now!

These are not new questions to ask

Ross, "Origins of the APT Language for Automatically Programmed Tools".

Fast forward 43 years from 1978

PERL for text manipulation; VHDL for hardware description; LaTeX for typesetting; HTML for document markup; SQL for database transactions; Maple for symbolic computing; AutoCAD for computer aided design; Prolog for logic ...

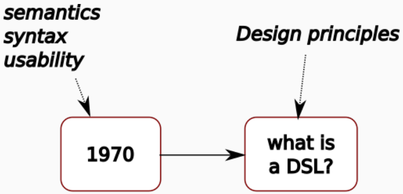
Excel macros
for spreadsheets and many things never intended

– Hudak

Paul Hudak. “Domain Specific Languages”. In: ed. by Peter Salus. Vol. 3. Handbook of Programming Languages, Little Languages and Tools. MacMillan, Indiana, 1998. Chap. 3.

This talk

- ”What” is a DSL
- ”Why” bother
- ”How” to build them
- Embedded DSLs


semantics
syntax
usability
Design principles
1970
what is a DSL?

Paul Hudak: ”A DSL is…”

- Programming language geared for application domain
- Capture semantics of a domain, no more no less
- User immersed in domain knows domain semantics
- Just need a notation to express those semantics

Hudak, “Domain Specific Languages”.

DSL Design Guidelines

1. Choose a domain
2. Design DSL to capture semantics of domain
3. Use KISS (keep it simple, stupid) principle
4. “Little languages” are a Good Thing
5. Concentrate on domain semantics; not too much on syntax
6. Prototype your design, refine, iterate
7. Build tools to support the DSL
8. Develop applications with the DSL
9. Keep end user in mind; Success = A Happy Customer

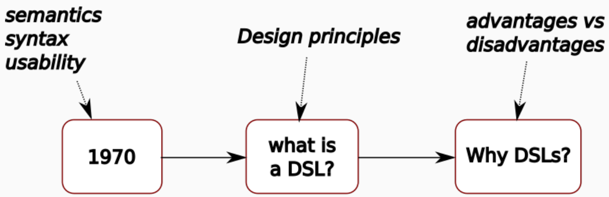
Hudak, “Domain Specific Languages”.

Domain Specificity

DSLs ACM Computing Survey:

- Some say Cobol a DSL for business applications, others say this pushes notion of application domain too far
- Think of DSLs in terms of a gradual scale: specialised DSLs e.g. BNF on left and GPLs such as C++ on right
- Domain-specificity is a matter of degree

Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Comput. Surv.* 37.4 (2005), pp. 316–344.

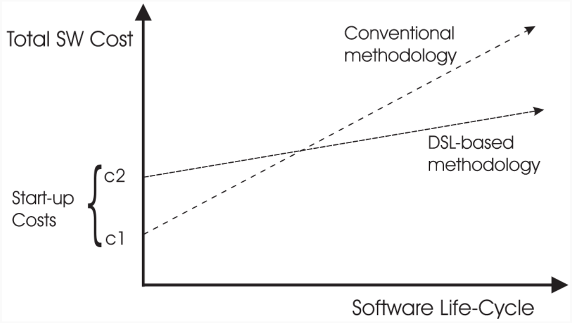

semantics
syntax
usability
Design principles
advantages vs
disadvantages
1970
what is
a DSL?
Why DSLs?

DSL Advantages

1. More concise: easy to look at, see, think about, show
2. Programmer productivity
    - DSLs tend to be high level meaning shorter programs
3. Programs easier to maintain
    - less code ==> less maintenance
4. Reasoning power
    - express programs at level of problem domain
    - domain knowledge conserved, validated, and reused

Debasish Ghosh. *DSLs in Action*. Greenwich, CT, USA: Manning Publications Co., 2010.

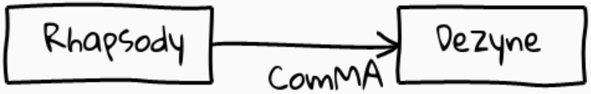
The DSL pay off



- Initial DSL costs high, but software development costs low
- Should eventually start saving time and money

Hudak, "Domain Specific Languages".

DSLs: Return On Investment



- Rhapsody: UML software modelling
- Philips had issues with Rhapsody (see paper)
- Dezyne another modelling language
- Dezyne verifies live-lock freedom and determinism
- Philips developed ComMA DSL

Mathijs Schuts, Jozef Hooman, and Paul Tielemans. “Industrial Experience with the Migration of Legacy Models using a DSL”. In: *Real World Domain Specific Languages Workshop, Vienna, Austria, February 24*. 2018, 1:1–1:10.

DSLs: Return On Investment

- Manual: 576 hours (16 person weeks)
  - manual transformation of 8 state machines
- Automated: 190 hours to develop automation
  - 60 hours: Rhapsody input, Dezyne output with ComMA
  - 15 hours: model learning, equivalence checking
  - 25 hours: Visual Studio integration
  - 90 hours: develop additional state machine support

$$ROI = \frac{gain\ from\ investment - cost\ of\ investment}{cost\ of\ investment}$$

ROI = (576 − 190)/190 ≈ 2

Schuts, Hooman, and Tielemans, “Industrial Experience with the Migration of Legacy Models using a DSL”.

Other motivations for DSLs

- Familiar notation for domain experts - SQL
- High level abstraction - Keras
- Speed - Halide
- Productivity - Halide
- Correctness - Ivory

Speed and Productivity

Halide

- High performance C++ embedded image/array processing
- Separates *algorithm* from *scheduling* code
- ”Why DSL?” - programmer productivity
    - Halide 24 lines, PyTorch 42 lines, CUDA 308 lines
- ”Why DSL?” - fast performance
    - Halide 10x faster than CUDA, 20x faster than PyTorch
- Halide extensions: Automatic Differentiation, Scheduling

Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *ACM SIGPLAN PLDI, Seattle, WA, USA, June*. 2013, pp. 519–530.

```
Func blur_3x3(Func input) {
  Func blur_x, blur_y;
  Var x, y, xi, yi;


  // The algorithm - no storage or order
  blur_x(x, y) = (input(x-1, y)  + input(x, y)  + input(x+1, y))/3;
  blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;


  // The schedule - defines order, locality; implies storage
  blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);

  // execute algorithm with the schedule
  blur_x.compute_at(blur_y, x).vectorize(x, 8);
  return blur_y;
}
```

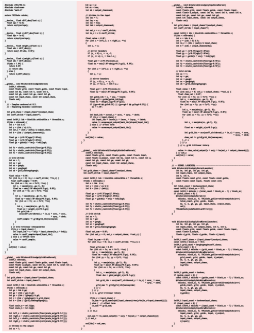
Productivity

```
// Slice an affine matrix from the grid and
// transform the color
Expr gx = cast<float>(x)/sigma_s;
Expr gy = cast<float>(y)/sigma_s;
Expr gz =
  clamp(guide(x,y,n),0.f,1.f)*grid.channels();
Expr fx = cast<int>(gx);
Expr fy = cast<int>(gy);
Expr fz = cast<int>(gz);
Expr wx = gx-fx, wy = gy-fy, wz = gz-fz;
Expr tent =
  abs(rt.x-wx)*abs(rt.y-wy)*abs(rt.z-wz);
RDom rt(0,2,0,2,0,2);
Func affine;
affine(x,y,c,n) +=
  grid(fx+rt.x,fy+rt.y,fz+rt.z,c,n)*tent;
Func output;
Expr nci = input.channels();
RDom r(0, nci);
output(x,y,co,n) = affine(x,y,co*(nci+1)+nci,n);
output(x,y,co,n) +=
  affine(x,y,co*(nci+1)+r,n) * in(x,y,r,n);

// Propagate the gradients to inputs
auto d = propagate_adjoints(output, adjoints);
Func d_in = d(in);
Func d_guide = d(guide);
Func d_grid = d(grid);
```

Halide Runtime
24 lines 64 ms (1 MPix)
165 ms (4 MPix)

```
xx = Variable(th.arange(0, w).cuda().view(1, -1).repeat(h, 1))
yy = Variable(th.arange(0, h).cuda().view(-1, 1).repeat(1, w))
gx = ((xx+0.5)/w) * gw
gy = ((yy+0.5)/h) * gh
gz = th.clamp(guide, 0.0, 1.0)*gd
fx = th.clamp(th.floor(gx - 0.5), min=0)
fy = th.clamp(th.floor(gy - 0.5), min=0)
fz = th.clamp(th.floor(gz - 0.5), min=0)
wx = gx - 0.5 - fx
wy = gy - 0.5 - fy
wx = wx.unsqueeze(0).unsqueeze(0)
wy = wy.unsqueeze(0).unsqueeze(0)
wz = th.abs(gz-0.5 - fz)
wz = wz.unsqueeze(1)
fx = fx.long().unsqueeze(0).unsqueeze(0)
fy = fy.long().unsqueeze(0).unsqueeze(0)
fz = fz.long()
cx = th.clamp(fx+1, max=gw-1);
cy = th.clamp(fy+1, max=gh-1);
cz = th.clamp(fz+1, max=gd-1)
fz = fz.view(bs, 1, h, w)
cz = cz.view(bs, 1, h, w)
batch_idx = th.arange(bs).view(bs, 1, 1, 1).long().cuda()
out = []
co = c // (ci+1)
for c_ in range(co):
  c_idx = th.arange((ci+1)*c_, (ci+1)*(c_+1)).view(\
           1, ci+1, 1, 1).long().cuda()
  a = grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(1-wz) + \
         grid[batch_idx, c_idx, cz, fy, fx]*(1-wx)*(1-wy)*(  wz) + \
         grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(1-wz) + \
         grid[batch_idx, c_idx, cz, cy, fx]*(1-wx)*(  wy)*(  wz) + \
         grid[batch_idx, c_idx, fz, fy, cx]*(  wx)*(1-wy)*(1-wz) + \
         grid[batch_idx, c_idx, cz, fy, cx]*(  wx)*(1-wy)*(  wz) + \
         grid[batch_idx, c_idx, fz, cy, cx]*(  wx)*(  wy)*(1-wz) + \
         grid[batch_idx, c_idx, cz, cy, cx]*(  wx)*(  wy)*(  wz)
  o = th.sum(a[:, :-1, ...]*input, 1) + a[:, -1, ...]
  out.append(o.unsqueeze(1))
out = th.cat(out, 1)

out.backward(adjoints)
d_input = input.grad
d_grid = grid.grad
d_guide = guide.grad
```

PyTorch Runtime
42 lines 1440 ms (1 MPix)
out of memory (4 MPix)


[illegible]

CUDA Runtime
308 lines 430 ms (1 MPix)
2270 ms (4 MPix)

Tzu-Mao Li et al. "Differentiable programming for image processing and deep learning in halide". In: *ACM Trans. Graph.* 37.4 (2018), 139:1–139:13.

Correctness

- Ivory: safe systems programming
- Memory and type safety guarantees
- Uses host language's type system
- Syntax is deeply embedded, from one AST:
    - generates ”safe” C code
    - SMT-based symbolic simulator
    - Theorem-prover back-end

Industry strength EDSL:

- Boeing use Ivory to implement level-of-interoperability for a NATO standard interface for Unmanned Control System (UCS) & Unmanned Aerial Vehicle (UAV) interoperability

Trevor Elliott et al. “Guilt free ivory”. In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Vancouver, BC, Canada, September 3-4, 2015.* 2015, pp. 189–200.

Correctness: Ivory

```
fib_loop :: Def ('[Ix 1000] :-> Uint32)
```

- `Def` is Ivory procedure (aka C function)
- `'[Ix 1000] :-> Uint32`
  - takes *index* argument `n`
  - *0 =< n < 1000*
  - this procedure returns unsigned 32 bit integer

```
fib_loop  = proc "fib_loop" $ \ n -> body $ do ...
```

- Ivory `body` func takes argument of type `Ivory eff ()`
- `eff` effect scope enforces type & memory safety

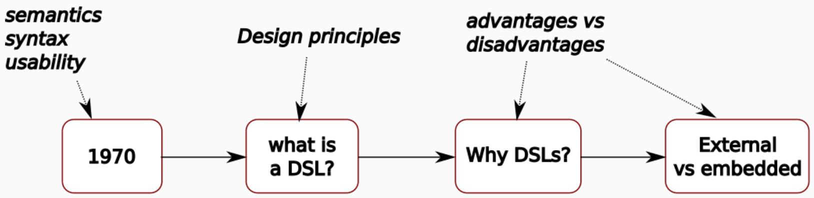
Notice distinguishing feature?

- Internal DSLs
    - Keras (Python)
    - Frenetic (Python)
    - Halide (C++)
    - Ivory (Haskell)
- External DSLs
    - SQL

Embedding of external languages too

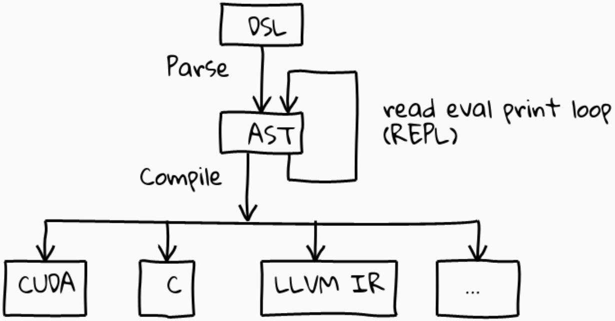
e.g. Selda: a type safe SQL EDSL, Ivory, etc.

Anton Ekblad. “Scoping Monadic Relational Database Queries”. In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany, 2019, pp. 114–124.


semantics
syntax
usability
Design principles
advantages vs
disadvantages
1970
what is a DSL?
Why DSLs?
External vs embedded

External DSLs

1. Parser + Interpreter interactive read–eval–print loop
2. Parser + Compiler DSL constructs to another language



Reasons for externalising DSLs

Syntax

- Domain specific notation not constrained by host's syntax
- Design Syntax notation familiar to domain experts

Domain reasoning

- Domain specific analysis, verification, optimisation, parallelisation and transformation (AVOPT)
- Domain specific error messages

Reasons against externalising

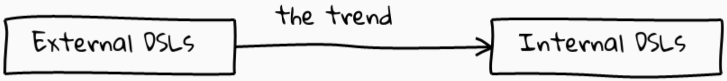
Large engineering effort

- Complex language processor must be implemented
- Parser, interpreter/compiler, debugging tools

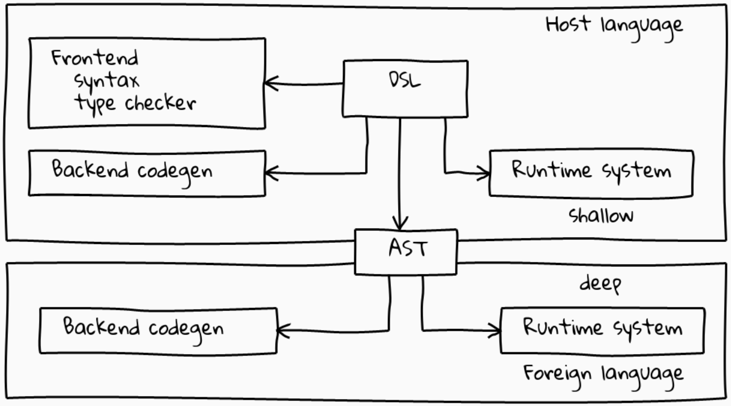
”Correct” abstraction of domain

- DSLs from scratch often lead to incoherent designs
- DSL design is hard. Semantics or syntax first?
- Requires (1) domain and (2) language development expertise. Few people have both.
- Mission creep: programmers want more features

Are we seriously proposing a new language for every domain?


External DSLs
the trend
Internal DSLs

Embedded DSLs



Focus on semantics not syntax

Metaprogramming tools if you don't want host language syntax

Reasons for internalising DSLs

- Modest development effort
- Rapid prototyping
- Many language processor features for free
- Host tooling (debugging, perf benchmarks, editors) for free
- Lower user training costs

Reasons against internalising

Usability

- bad error reporting
- host syntax may be far from optimal
- cannot (easily) introduce arbitrary syntax

Performance

- difficult to implement domain specific code optimisations
- limited support for analysis, verification, optimisation, parallelisation and transformation (AVOPT)
- cannot easily extend compiler

Mernik, Heering, and Sloane, “When and how to develop domain-specific languages”.

"You cannot easily extend compiler"...

But you can! With user defined rewrite rules

Compiler makes no attempt to...

- verify correctness of rules
- ensure rewrite produces more efficient code

```haskell
blur5x5 :: Image -> Image
blur3x3 :: Image -> Image

{-# RULES
  "blur5x5/blur3x3" forall image.
      (blur3x3 (blur3x3 image)) = blur5x5 image
 #-}
```

"Embedded DSLs have bad error reporting"...

Not so! Modern language support customised error messages

```
3 + False
```

```
<interactive>:1:1 error:
    • No instance for (Num Bool) arising from a use of `+'
    • In the expression: 3 + False
      In an equation for `it': it = 3 + False
```

George Wilson. "Functional Programming in Education". YouTube. July 2019.

Custom Error Message

```haskell
import GHC.TypeLits

instance TypeError
    (Text "Booleans are not numbers" :$$:
     Text "so we cannot add or multiply them")
  => Num Bool where ...

3 + False
```

```
<interactive>:1:1 error:
    • Booleans are not numbers
      so we cannot add or multiple them
    • In the expression: 3 + False
      In an equation for `it': it = 3 + False
```

Building Embedded DSLs

Syntax

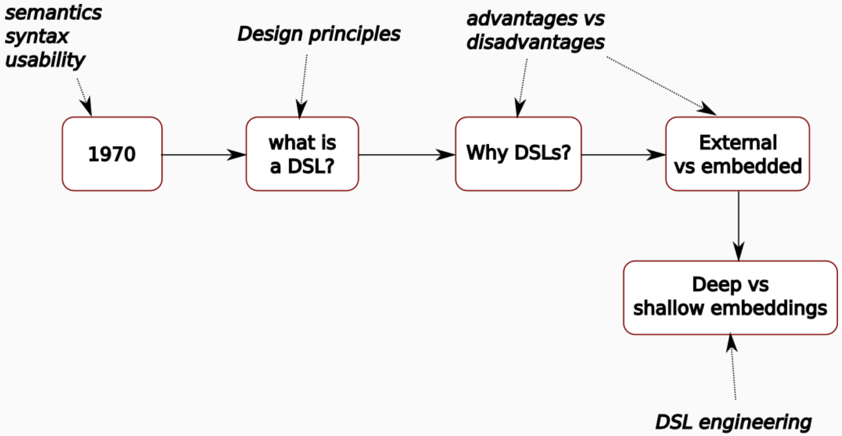
- Pre-processors e.g. macros
- Type embedding - domain types and operators over them

Meta-programming

- Runtime meta-programming e.g. MetaOCaml, Scala LMS
- Compile-time meta-programming e.g. Template Haskell

Custom compilation

- Syntax tree manipulation - deeply embedded compilers
- Extend a compiler for domain specific code generation


semantics
syntax
usability
Design principles
advantages vs disadvantages
1970
what is a DSL?
Why DSLs?
External vs embedded
Deep vs shallow embeddings
DSL engineering

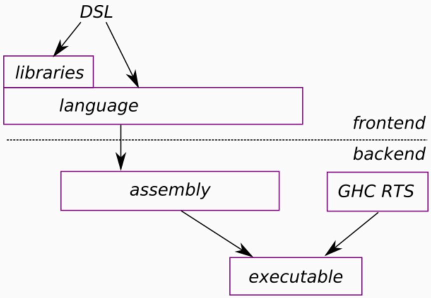
Shallow embedding

- Design types to capture domain
- Implement operators over those domain types

In Haskell a monad might be *the* central construct

Other languages have other design patterns

Shallow Embeddings



Par monad - shallowly embedded for fork/join parallelism

```
newtype Par a
instance Monad Par
data IVar a

runPar :: Par a -> a
spawn  :: NFData a => Par a -> Par (IVar a)
get    :: IVar a -> Par a
```

- Shallow embedding: borrow host's compiler and runtime
- Simple to implement
- Host compiler has no domain knowledge
- Relies on efficiency of host language's backend codegen

Simon Marlow, Ryan Newton, and Simon L. Peyton Jones. "A monad for deterministic parallelism". In: *Haskell Symposium*. 2011, pp. 71–82.

Repa - shallowly embedded array processing language

Goal: array fusion to remove memory IO

- *map/map*, permutation, replication, slicing, etc.

Implementation:

- types for array ”delayed” representations
- fuse function compositions over ”delayed” arrays

Performance:

- at mercy of GHC’s code generation capabilities

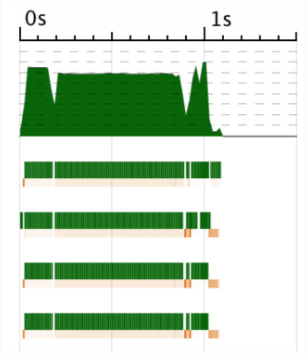
Ben Lippmeier et al. “Guiding parallel array fusion with indexed types”. In: *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*. 2012, pp. 25–36.

```haskell
gradientX :: Monad m => Image -> m Image
gradientX img = computeP
        $ forStencil2 (BoundConst 0) img
          [stencil2|    -1  0  1
                        -2  0  2
                        -1  0  1 |]

gradientY :: Monad m => Image -> m Image
gradientY img = computeP
        $ forStencil2 (BoundConst 0) img
          [stencil2|     1  2  1
                         0  0  0
                        -1 -2 -1 |]

magnitude :: Float -> Float -> Float
magnitude x y = sqrt (x * x + y * y)

sobel img = do
  gX <- gradientX img
  gY <- gradientY img
  zipWith magnitude gX gY
```

Repa types and `computeP` executor

```haskell
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
```

Shallow Embeddings: Repa



Performance profiling: use host tooling

Is that good or bad?

Haskell is general purpose – suggests ”bad”?

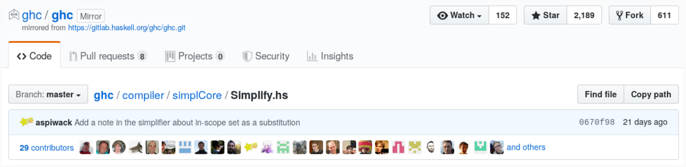
but does have a parallel profiling tool `threadscope`!

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 / ghc Mirror
mirrored from https://gitlab.haskell.org/ghc/ghc.git

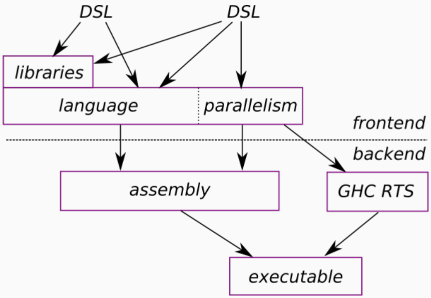
ghc / compiler / simplCore / Simplify.hs

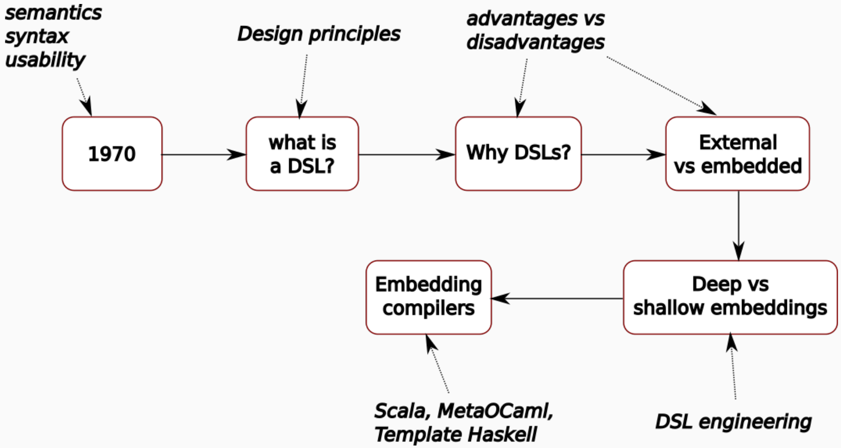
aspiwack Add a note in the simplifier about in-scope set as a substitution 0670f98 21 days ago

29 contributors and others

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

Parallel Shallow Embeddings




semantics
syntax
usability
Design principles
advantages vs
disadvantages
1970
what is a DSL?
Why DSLs?
External vs embedded
Deep vs shallow embeddings
Embedding compilers
Scala, MetaOCaml,
Template Haskell
DSL engineering

Extend host compiler's code generator...

Let's look at three approaches:

1. Deeply embedded compiler
2. Compile time metaprogramming
3. Multi-Stage Programming

Deep EDSL may be ”library + *embedded compiler*”

or

Deep EDSL may be ”library + *embedded runtime system*”

How?

Library functions return *structures* not values

Accelerate

- Functional array language: `map`, `zipWith`, `fold`, ...
- User programs *generate* CUDA/LLVM programs at runtime

```
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

```
run :: Arrays a => Acc a -> a
```

Manuel M. T. Chakravarty et al. “Accelerating Haskell array codes with multicore GPUs”. In: *DAMP 2011, Austin, TX, USA, January 23, 2011*. ACM, 2011, pp. 3–14.

My function:

```haskell
brightenBy :: Int -> Acc Image -> Acc Image
brightenBy i = map (+ (lift i))
```

The *structure* returned is:

```haskell
Map (\x y -> PrimAdd `PrimApp` ...)
```

NOT computed values!

~~[23 98 245 148 198 101 11 3 240 239 201 188 ...]~~

Accelerate Internal IR

```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' )
```

Becomes:

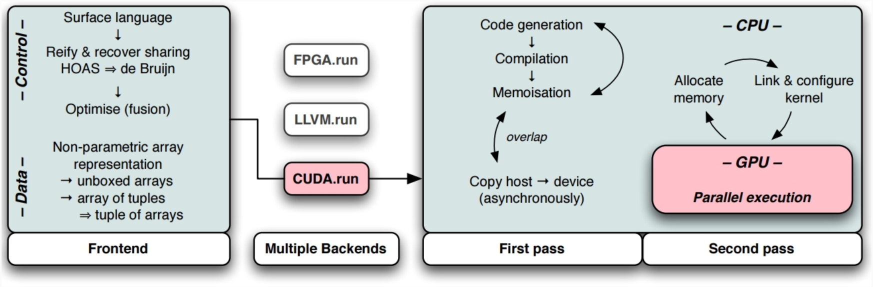
```haskell
Fold add (Const 0) (ZipWith mul xs' ys')
  where
    add = Lam (Lam (Body (
     PrimAdd (FloatingNumType (TypeFloat FloatingDict))
             `PrimApp`
             Tuple (NilTup `SnocTup` (Var (SuccIdx ZeroIdx))
                           `SnocTup` (Var ZeroIdx)))))
    mul = -- same as add, but using PrimMul ...
```

Benefits of preserving ASTs

Things you can do with an Accelerate program

1. Pretty print it
2. Interpret it
3. Optimise program AST at runtime
4. Generate & execute CUDA for GPUs
5. Generate & execute LLVM for CPUs/GPUs
6. Visualise program graph with GraphViz

```
let program = A.zip (f arr1) (g arr2)
print program              -- show it
print print (A.run program) -- run it
```



LLVM/CUDA codegen at (host) runtime

Optimise with runtime knowledge

Similar idea to Multi-Stage Programming (later)

Chakravarty et al., “Accelerating Haskell array codes with multicore GPUs”.

Template based skeleton compilation

Accelerate AST → LLVM IR

e.g. runtime compilation of `map`

```
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_
```

The embedded Haskell → LLVM IR compiler in Accelerate

```haskell
run :: Arrays a => Acc a -> a
run a = unsafePerformIO (runIO a)

runIO :: Arrays a => Acc a -> IO a
runIO a = withPool defaultTargetPool (\target -> runWithIO target a)

runWithIO :: Arrays a => PTX -> Acc a -> IO a
runWithIO target a = execute
  where
    !acc    = convertAcc a
    execute = do
      dumpGraph acc
      evalPTX target $ do
        build <- phase "compile" (compileAcc acc) >>= dumpStats
        exec  <- phase "link"    (linkAcc build)
        res   <- phase "execute"
                    (evalPar (executeAcc exec >>= copyToHostLazy))
        return res
```

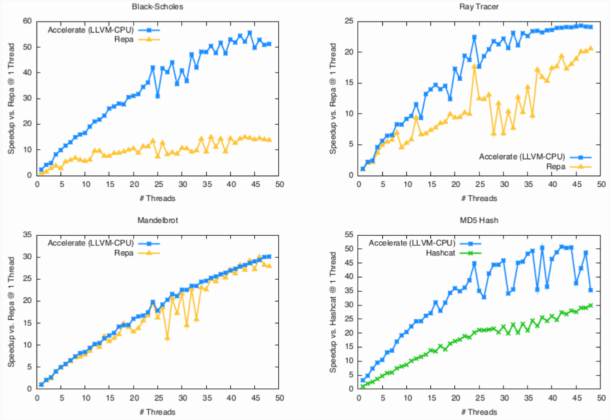
Accelerate's use of host language

- GHC frontend: yes
- GHC code generator backend: no
- GHC runtime system: no

Has multiple backends from Accelerate AST

- LLVM IR
- CUDA

Accelerate performance



Trevor L. McDonell et al. “Type-safe runtime code generation: Accelerate to LLVM”. In: *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Vancouver, BC, Canada, September 3-4, 2015.* ACM, 2015, pp. 201–212.

Accelerate → LLVM IR

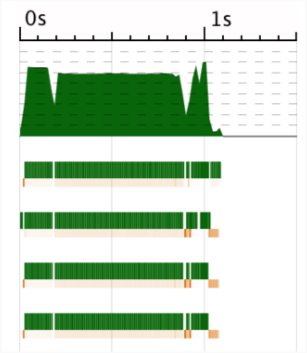
Keep It Simple Stupid (KISS) principle

- Obscure LLVM code might rule out LLVM optimisations
- Generate ”simple” LLVM IR that LLVM will optimise well
- Accelerate tells LLVM exactly which CPU is being used
- LLVM which CPU being used
- LLVM will vectorise for specific CPU architecture

Don’t generate SIMD instructions

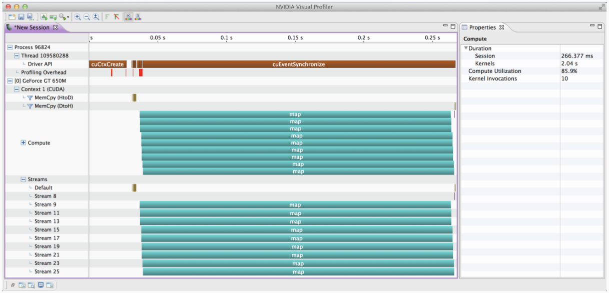
- Rely on LLVM auto-vectorisation
- vectorisation will happen if LLVM thinks it is beneficial
- Accelerate produces code it *knows* LLVM can vectorise well

Repa Profiling



- Repa uses GHC runtime system
- Threadscope for profiling GHC generated parallel code
- Hence: Repa can inherit Threadscope profiling tool

Accelerate Profiling



Duration	
Session	266.377 ms
Kernels	2.04 s
Compute Utilization	85.9%
Kernel Invocations	10

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

Figure 4.2 from McDonell's thesis.

Trevor L. McDonell. "Optimising Purely Functional GPU Programs". PhD thesis. University of New South Wales, Sydney, Australia, 2015.

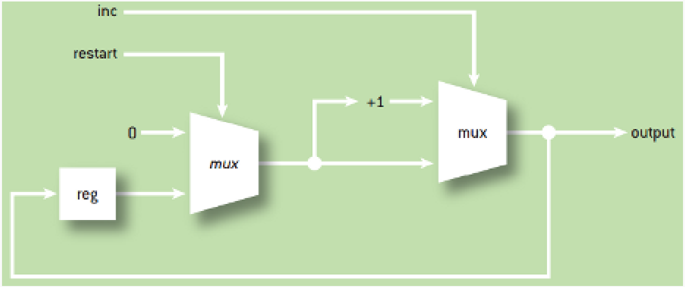
Another deep embedding: hardware design

- 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

Following example from Andy Gill's *ACM Comms* paper.

Andy Gill. "Domain-specific languages and code synthesis using Haskell". In: *Commun. ACM* 57.6 (2014), pp. 42–49.

Counting Pulses Schematic



```
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 (shallow host language interpreter):

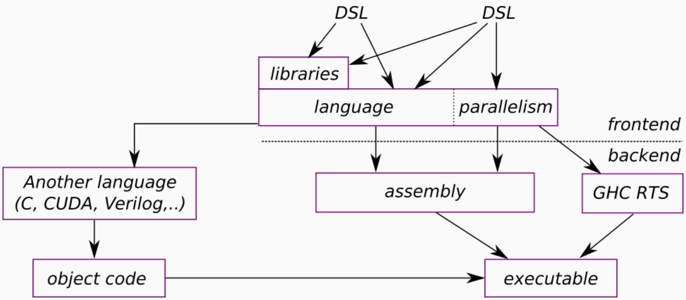
```
GHCi> counter low (toSeq (cycle [True,False,False])
1 : 1 : 1 : 2 : 2 : 2 : 3 : 3 : 3 : ...
```

Reify `counter` to 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)]
```

Deep: Translating Reified AST to VHDL

```
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 <= stf_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 architecture;
```



DSL
DSL
libraries
language
parallelism
frontend
backend
Another language (C, CUDA, Verilog,..)
assembly
GHC RTS
object code
executable

What's wrong with deeply embedded compilers?

- cannot access host language compiler's optimisations!

Shallow embeddings don't suffer this problem, but...

- no domain specific optimisations
- compiler optimisations are general purpose

For example GHC compiler:

- optimiser not designed to perform e.g. array fusion
- "domain" of GHC simplifier is the *lambda calculus*
    - common subexp elimination, unboxing, inlining..

Simon L. Peyton Jones and Simon Marlow. "Secrets of the Glasgow Haskell Compiler inliner". In: *J. Funct. Program.* 12.4&5 (2002), pp. 393–433.

Compile time metaprogramming

Transforms code to syntactic structures

Host language $\rightarrow$ AST transformations $\rightarrow$ host language

All happens at (host) compile time

Sean Seefried, Manuel M. T. Chakravarty, and Gabriele Keller. “Optimising Embedded DSLs Using Template Haskell”. In: *GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings.* Springer, 2004, pp. 186–205.

Compile time metaprogramming with Template Haskell

For a $n \times n$ matrix M,

domain knowledge: $M \times M^{-1} = I$

Host language compiler does not know this matrices property

Consider this Haskell code:

```
m * inverse m * n
```

How can we apply the $M \times M^{-1} = I$ domain knowledge?

Meta-programming approach

1. *reify* code into an AST data structure

   ```
   exp_mat = [| \m n -> m * inverse m * n |]
   ```
2. Apply the AST $\rightarrow$ AST transformation
   This will apply the $M \times M^{-1} = I$ optimisation
3. *reflect* AST back into code (also called *splicing*)

Seefried, Chakravarty, and Keller, "Optimising Embedded DSLs Using Template Haskell".

Apply the optimisation:

```
rmMatByInverse (InfixE (Just 'm) 'GHC.Num.*
                (Just (AppE 'inverse 'm))) =
  VarE (mkName "identity")
```

Otherwise apply optimisation into subexpressions...

Pattern match with $\lambda p.e$

```
rmMatByInverse (LamE pats exp) =
  LamE pats (rmMatByInverse exp)
```

Pattern match with $f\ a$

```
rmMatByInverse (AppE exp exp') =
  AppE (rmMatByInverse exp) (rmMatByInverse exp')
```

And the rest

```
rmMatByInverse exp = exp
```

Our computation:

```
\m n -> m * inverse m * n
```

Reify:

```
exp_mat = [| \m n -> m * inverse m * n |]
```

Splice this back into program:

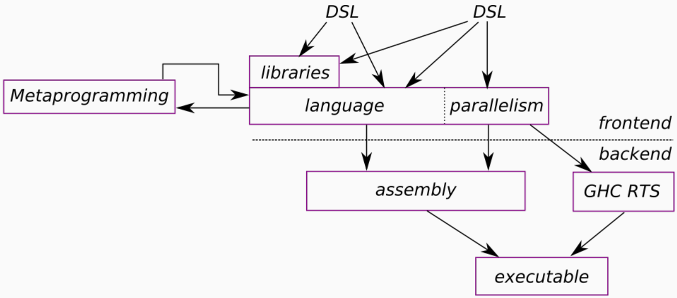
```
$(rmMayByInverse exp_mat)
```

Becomes

```
\m n -> n
```

At compile time. Hurray! We optimised with $M \times M^{-1} = I$

Compile Time Metaprogramming



Staged program = conventional program + staging annotations

Programmer delays evaluation of program expressions

A ”stage” is code generator that constructs *next stage*

Partial evaluation performs constant propagation

Produces intermediate code specialised to static inputs

Code generation at multiple stages during runtime

Multi Stage Programming (MSP) with MetaOCaml

Brackets (`.<..>.`) around expression delays computation

```
# let a = 1+2;;
val a : int = 3
# let a = .<1+2>.;;
val a : int code = .<1+2>.
```

Escape (`.~`) splices in delayed values

```
# let b = .<.~a * .~a >. ;;
val b : int code = .<(1 + 2) * (1 + 2)>.
```

Run (`.!`) compiles and executes code

```
# let c = .! b;;
val c : int = 9
```

Walid Taha. "A Gentle Introduction to Multi-stage Programming". In: *Domain-Specific Program Generation, Dagstuhl Castle, Germany, Revised Papers*. Springer, 2003, pp. 30–50.

MetaOCaml Example

```
let rec power (n, x) =
  match n with
    0 -> 1 | n -> x * (power (n-1, x));;

# power (2,3);;
- : int = 9
# power (4,2);;
- : int = 16
# power (3,5);;
- : int = 125
```

```
let slow_power2  = fun x -> power (2,x);;

slow_power2 3
=> power (2,3)
=> 3 * power (1,3)
=> 3 * (3 * power (0,3))
=> 3 * (3 * 1)
=> 9
```

3x calls to `power`

Memory for stack frames for recursive calls

```
(* what I really want is this *)
let fast_power2 = fun x -> x*x*1;;

# fast_power2 3;;
- : int = 9
```

Specialising code with MetaOCaml

```
let rec power (n, x) =
  match n with
    0 -> .<1>. | n -> .<.~x * .~(power (n-1, x))>.;;
```

`power` function returns code of type integer, not *integer* value

escape of `power` splices in more code

Compile `power2` at *runtime* then execute generated code

```
let power2 = .! .<fun x -> .~(power (2,.<x>.))>.;;
```

generated code behaves just like:

```
fun x -> x*x*1;;
```

We can specialise `power` *at runtime* as needed

```
let power3 = .! .<fun x -> .~(power (3,.<x>.))>.;;
let power4 = .! .<fun x -> .~(power (4,.<x>.))>.;;
```

Zero cost abstraction without the code bloat of

```
let fast_power2 = fun x -> x*x*1;;
let fast_power3 = fun x -> x*x*x*1;;
let fast_power4 = fun x -> x*x*x*x*1;;
let fast_power5 = fun x -> x*x*x*x*x*1;;
```

Compile time versus runtime meta-programming

MetaOCaml (staged interpreter)	Template Haskell (templates)
`.<E>.` (bracket)	[| E |] (quotation)
`.~` (escape)	`$s` (splice)
`.<t>.` (type for staged code)	`Q Exp` (quoted values)
`.!` (run)	*none*

Template Haskell: compile time, no runtime overhead

MetaOCaml: runtime code gen, *some* runtime overhead

Key idea of runtime meta-programming:

> *incremental compilation optimises away condition checks, specialises functions.. speedups possible when dynamic variables become static constant values.*

Imitation is the sincerest form of flattery

– Oscar Wilde

Lightweight Modular Staging (LMS) in Scala

- L = lightweight, just a library
- M = modular, easy to extend
- S = staging

Types distinguish expressions evaluated

”execute now” has type:

```
T
```

”execute later” (delayed) has type:

```
Rep[T]
```

Tiark Rompf and Martin Odersky. “Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs”. In: *Commun. ACM* 55.6 (2012), pp. 121–130.

Programmers want

- Higher order functions
- Partial application
- Polymorphism
- … reusable generic abstractions …

Usually programmers usually have to decide:

either performance or productivity

Multi-Stage Programming (MSP) provides *zero cost abstraction*

Lightweight Modular Staging (LMS) in Scala

Scala:

```
def power(b: Double, p: Int): Double =
  if (p==0) 1.0 else b * power(b, p - 1)
```

```
power(x,5)
=> x * power(x, 4)
=> x * x * power(x, 3)
=> x * x * x * power(x, 2)
=> x * x * x * x * power(x, 1)
=> x * x * x * x * x * power(x, 0)
=> x * x * x * x * x * 1.0
```

Lightweight Modular Staging (LMS) in Scala

Scala LMS:

```
def power(b: Rep[Double], p: Int): Rep[Double] =
  if (p==0) 1.0 else b * power(b, p - 1)

power(x,5)

def apply(x1: Double): Double = {
  val x2 = x1 * x1
  val x3 = x1 * x2
  val x4 = x1 * x3
  valx5 = x1 * x4
  x5
}
```

Alexey Rodriguez Blanter et al. *Concepts of Programming Design: Scala and Lightweight Modular Staging (LMS).* Universiteit Utrecht.

LMS success study: Delite

Compiler framework & runtime for high performance EDSLs

DSLs developed with Delite

- OptiML: Machine Learning and linear algebra
- OptiQL: Collection and query operations
- OptiMesh: Mesh-based PDE solvers
- OptiGraph: Graph analysis

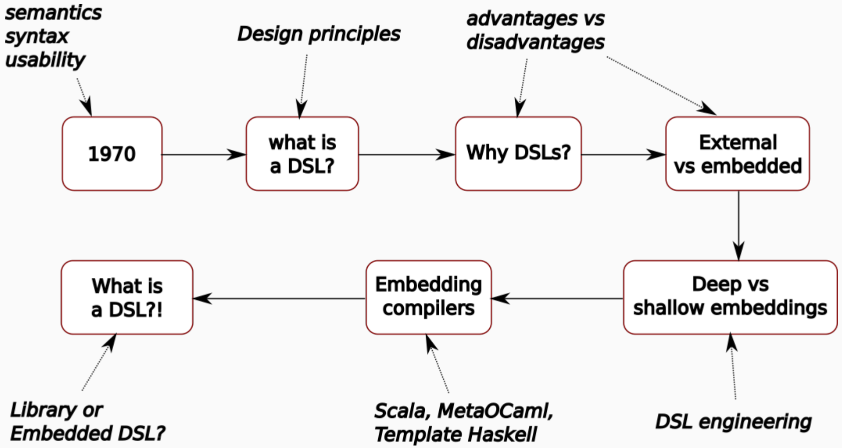
Hassan Chafi et al. “A domain-specific approach to heterogeneous parallelism”. In: *PPOPP*. 2011, pp. 35–46.

Summary of compiler embeddings

Approach	Host frontend	Host backend	Optimise via
Embedded compiler	yes	no	DSL transformation
Staged compiler	no	yes	Delayed expressions
Ext. metaprogramming	yes	yes	Host transformation

- Embedded compilers: Accelerate (Haskell)
- Extensional metaprogramming: Template Haskell
- Staged compilers: MetaOCaml, Scala LMS

Seefried, Chakravarty, and Keller, “Optimising Embedded DSLs Using Template Haskell”.


semantics
syntax
usability
Design principles
advantages vs
disadvantages
1970
what is
a DSL?
Why DSLs?
External
vs embedded
Deep vs
shallow embeddings
Embedding
compilers
What is
a DSL?!
Library or
Embedded DSL?
Scala, MetaOCaml,
Template Haskell
DSL engineering

1. Where’s the boundary between EDSL and host language?
2. When does a library become an EDSL?

Where does EDSL stop and host start?

In February 2016 I asked on Halide-dev about my functions:

```
Image<uint8_t> blurX(Image<uint8_t> image);
Image<uint8_t> blurY(Image<uint8_t> image);
Image<uint8_t> brightenBy(Image<uint8_t> image, float);
```

> *Hi Rob,*
> *You've constructed a library that passes whole images across C++ function call boundaries, so no fusion can happen, and so you're missing out on all the benefits of Halide. This is a long way away from the usage model of Halide. The tutorials give a better sense of ...*

On [Halide-dev]: https://lists.csail.mit.edu/pipermail/halide-dev/2016-February/002188.html

Where does EDSL stop and host start?

Correct solution:

```
Func blurX(Func image);
Func blurY(Func image);
Func brightenBy(Func image, float);
```

Reason: Halide is a *functional language* embedded in C++

But my program compiled and was executed (slowly)

I discovered the error of my ways by:

1. Emailing Halide-dev
2. Reading Halide code examples

Why not a type error?

Are EDSL just libraries?

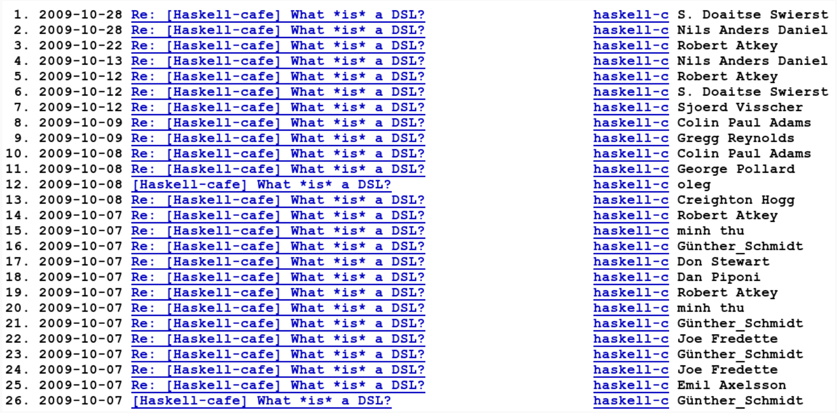
- X is an EDSL for image processing
- Y is an EDSL for web programming
- Z is an EDSL for ....

When is a library *not* domain specific?

Are all libraries EDSLs?

```
Subject:    [Haskell-cafe] What *is* a DSL?
From:       Günther_Schmidt <gue.schmidt () web ! de>
Date:       2009-10-07 15:10:58

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
```

haskell-cafe mailing list



1.	2009-10-28	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	S. Doaitse Swierst
2.	2009-10-28	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Nils Anders Daniel
3.	2009-10-22	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Robert Atkey
4.	2009-10-13	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Nils Anders Daniel
5.	2009-10-12	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Robert Atkey
6.	2009-10-12	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	S. Doaitse Swierst
7.	2009-10-12	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Sjoerd Visscher
8.	2009-10-09	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Colin Paul Adams
9.	2009-10-09	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Gregg Reynolds
10.	2009-10-08	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Colin Paul Adams
11.	2009-10-08	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	George Pollard
12.	2009-10-08	[Haskell-cafe] What *is* a DSL?	haskell-c	oleg
13.	2009-10-08	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Creighton Hogg
14.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Robert Atkey
15.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	minh thu
16.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Günther_Schmidt
17.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Don Stewart
18.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Dan Piponi
19.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Robert Atkey
20.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	minh thu
21.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Günther_Schmidt
22.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Joe Fredette
23.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Günther_Schmidt
24.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Joe Fredette
25.	2009-10-07	Re: [Haskell-cafe] What *is* a DSL?	haskell-c	Emil Axelsson
26.	2009-10-07	[Haskell-cafe] What *is* a DSL?	haskell-c	Günther_Schmidt

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.

– 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 charateristics 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

Library versus EDSL?

When does a library become an EDSL?

1. Well defined formal semantics of library functions?
2. Embedded compiler *e.g.* Accelerate
3. Language restriction library restricts expressivity of host?
4. Extends syntax library extends host's syntax *e.g.* quasi-quotating

Library versus EDSL?

States $R, S, T ::=$	$S \mid T$	parallel composition
$\mid$	$\langle M \rangle_p$	thread on node p, executing M
$\mid$	$\langle\!\langle M \rangle\!\rangle_p$	spark on node p, to execute M
$\mid$	$i\{M\}_p$	full IVar i on node p, holding M
$\mid$	$i\{\langle M \rangle_q\}_p$	empty IVar i on node p, supervising thread $\langle M \rangle_q$
$\mid$	$i\{\langle\!\langle M \rangle\!\rangle_Q\}_p$	empty IVar i on node p, supervising spark $\langle\!\langle M \rangle\!\rangle_q$
$\mid$	$i\{\bot\}_p$	zombie IVar i on node p
$\mid$	dead_p	notification that node p is dead

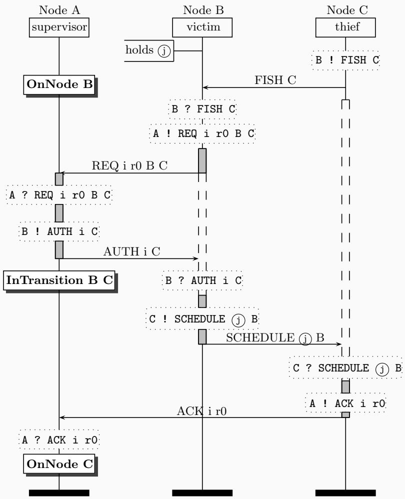
$$\langle \mathcal{E}[\texttt{spawn}\, M] \rangle_p \longrightarrow \nu i.(\langle \mathcal{E}[\texttt{return}\, i] \rangle_p \mid i\{\langle\!\langle M \texttt{ >>= } \texttt{rput}\, i \rangle\!\rangle_{\{p\}}\}_p \mid \langle\!\langle M \texttt{ >>= } \texttt{rput}\, i \rangle\!\rangle_p), \tag{spawn}$$

$$\langle \mathcal{E}[\texttt{spawnAt}\, q\, M] \rangle_p \longrightarrow \nu i.(\langle \mathcal{E}[\texttt{return}\, i] \rangle_p \mid i\{\langle M \texttt{ >>= } \texttt{rput}\, i \rangle_q\}_p \mid \langle M \texttt{ >>= } \texttt{rput}\, i \rangle_q), \tag{spawnAt}$$

$$\langle\!\langle M \rangle\!\rangle_{p_1} \mid i\{\langle\!\langle M \rangle\!\rangle_P\}_q \longrightarrow \langle\!\langle M \rangle\!\rangle_{p_2} \mid i\{\langle\!\langle M \rangle\!\rangle_P\}_q, \text{ if } p_1, p_2 \in P \tag{migrate}$$

$$\langle\!\langle M \rangle\!\rangle_p \mid i\{\langle\!\langle M \rangle\!\rangle_{P_1}\}_q \longrightarrow \langle\!\langle M \rangle\!\rangle_p \mid i\{\langle\!\langle M \rangle\!\rangle_{P_2}\}_q, \text{ if } p \in P_1 \cap P_2 \tag{track}$$

Library versus EDSL?

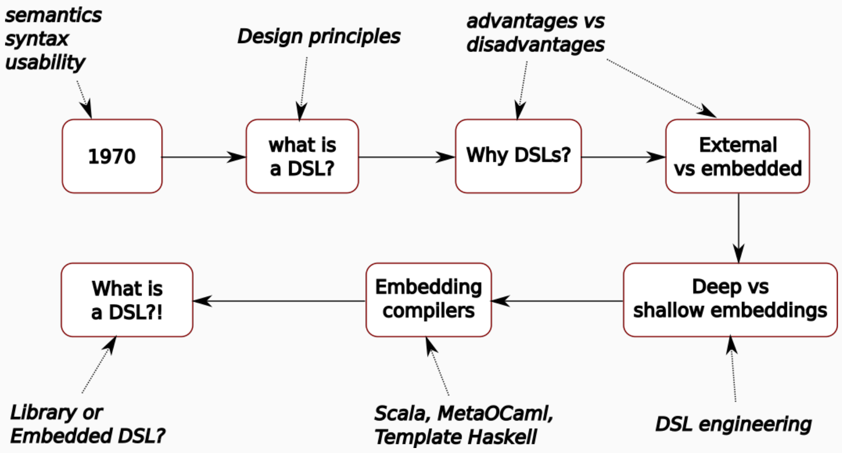


$$\begin{tikzcd}
i\{\langle\!\langle M\rangle\!\rangle_{\{B\}}\}_A \quad \mid \quad \langle\!\langle M\rangle\!\rangle_B \arrow[d, "(track)"] \\
i\{\langle\!\langle M\rangle\!\rangle_{\{B,C\}}\}_A \quad \mid \quad \langle\!\langle M\rangle\!\rangle_B \arrow[d, "(migrate)"] \\
i\{\langle\!\langle M\rangle\!\rangle_{\{B,C\}}\}_A \quad \mid \quad \langle\!\langle M\rangle\!\rangle_C \arrow[d, "(track)"] \\
i\{\langle\!\langle M\rangle\!\rangle_{\{C\}}\}_A \quad \mid \quad \langle\!\langle M\rangle\!\rangle_C
\end{tikzcd}$$

1. Formal operational semantics
2. Verified scheduler with model checking
3. In a monad
4. Uses GHC
    - frontend
    - backend
    - runtime system

DSL or library?

Robert J. Stewart, Patrick Maier, and Phil Trinder. "Transparent fault tolerance for scalable functional computation". In: *J. Funct. Program.* 26 (2016), e5.


semantics
syntax
usability
Design principles
advantages vs
disadvantages
1970
what is
a DSL?
Why DSLs?
External
vs embedded
Deep vs
shallow embeddings
Embedding
compilers
What is
a DSL?!
Library or
Embedded DSL?
Scala, MetaOCaml,
Template Haskell
DSL engineering