Real World (E)DSLs

Scottish Programming Languages and Verification Summer School 2019

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

August 2019

Heriot-Watt University, Edinburgh

What is a DSL

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

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.

DSL Design Guidelines

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

Hudak, “Domain Specific Languages”.

Domain Specificity

Application Domain Examples

- Scheduling
- Simulation
- Lexing/parsing
- Robotics
- Graphics & animation
- Databases
- Logic
- Security
- Modelling
- Graphical user interfaces
- Symbolic computing
- Hardware description
- Text processing
- Computer music
- Distributed & parallel computing

Domain Specificity

DSLs ACM Computing Survey:

- Some consider Cobol a DSL for **business applications**, others argue this is pushing the **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
- Hard to tell if command languages like the Unix shell or scripting languages like Tcl are DSLs
- 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.

Why DSLs?

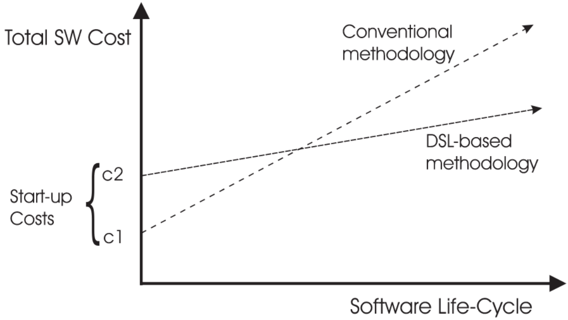
DSL Advantages

1. More **concise**: easy to look at, see, think about, show
2. Increase **programmer productivity**: DSLs tend to be high level meaning shorter programs
3. Programs **easier to maintain**
    - less code == less maintenance
4. Are **easier to reason about**: programs expressed at level of problem domain, domain knowledge can be 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 model to develop software components
- Philips had issues with Rhapsody (see paper)
- Dezyne: another modelling language, verifies live-lock freedom, determinism etc. properties
- Philips developed **ComMA DSL**
    - automates translation of Rhapsody to Dezyne

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

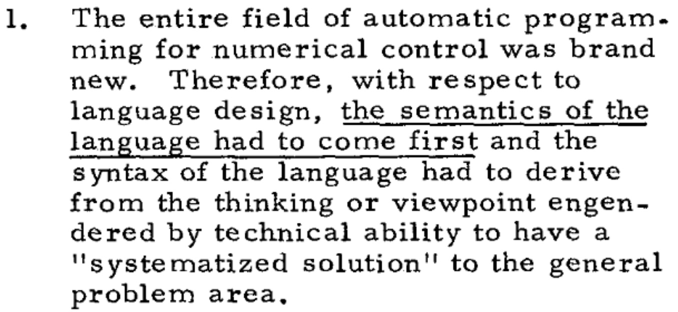
Schuts, Hooman, and Tielemans, "Industrial Experience with the Migration of Legacy Models using a DSL".

Early DSL example

APT

APT (Automatically Programmed Tool):

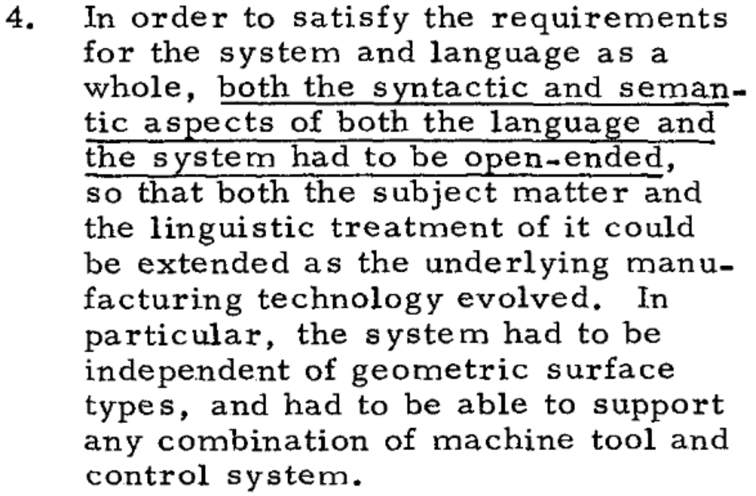
- Numerically controlled machine tools
- One of the 1^{st} DSLs



1. The entire field of automatic programming for numerical control was brand new. Therefore, with respect to language design, the semantics of the language had to come first 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.

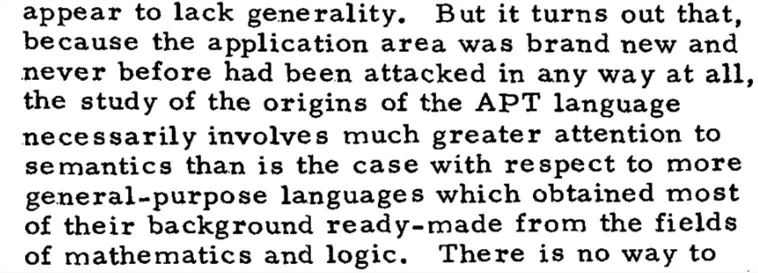
Douglas T. Ross. "Origins of the APT Language for Automatically Programmed Tools". In: *SIGPLAN Not.* 13.8 (Aug. 1978), pp. 61–99.

Future Proofing APT



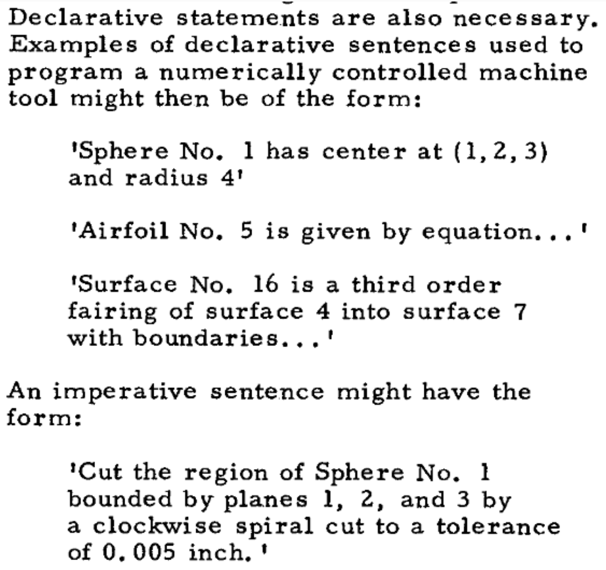
4. In order to satisfy the requirements for the system and language as a whole, both the syntactic and semantic aspects of both the language and the system had to be open-ended, so that both the subject matter and the linguistic treatment of it could be extended as the underlying manufacturing technology 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 Specifity of APT Semantics



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

APT Declarative vs Imperative



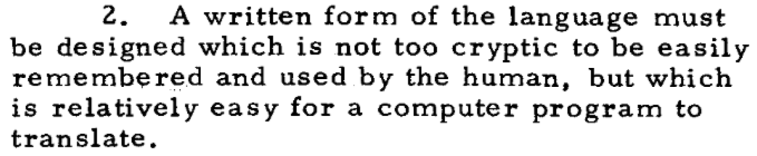
Declarative statements are also necessary. Examples of declarative sentences used to program a numerically controlled machine tool might then be of the form:

> '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:

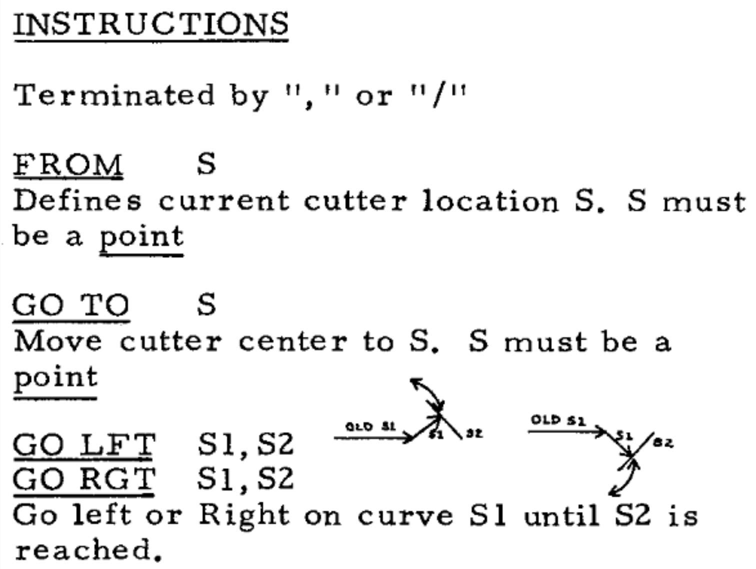
> 'Cut the region of Sphere No. 1 bounded by planes 1, 2, and 3 by a clockwise spiral cut to a tolerance of 0.005 inch.'

APT Implementation Concerns



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



INSTRUCTIONS

Terminated by "," or "/"

FROM S

Defines current cutter location S. S must be a point

GO TO S

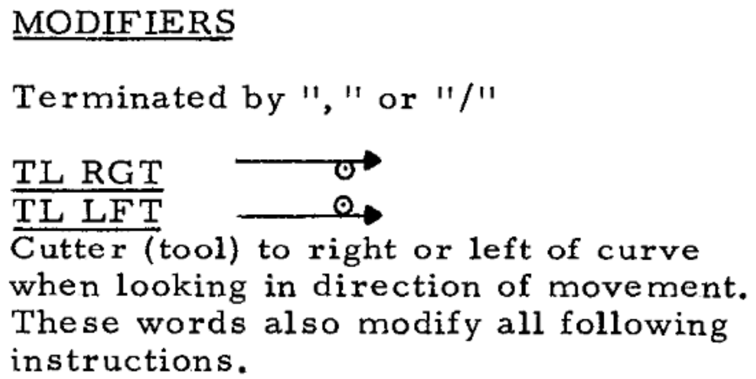
Move cutter center to S. S must be a point

GO LFT S1, S2

GO RGT S1, S2

Go left or Right on curve S1 until S2 is reached.

APT



MODIFIERS

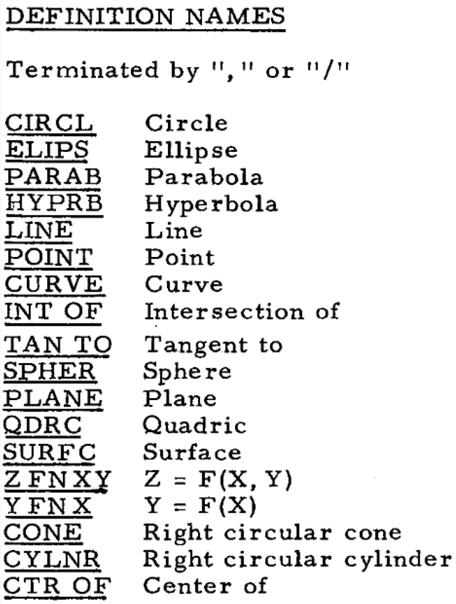
Terminated by "," or "/"

TL RGT

TL LFT

Cutter (tool) to right or left of curve when looking in direction of movement. These words also modify all following instructions.

APT

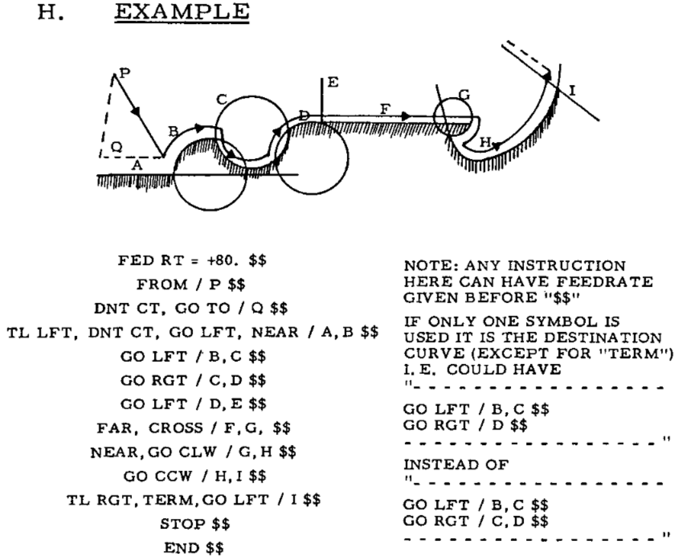


DEFINITION NAMES

Terminated by "," or "/"

CIRCL	Circle
ELIPS	Ellipse
PARAB	Parabola
HYPRB	Hyperbola
LINE	Line
POINT	Point
CURVE	Curve
INT OF	Intersection of
TAN TO	Tangent to
SPHER	Sphere
PLANE	Plane
QDRC	Quadric
SURFC	Surface
ZFNXY	Z = F(X, Y)
YFNX	Y = F(X)
CONE	Right circular cone
CYLNR	Right circular cylinder
CTR OF	Center of

APT



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 "$$"

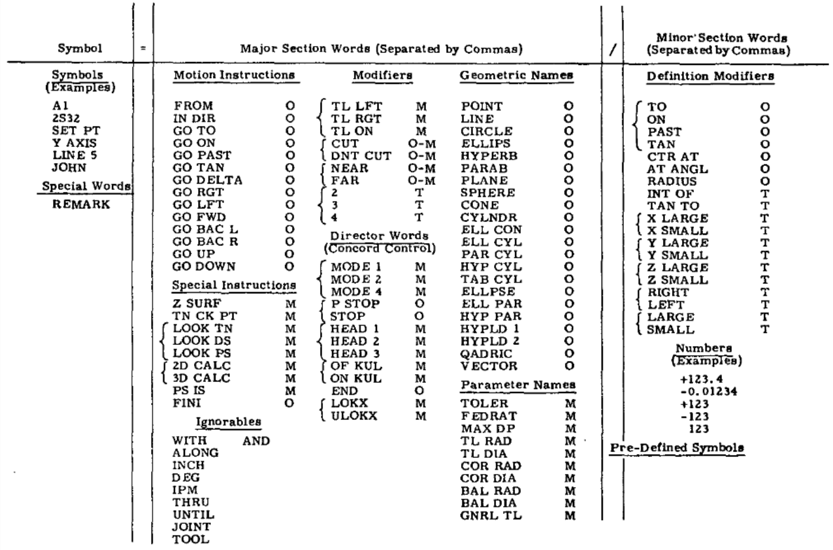
IF ONLY ONE SYMBOL IS USED IT IS THE DESTINATION CURVE (EXCEPT FOR "TERM") I. E. COULD HAVE

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

INSTEAD OF

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

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

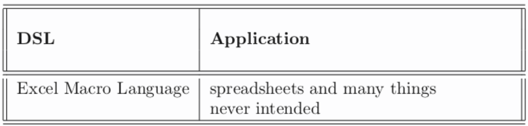
Numbers (Examples)

+123.4
-0.01234
+123
-123
123

Pre-Defined Symbols

DSLs used Today

- PERL: text manipulation
- VHDL: hardware description
- LaTeX: typesetting
- HTML: document markup
- SQL: database transactions
- Maple: symbolic computing
- AutoCAD: computer aided design
- Prolog: logic
- Excel



DSL	**Application**
Excel Macro Language	spreadsheets and many things never intended

---

Hudak, “Domain Specific Languages”.

The rest of this talk

1. Counterexamples for many ”in general” observations
2. Code examples mostly extracted from publications
    - Footnote citations on these slides

Modern DSL examples

Motivations for DSLs: Examples

- **Familiar notation** for domain experts (SQL)
- High level **abstraction** (Keras)
- **Compositionality** (Frenetic)
- **Speed** (Halide)
- **Productivity** (Halide)
- **Correctness** (Ivory)

Domain Expert Familiarity: SQL

```
SELECT firstName, lastName, address
FROM   employee
WHERE  salary > ALL
   (SELECT salary
    FROM employee
    WHERE firstName = 'Paul')
```

- Programmer training
    - 1 day to become SQL competent
    - months to become SQL expert

Hudak, “Domain Specific Languages”.

Abstraction: Keras

Embedded in Python for defining neural networks

```
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu')
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

- High level API on top of Tensorflow
- Rapid prototyping of neural networks
- Insert Tensorflow code to Keras model/training pipeline
    - TF flexibility: custom cost function or layer
    - TF functionality: threads, debugger
    - TF control: set variables to be trainable or not
- Analogous to inline ASM, inline C, etc.

Compositionality: Frenetic Network programming language

- **Problem** with OpenFlow and NOX (SDN languages)
    - lack compositionality
    - low level: programs **unnecessarily complicated**
    - two-tier programs lead to **race conditions**
- **Solution**: Frenetic DSL
    - high level compositional patterns (translates to OpenFlow)
    - two sub-languages
        1. ”see every packet” network *query* language
        2. functional reactive network *policy* language
    - queries and policies **compose**

Nate Foster et al. “Frenetic: a network programming language”. In: *Proceeding of the 16th ACM SIGPLAN ICFP 2011, Tokyo, Japan, September 19-21, 2011.* ACM, 2011, pp. 279–291.

Compositionality: Frenetic

Embedded in Python... *"to ease adoption"*

```
def host_query():
  return (Select(sizes) *
          Where(inport_fp(2)) *
          GroupBy([dstmac]) *
          Every(60))

def all_stats():
  Merge(host_query(),web_query()) >> Print()

def repeater_web_monitor():
  repeater()
  all_stats()
```

Speed: Halide

- High performance C++ embedded image/array processing
- Separates *algorithm* from *scheduling* code

```
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);
  blur_x.compute_at(blur_y, x).vectorize(x, 8);
  return blur_y;
}
```

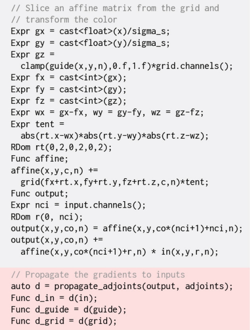
Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *ACM SIGPLAN PLDI, Seattle, WA, USA, June*. 2013, pp. 519–530.

Speed and Productivity: Halide

- Programmer productivity **and** fast performance
- Bilateral slicing layer
  - high-performance image processing architecture to approximate complicated image processing pipelines
- Halide extensions
  - Automatic Differentiation
  - Scheduling
- Programmer productivity
  - Halide 24 lines, PyTorch 42 lines, CUDA 308 lines
- Halide 10x faster than CUDA, 20x faster than PyTorch

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.

Speed and Productivity: Halide



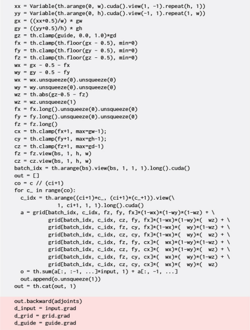
```
// Slice an affine matrix from the grid and
// transform the color
Expr gx = cast<float>(x)/sigma_s;
Expr gy = cast<float>(y)/sigma_s;
Expr gz =
  clamp(guide(x,y,n),0.f,1.f)*grid.channels();
Expr fx = cast<int>(gx);
Expr fy = cast<int>(gy);
Expr fz = cast<int>(gz);
Expr wx = gx-fx, wy = gy-fy, wz = 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)

CUDA Runtime

308 lines 430 ms (1 MPix)

2270 ms (4 MPix)

Li et al., “Differentiable programming for image processing and deep learning in halide”.

Correctness: Ivory

- Ivory: safe systems programming, memory and type safety
- Type system **shallowly embedded** using GHC type features
- Syntax is **deeply embedded**, from one AST:
    - Embedded C generation
    - 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

Correctness: Ivory

```
a <- local (ival 0)
b <- local (ival 1)
```

- `a` and `b` local stack variables

```
n `times` \_ith -> do
  a' <- deref a
  b' <- deref b
  store a b'
  store b (a' + b')
```

- Run a loop 1000 times (inferred from `[Ix 1000]`)

Correctness: Ivory

```
fib_loop :: Def ('[Ix 1000] :-> Uint32)
fib_loop  = proc "fib_loop" $ \ n -> body $ do
  a <- local (ival 0)
  b <- local (ival 1)
  n `times` \_ith -> do
    a' <- deref a
    b' <- deref b
    store a b'
    store b (a' + b')
  result <- deref a
  ret result

fib_module :: Module
fib_module = package "fib" (incl fib_loop)

main = C.compile [ fib_module ]
```

---

https://ivorylang.org/ivory-fib.html

Implementations

Notice distinguishing feature?

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

Embedding of external languages too

e.g. Selda: a type safe SQL EDSL

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.

Internal and External DSLs

DSL Implementation Choices

External

1. **Parser + Interpreter**: interactive read–eval–print loop
2. **Parser + Compiler**: DSL constructs to another language
    - LLVM a popular IR to target for CPUs/GPUs

Internal

- Embed in a general purpose language
- Reuse features/infrastructure of existing language
    - frontend (syntax + type checker)
    - *maybe* its backend too
    - *maybe* its runtime system too
- Concentrate on *semantics*
- Metaprogramming tools to have uniform look and feel

Trend: language *embeddings*, away from external approaches

External Advantages

- Domain specific notation **not constrained by host's syntax**
- Building DSLs from scratch: **better error messages**
- DSL syntax **close to notations** used by domain experts
- Domain specific **analysis, verification, optimisation, parallelisation** and **transformation** (AVOPT) is possible
- **AVOPT for internal?** host's syntax or semantics may be too complex or not well defined, limiting AVOPT

External Disadvantages

- External DSLs is large development effort because a complex language processor must be implemented
    - syntax, semantics, interpreter/compiler, tools
- DSLs from scratch often lead to incoherent designs
- DSL design is hard, requiring both domain and language development expertise. **Few people have both**.
- Mission creep: programmers want more features
- A new language for every domain?

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

Implementation of Internal DSLs

- **Syntax tree manipulation** (deeply embedded compilers)
    - create & traverse AST, AST manipulations to generate code
- **Type embedding** (e.g. `Par` monad, parser combinators)
    - DS types, operations over them
- **Runtime meta-programming** (e.g. MetaOCaml, Scala LMS)
    - Program fragments generated at **runtime**
- **Compile-time meta-programming** (e.g. Template Haskell)
    - Program fragments generated at **compile time**
- **Preprocessor** (e.g. macros)
    - DSL translated to host language before compilation
    - Static analysis limited to that performed by base language
- **Extend a compiler** for domain specific code generation

Internal DSL Advantages/Disadvantages

- Advantages
  - modest development effort, rapid prototyping
  - many language features for free
  - host tooling (debugging, perf benchmarks, editors) for free
  - lower user training costs
- Disadvantages
  - syntax may be far from optimal
  - cannot easily introduce arbitrary syntax
  - difficult to express/implement domain specific optimisations, affecting efficiency
  - cannot easily extend compiler
  - bad error reporting

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

Counterexamples

Claimed disadvantages of EDSLs:

1. Difficult to extend a host language compiler
2. Bad error messages

Are these fair criticisms?

Extending a Compiler

Counterexample to ”extensible compiler” argument:

- user defined GHC rewrites
- GHC makes no attempt to verify rule is an identity
- GHC makes no attempt to ensure that the right hand side is mor efficient than the left hand side
- Opportunity for domain specific optimisations?

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

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

Custom Error Message

EDSL *"bad error reporting"* claim not entirely true.

```
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
```

Library versus EDSL?

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?

DSL design patterns

- Language exploitation
  1. Specialisation: restrict host for safety, optimisation..
  2. Extension: host language syntax/semantics extended
- Informal designs
  - Natural language and illustrative DSL programs
- Formal designs
  - BNF grammars for syntax specifications
  - Rewrite systems
  - Abstract state machines for semantic specification

If library **formally defined** does it constitute ”language” status?

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

Library versus EDSL?

When is a library an EDSL?

1. **Well defined DS semantics** library has a formal semantics e.g. HdpH-RS has a formal operational semantics for its constructs?
2. **Compiler** library has its own compiler for its constructs *E.g.* Accelerate?
3. **Language restriction** library is a restriction of expressivity e.g. *lifting* values into the library's types?
4. **Extends syntax** library extends host's syntax *e.g.* use of compile time meta-programming?

Library versus EDSL?

HdpH-RS embedded in Haskell

```
-- task distribution
data Par a  -- monadic parallel computation of type 'a'
type Task a
spawn   ::         Task a -> Par (Future a)  -- lazy
spawnAt :: Node -> Task a -> Par (Future a)  -- eager

-- communication of results via futures
type Future a
get :: Future a -> Par a  -- local read
```

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

Library versus EDSL?

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

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

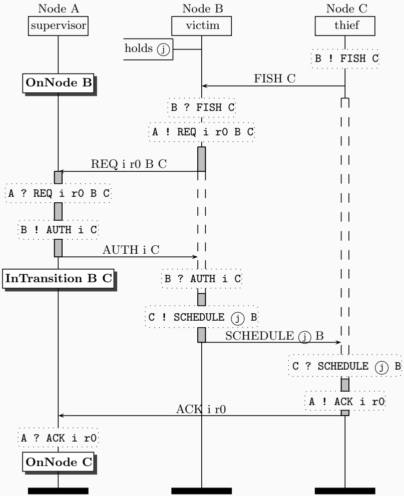
$$\langle \mathcal{E}[\texttt{spawnAt}\,q\,M]\rangle_p \longrightarrow \nu i.(\langle \mathcal{E}[\texttt{return}\,i]\rangle_p \mid i\{\langle M\ \texttt{>>=}\ \texttt{rput}\,i \rangle_q\}_p \mid \langle M\ \texttt{>>=}\ \texttt{rput}\,i \rangle_q), \tag{spawnAt}$$

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

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

etc...

Library versus EDSL?



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

Library versus EDSL?

HdpH-RS domain: scalable fault tolerant parallel computing

1. 3 primitives, 3 types
2. An operational semantics for these primitives
    - domain: task parallelism + fault tolerance
3. A verified scheduler

It is a shallow embedding:

- primitives implemented in Haskell that return *values*
- uses GHCs frontend, backend and its RTS

Is HdpH-RS ”just” library, or a DSL?

Library versus EDSL?

Accelerate DSL for parallel array processing

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

Has multiple backends from Accelerate AST

- LLVM IR
- CUDA

Language Embeddings

Shallow Embeddings: Par monad

- Abstract data types for the domain
- Operators over those types
- In Haskell a monad might be *the* central construct

```
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 embeddings simple to implement
  - no compiler construction
- Host compiler has no domain knowledge
  - applies host language’s backend to generate machine code

Shallow Embeddings: Repa

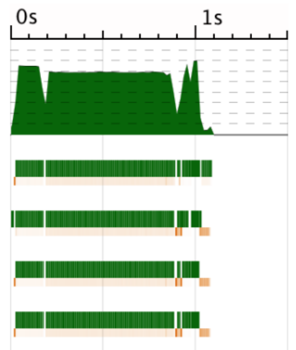
```
data family Array rep sh e
data instance Array D sh e = ADelayed sh (sh -> e)
data instance Array U sh e = AUnboxed sh (Vector e)

-- 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
```

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.

Shallow Embeddings: Repa



- function composition on delayed arrays
- fusion e.g. *map*/*map*, permutation, replication, slicing, etc.
- relies on GHC for code generation
- makes careful use of GHCs primops (more next lecture)
- at mercy of GHC code gen capabilities

Language and Compiler Embeddings

Overview

Let’s look at three approaches:

1. Deeply embedded compilers .*e.g.* Accelerate
2. Compile time metaprogramming *e.g.* Template Haskell
3. Compiler staging *e.g.* MetaOCaml, Scala

Deeply Embedded Compilers

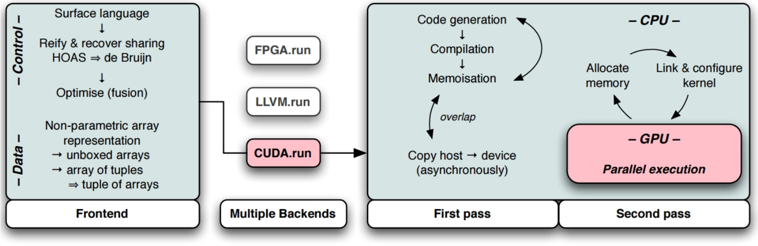
Deep Embeddings

- Deep EDSLs don’t use all host language
    - may have its own compiler
    - or runtime system
- constructs return AST structures, not values

Deep EDSL: Accelerate

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

dotProductGPU xs ys = LLVM.run (dotp xs ys)
```



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.

Deep EDSL: Accelerate

My function:

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

The *structure* returned:

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

Deep EDSL: Compiling and Executing Accelerate

```
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
```

Compile Time Metaprogramming

Compile time metaprogramming

- Main **disadvantage of embedded compilers**
    - cannot access to host language's optimisations
    - cannot use language constructs requiring host language types e.g. *if/then/else*
- **Shallow embeddings** don't suffer these problems
    - but **inefficient execution performance**
    - **no domain specific optimisations**
- **Compile time metaprogramming** transforms user written code to syntactic structures
    - host language -> AST transforms -> host language
    - all happens at **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 is: $M \times M^{-1} = I$

Host language does not know this property for matrices.

Consider the computation: *m * inverse m * n*

- Metaprogramming algorithm:
    1. *reify* code into an AST data structure

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

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

Compile time metaprogramming with Template Haskell

Apply the optimisation:

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

Pattern match with $\lambda p.e$

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

Pattern match with $f\ a$

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

And the rest

```haskell
rmMatByInverse exp = exp
```

Compile time metaprogramming with Template Haskell

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**.

Comparison with Deeply Embedded Compiler Approach

Our computation:

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

Optimised at runtime:

```
rmMatByInverse :: Exp -> Exp
rmMatByInverse exp@(Multiply (Var x) (Inverse (Var y))) =
  if x == y then Identity else exp
rmMatByInverse (Lambda pats exp) =
  Lambda (pats) (rmMatByInverse exp)
rmMatByInverse (App exp exp') =
  App (rmMatByInverse exp) (rmMatByInverse exp')
rmMatByInverse exp = exp

optimise :: AST -> AST
optimise = .. rmMatByInverse ..
```

Deep Compilers vs Metaprogramming

- Pan: **Deeply embedded compiler** for image processing
  - *"Compiling embedded languages"*
- PanTHeon: **Compile time metaprogramming**
  - *"Optimising Embedded DSLs Using Template Haskell"*
- **Performance**: both sometimes faster/slower
  - Pan aggressively unrolls expressions, PanTHeon doesn't
- PanTHeon: **cannot profile spliced code** (TemplateHaskell)
- Source lines of code implementation
  - Pan: ~13k
  - PanTHeon: ~4k (code generator + optimisations for free)

Conal Elliott, Sigbjørn Finne, and Oege de Moor. "Compiling embedded languages". In: *J. Funct. Program.* 13.3 (2003), pp. 455–481.

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

Staged Compilation

Staging

Staged program = **conventional** program + **staging annotations**

- Programmer delays evaluation of program expressions
- A stage is code generator that constructs next stage
- Generator and generated code are expressed in single program
- Partial evaluation
    - performs aggressive constant propagation
    - produces intermediate program specialised to static inputs
- Partial evaluation is a form of program specialization.

Multi Stage Programming (MSP) with MetaOCaml

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

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

1. Escape (`.~`) splices in delayed values

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

1. 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));;

let power2  = fun x -> power (2,x);;
(* power2 3 *)
(* => power (2,3) *)
(* => 3 * power (1,3) *)
(* => 3 * (3 * power (0,3) *)
(* => 3 * (3 * 1) *)
(* => 6 *)

let my_fast_power2 = fun x -> x*x*1;;
```

MetaOCaml Example: Specialising Code

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

- this returns *code of type integer*, not *integer*
- bracket around multiplication returns *code of type integer*
- escape of `power` splices in more code

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

behaves just like:

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

We can keep specialising `power`

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

MetaOCaml Example: Staged Interpreter

A DSL for quantified boolean logic (QBF)

```
type bexp = True
          | False
          | And of bexp * bexp
          | Or of bexp * bexp
          | Not of bexp
          | Implies of bexp * bexp
          (* forall x. x and not x*)
          | Forall of string * bexp
          | Var of string
```

$\forall p.T \Rightarrow p$

```
Forall ("p", Implies(True, Var "p"))
```

Krzysztof Czarnecki et al. "DSL Implementation in MetaOCaml, Template Haskell, and C++". In: *Domain-Specific Program Generation, Dagstuhl Castle, Germany, March, 2003, Revised Papers*. Springer, 2003, pp. 51–72.

MetaOCaml Example: Staged Interpreter

```
let rec eval b env =
  match b with
    True -> true
  | False -> false
  | And (b1,b2) -> (eval b1 env) && (eval b2 env)
  | Or  (b1,b2) -> (eval b1 env) || (eval b2 env)
  | Not b1 -> not (eval b1 env)
  | Implies (b1,b2) -> eval (Or(b2,And(Not(b2),Not(b1)))) env
  | Forall (x,b1) ->
      let trywith bv = (eval b1 (ext env x bv))
      in (trywith true) && (trywith false)
  | Var x -> env x

eval (parse "forall x. x and not x");;
```

- Staging separates 2 phases of computation
  1. **traversing** a program
  2. **evaluating** a program

MetaOCaml Example: Staged Interpreter

```
let rec eval' b env =
  match b with
    True -> .<true>.
  | False -> .<false>.
  | And (b1,b2) -> .< .~(eval' b1 env) && .~(eval' b2 env) >.
  | Or  (b1,b2) -> .< .~(eval' b1 env) || .~(eval' b2 env) >.
  | Not b1 -> .< not .~(eval' b1 env) >.
  | Implies (b1,b2) -> .< .~(eval' (Or(b2,And(Not(b2),Not(b1)))) env)
  | Forall (x,b1) ->
     .< let trywith bv = .~(eval' b1 (ext env x .<bv>.))
        in (trywith true) && (trywith false) >.
  | Var x -> env x

# let a = eval' (Forall ("p", Implies(True, Var "p"))) env0;;
a : bool code =
  .<let trywith = fun bv -> (bv || ((not bv) && (not true)))
    in ((trywith true) && (trywith false))>.

# .! a;;
- : bool = false
```

Metaprogramming: MetaOCaml versus Template Haskell

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 allows inspection of quoted values can alter code's semantics before reaches compiler
- Template Haskell: **compile time** code gen, no runtime overhead
- MetaOCaml: **runtime** code gen, some runtime overhead
  - speedups possible when dynamic variables become static values, incremental compiler optimises away condition checks, specialises functions, etc.

Lightweight Modular Staging (LMS) in Scala

- Programming abstractions used during code *generation*, not reflected in *generated* code
- **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]
```

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)
```

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
}
```

Lightweight Modular Staging (LMS) in Scala

```
def power(b: Rep[Double], p: Int): Rep[Double] = {
  def loop(x: Rep[Double], ac: Rep[Double], y: Int): Rep[Double] = {
    if(y ==0) ac
    else if (y%2==0) loop(x * x, ac, y /2)
    else loop(x, ac * x, y -1)
  }
  loop(b,1.0, p)
}

power(x,11)

def apply(x1: Double): Double = {
  val x2 = x1 * x1  // x * x
  val x3 = x1 * x2  // ac * x
  val x4 = x2 * x2  // x * x
  val x8 = x4 * x4  // x * x
  val x11 = x3 * x8 // ac * x
  x11
}
```

LMS in Practise: Delite

- Delite: compiler framework and runtime for parallel EDSLs
- Scala success story: Delite uses LMS for high performance
- Successful DSLs developed with Delite
    - OptiML: Machine Learning and linear algebra
    - OptiQL: Collection and query operations
    - OptiMesh: Mesh-based PDE solvers
    - OptiGraph: Graph analysis

Summary

Approach	Host frontend	Host backend	Optimise via
Embedded compiler	yes	no	traditional compiler opts
Staged compiler	no	yes	MP: delayed expressions
Ext. metaprogramming	yes	yes	MP: transformation

MP: metaprogramming

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

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

Leaking Abstractions

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**?

Conclusions

Conclusions

- DSL: **notation that captures domain semantics**
- **Why DSLs?**
    - AVOPT: Analysis, Verification (ComMA), Optimisation, Parallelisation (Hdph-RS, Accelerate) and Transformation
    - Compositionality (Frenetic), performance and productivity (Halide), correctness (Ivory)
- **Drawbacks**
    - engineering effort, incoherent designs
    - poor implementation choice from plethora of options
    - unenforced boundaries between EDSL and host language
- **Implementation choices**
    - Internal or external
    - Shallow embed language (Repa), deeply embed compiler (Accelerate), compile time metaprogramming (Template Haskell), staged metaprogramming (MetaOCaml, Scala LMS)