AnyDSL:
A PE Framework for Programming High-Performance Libraries

Sebastian Hack
compilers.cs.uni-saarland.de

joint work with:

Roland Leißa, Klaas Boesche, Arsène Pérard-Gayot (U Saarland),

Richard Membarth, Philipp Slusallek (DFKI),

André Müller, Bertil Schmidt (U Mainz)

UNIVERSITÄT
DES
SAARLANDES

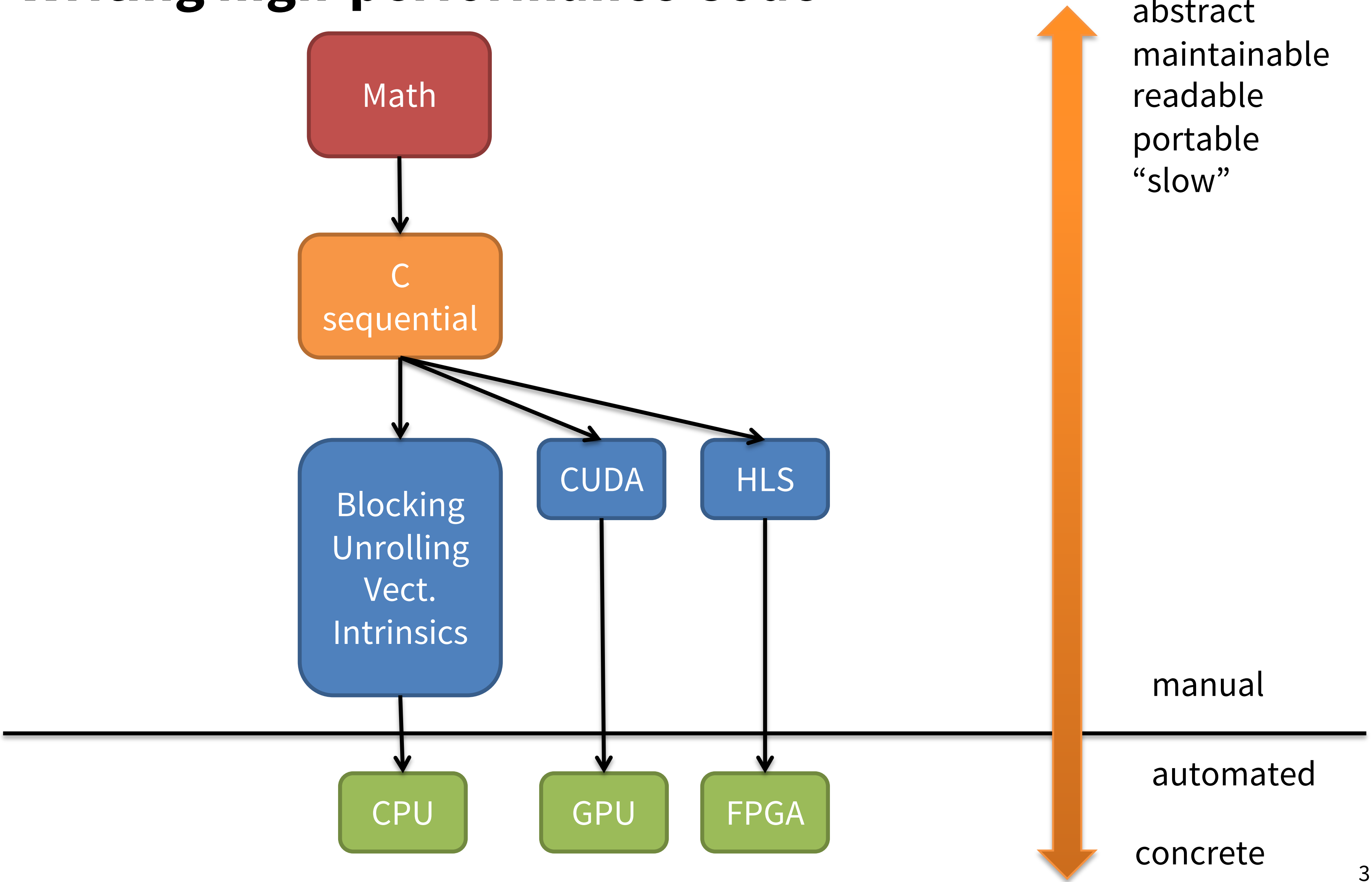
SIC Saarland Informatics Campus

Modern Hardware is hard to program …

… If you are interested in performance

- Multi-Grained Parallelism
  - instruction-level
  - data-parallel (vector units)
  - thread-level (multi-core)
- Multi-Level Mem Hierarchy
  - Registers (no latency)
  - L1-L3 cache
  - contiguity, prefetching
- Heterogeneous
  - Shared/Non-Shared Memory
  - Cache coherence?
  - network on chip
  - GPUs
  - FPGAs
  - Fixed function units

Writing high-performance Code



Math

C
sequential

Blocking
Unrolling
Vect.
Intrinsics

CUDA

HLS

CPU

GPU

FPGA

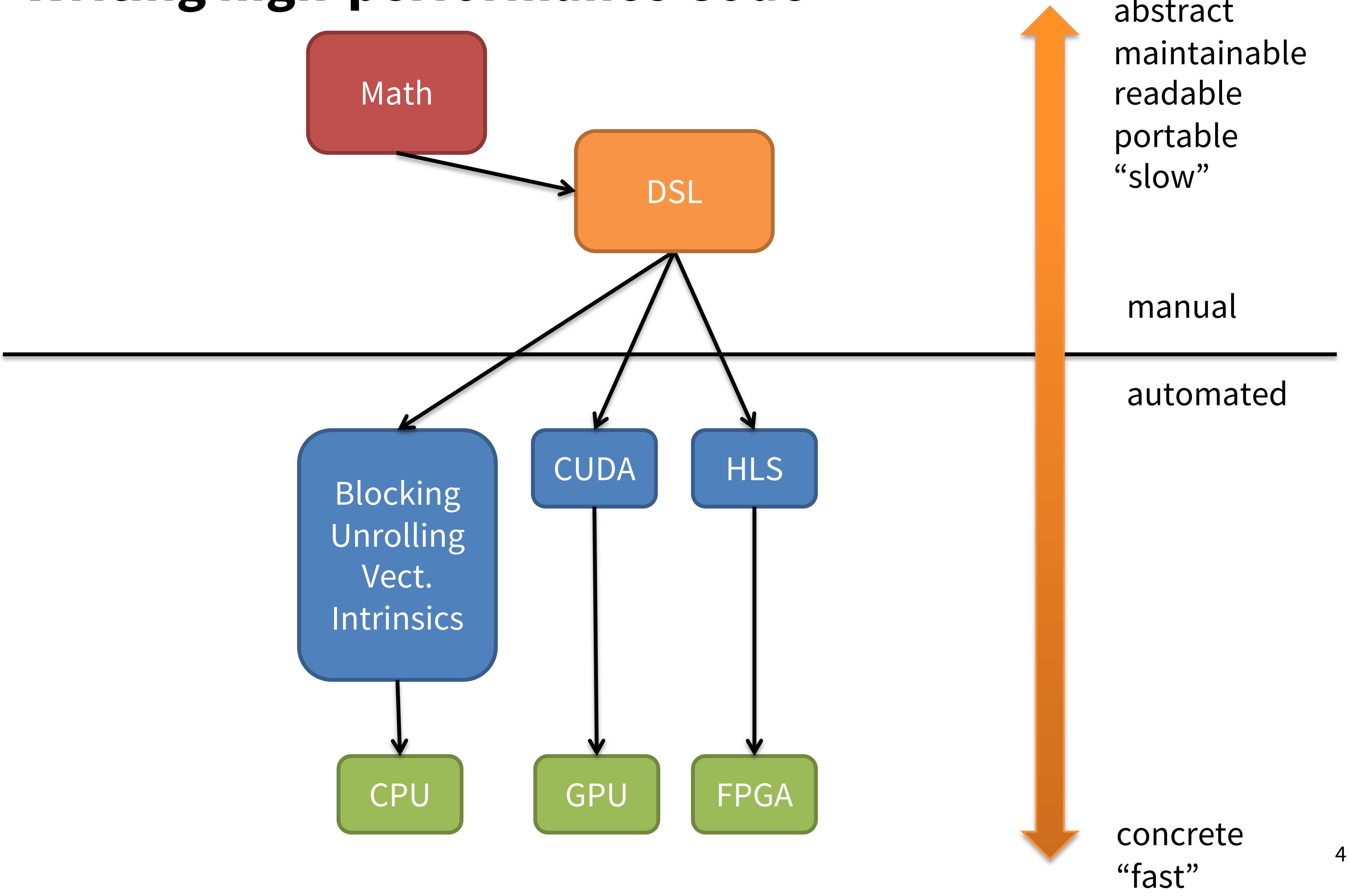
abstract
maintainable
readable
portable
“slow”

manual

automated

concrete
“fast”

Writing high-performance Code



Math

DSL

Blocking
Unrolling
Vect.
Intrinsics

CUDA

HLS

CPU

GPU

FPGA

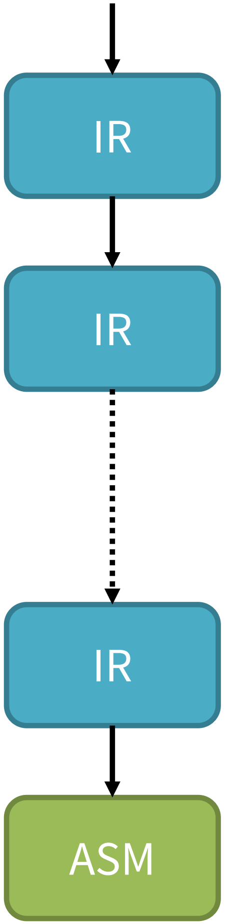
abstract
maintainable
readable
portable
“slow”

manual

automated

concrete
“fast”

Anatomy of a (embedded DSL) Compiler



Middle end:
Transformations /
Code „Optimization“

Back end:
Machine-specific
code generation

- Embed DSL into existing language: deep or shallow
- Provide abstractions to express domain specific constructs
- Middle end:
  - **Lower** DSL abstractions to lower-level constructs
  - Pattern match constructs and **rewrite** them with others
  - Classical (scalar) opts
- Backend translates into HW language

Deep Embedding - HALIDE

```
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;
}
```

- C++ program constructs AST of HALIDE program
- Use C++ overloading to “conceal” that you write a meta program
- Pro: compiler (lib in C++) can manipulate/optimize HALIDE prg
- Con: Hard to understand because you look at the generator, not polyvariant, languages typically don’t blend very well

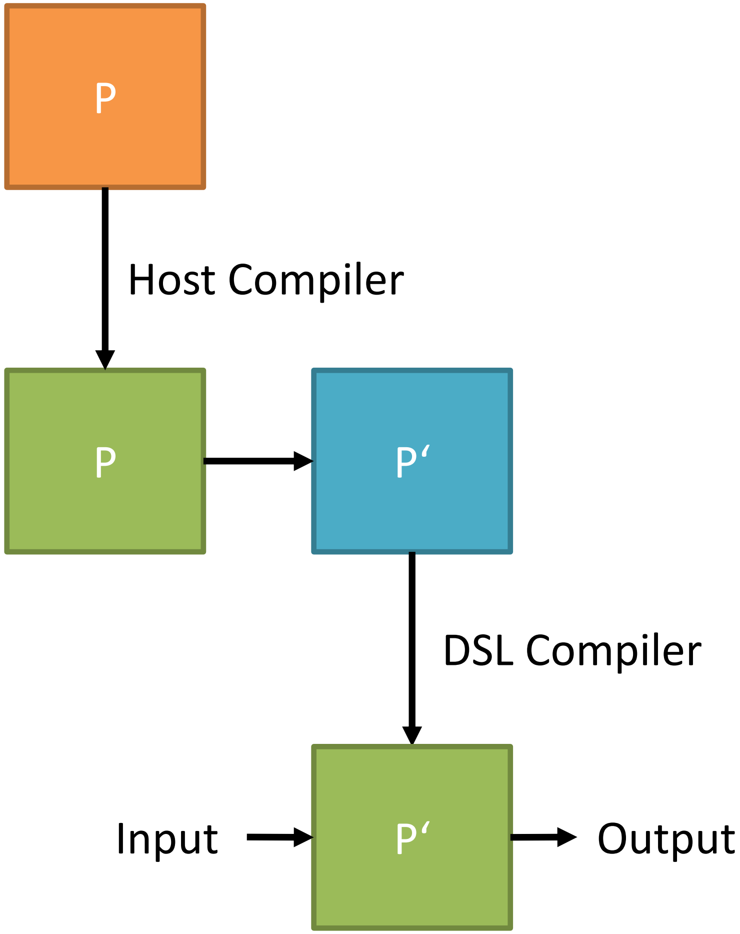
Shallow Embedding – HiPACC

```
class JacobiKernel : public Kernel<float> {
private:
  Accessor<float> &RHS, &Sol;
  Mask<float> &cMask;
public:
  JacobiKernel(IterationSpace<float> &IS, Accessor<float> &RHS,
               Accessor<float> &Sol, Mask<float> &cMask)
    : Kernel(IS), RHS(RHS), Sol(Sol), cMask(cMask) {
    addAccessor(&RHS);
    addAccessor(&Sol);
  }

  void kernel() {
    output() = Sol() + 0.25f*RHS()
             + convolve(cMask, SUM,
             [&] () -> float {return cMask() * Sol(cMask); });
  }
};
```

- DSL is actually C++
- Runs with unmodified C++ compiler (but not so fast)
- Pro: Easy to understand: there is only one program, polyvariant
- Con: Need to modify C++ compiler to get DSL compiler

Deep and Shallow Embeddings

Deep



P

Host Compiler

P

P‘

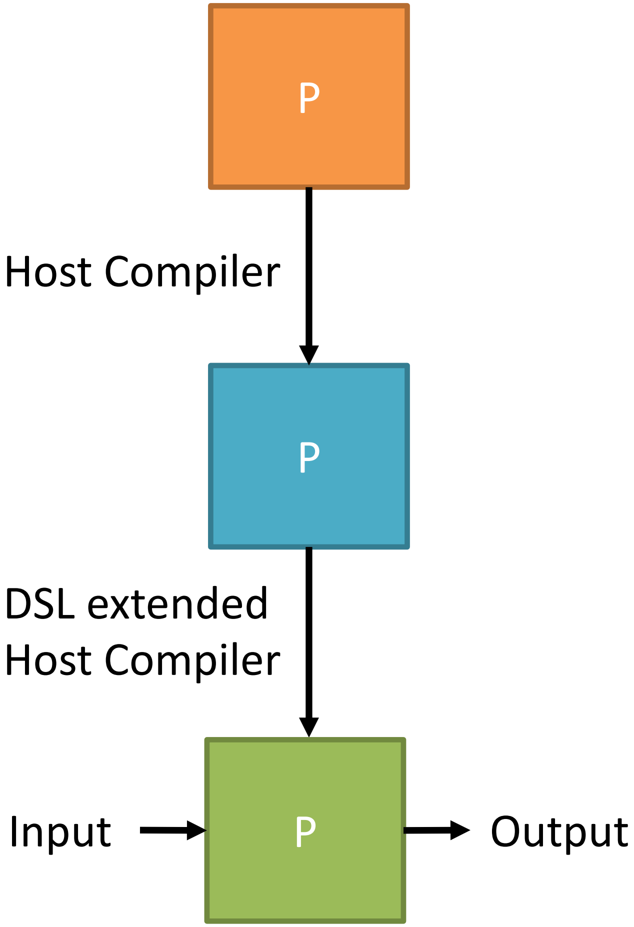
DSL Compiler

Input

P‘

Output

Shallow



P

Host Compiler

P

DSL extended
Host Compiler

Input

P

Output

My Message for Today

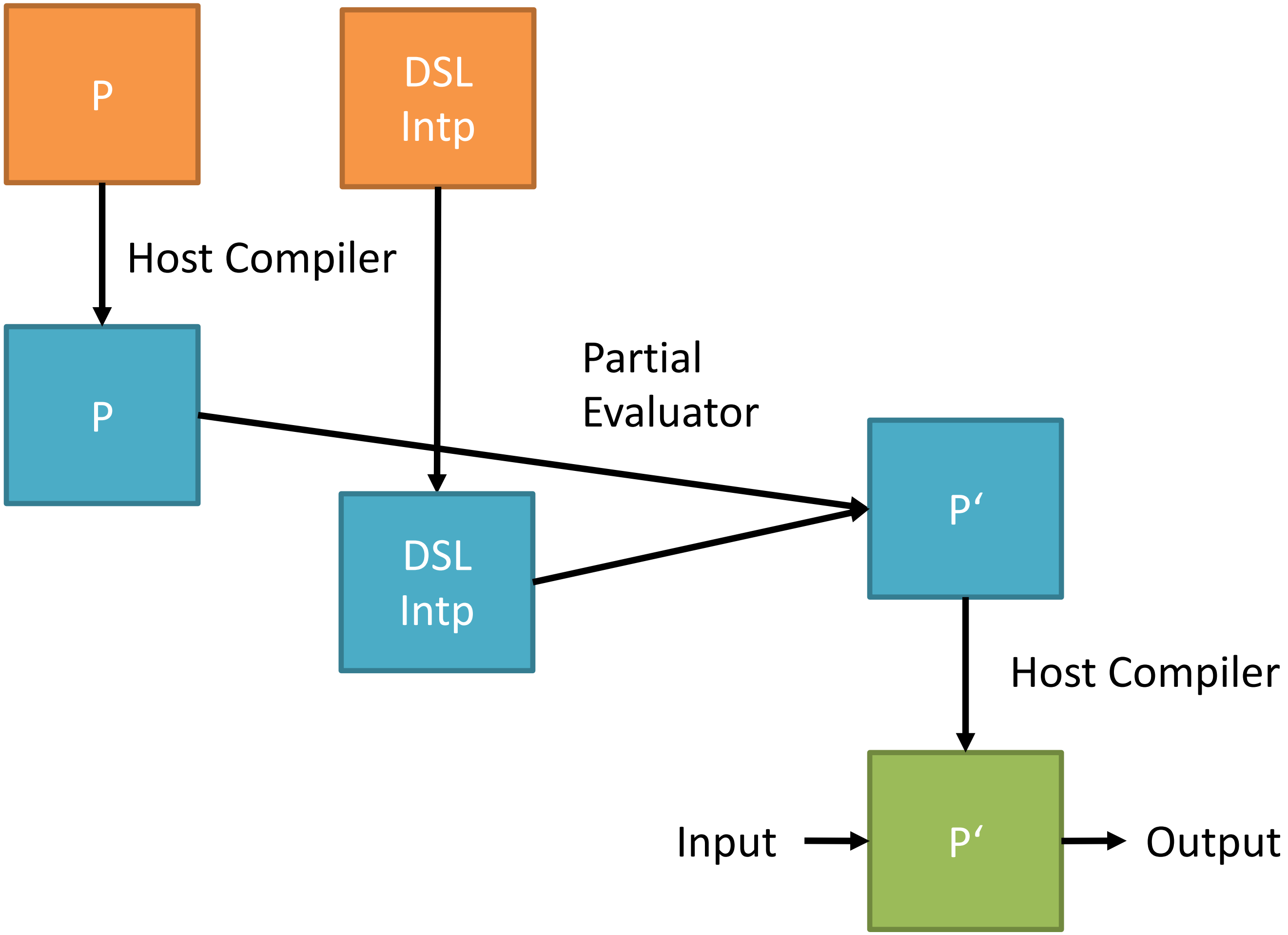
- Writing an embedded DSL compiler is a lot of work

- You can get
  - the same **performance** and **portability**
  - **without** a single line of **compiler writing**

  by
  - writing a library in a **functional language**
  - whose compiler has a **partial evaluator**

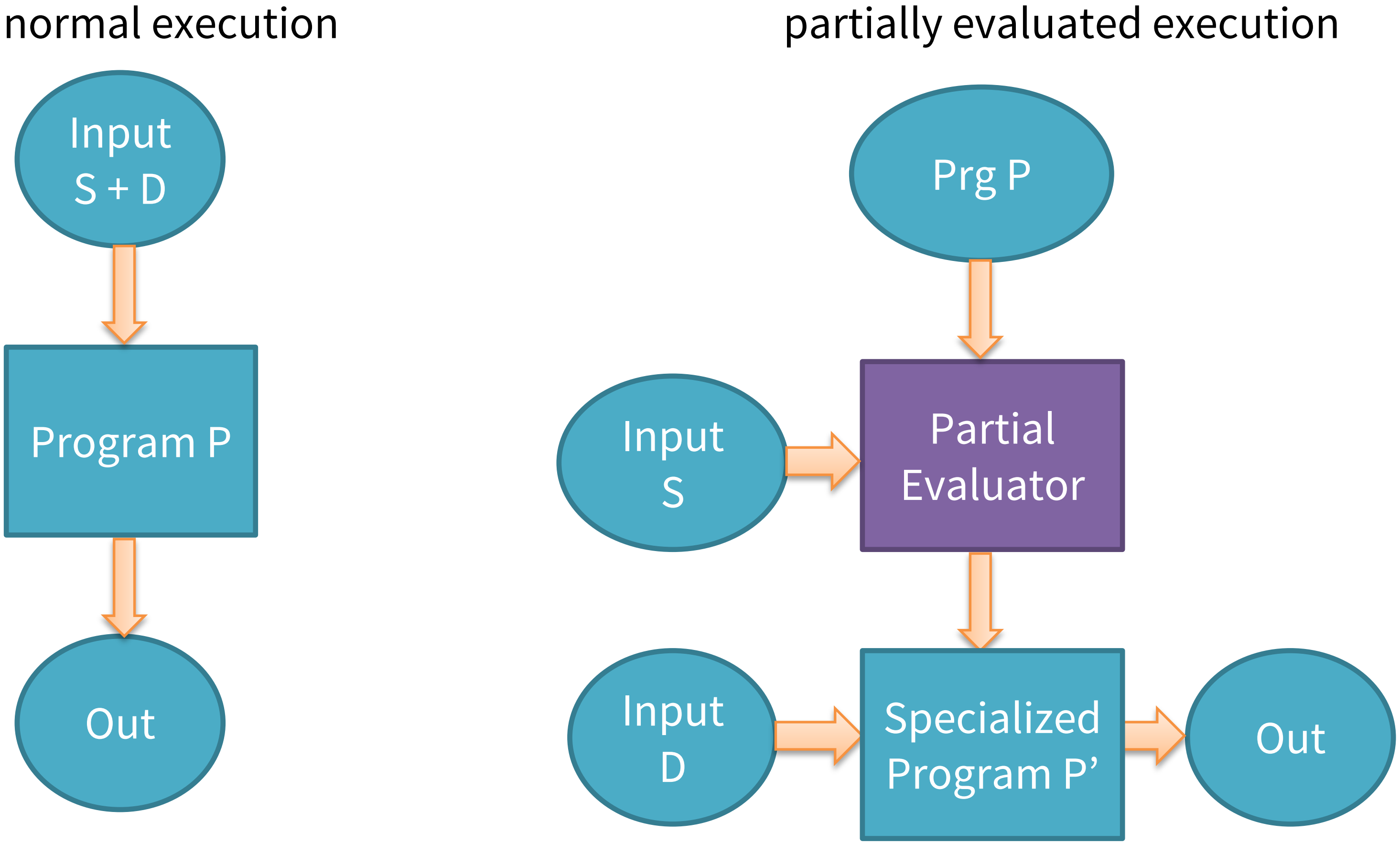
- All that is very well known. But there are hardly any tools

Shallow DSL Embedding using PE

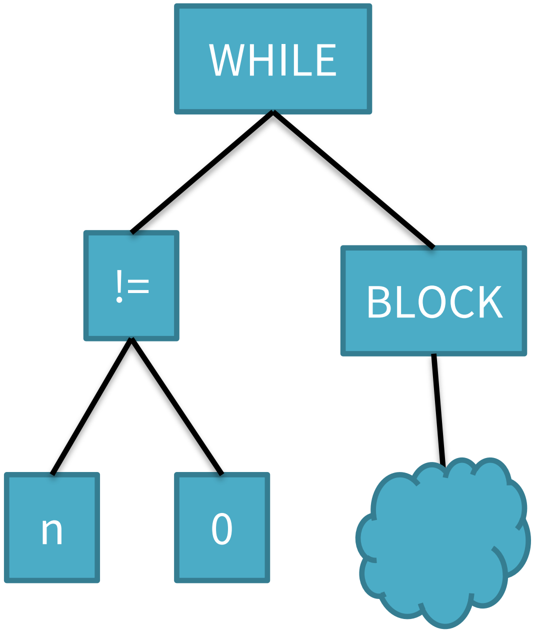


Partial Evaluation in a Nutshell

- Programmer partitions input space
  (S = static, D = dynamic; names misleading → historic)
- Here: P = interpreter, S = DSL program, D = input of DSL program



Implementing Compilers with PE



```
void interpret_stmt(Interpreter* i, AST* n)
{
  switch (n->kind) {
  case WHILE:
    for (;;) {
      VALUE v = interpret_expr(i, n->cond);
      if (! is_true(v))
        break;
      interpret_stmt(I, n->body);
    }
…
```

- what's the simplest implementation of an embedded language?
- An interpreter!

- How to get performance?
- Partially evaluate ("Inline") the program into the interpreter

- First Futamura Projection (1971)

Tagless Interpreters

with tags

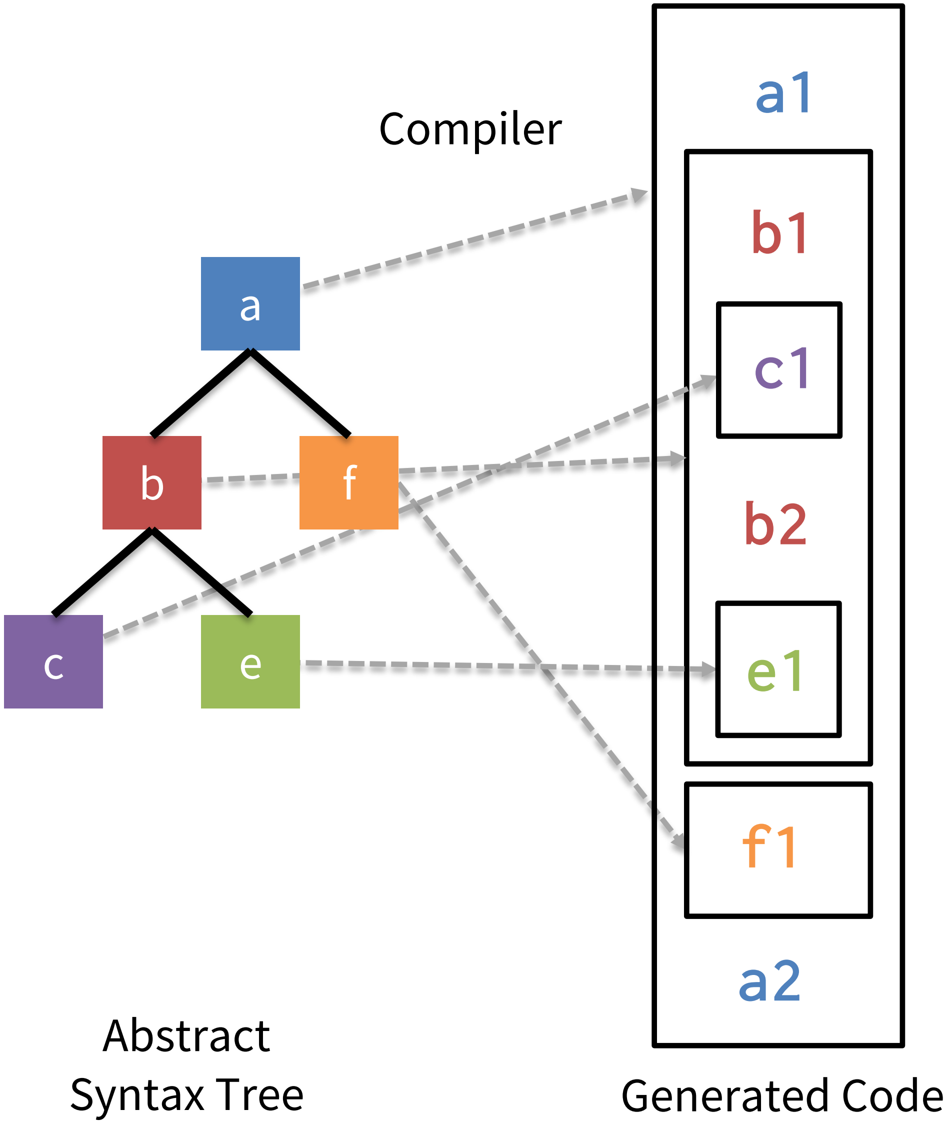
```
void interpret_stmt(Interpreter* i, AST* n)
{
  switch (n->kind) {
  case WHILE:
    for (;;) {
      VALUE v = interpret_expr(i, n->cond);
      if (! is_true(v))
        break;
      interpret_stmt(I, n->body);
    }
```

tagless

```
fn while(cond: () -> int, body: () -> ()) {
  if (cond()) {
    body();
    while(cond, body);
  }
}
```

- Use closures as AST nodes → interpreter becomes library
- semantics of domain specific constructs can be implemented as program in host language (→ shallow embedding)
- Partial evaluation of tagless interpreter removes overhead (entirely)!

Lowering / Codegen is PE

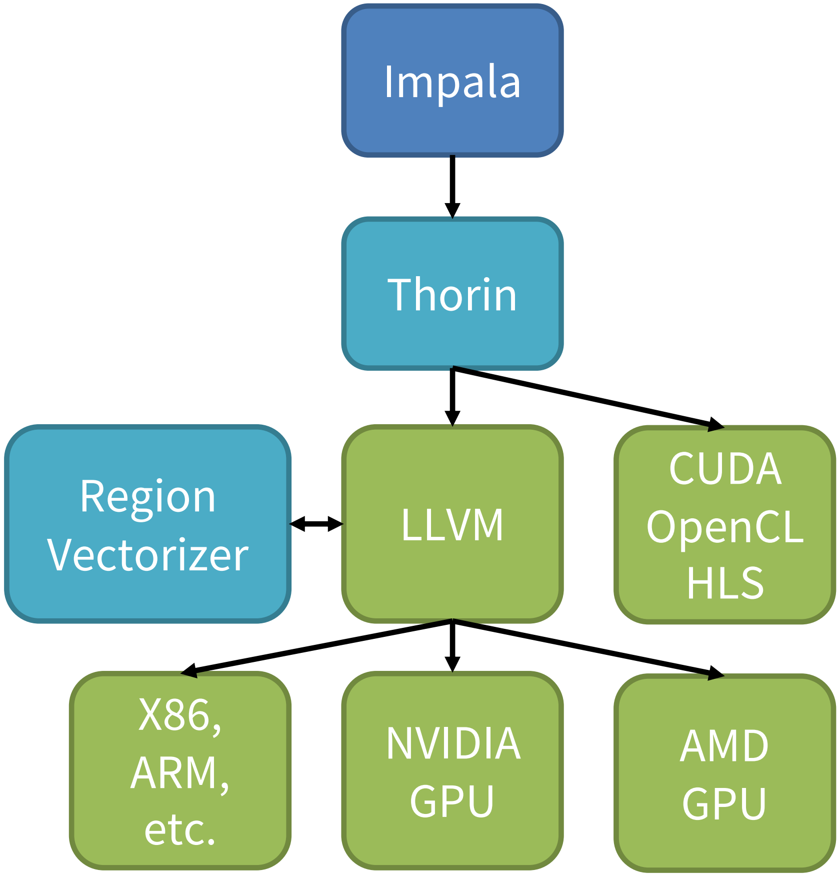


```
fn a(b : fn() -> (),
     f : fn() -> ()) {

  // a1
  b();
  f();
  // a2
}

fn b(c : fn() -> (),
     e : fn() -> ()) {

  // b1
  c();
  // b2
  e();
}

a(
  || { b(c, e) },
  f)
```

PE’ing tagless intp

AnyDSL Framework

- **Impala** functional, imperative language
- **Thorin** IR performs partial evaluation
- Backends for CPU, GPU, FPGA
- Expose HW specific code gen techniques in language



Example: Image Processing in Impala

```
let blur_x = |x, y| (img.get(x-1, y)
                   + img.get(x, y)
                   + img.get(x+1, y)) / 3;
let blur_y = |x, y| (blur_x(x, y-1)
                   + blur_x(x, y)
                   + blur_x(x, y+1)) / 3;

let seq = combine_xy(range, range);
let opt = tile(512, 32, vec(8), par(16));
let gpu = tile_cuda(32, 4);

compute(out_img_seq, seq, blur_y);
compute(out_img_opt, opt, blur_y);
compute(out_img_gpu, gpu, blur_y);
```

AnyDSL vs. Halide: Blur CPU +12%, GPU +7%; Harris Corner: CPU +37%, GPU +44%

Example: Image Processing in Impala

```
fn compute(out: Img, loop: Loop2D, op: BinOp) -> BinOp {
  for x, y in loop(0, 0, img.width, img.height) {
    out.set(x, y, op(x, y))
  }
  |x, y| out.get(x, y)
}

fn combine_xy(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
  |xbeg, ybeg, xend, yend, body|
    loop_y(ybeg, yend, |y| loop_x(xbeg, xend, |x| body(x, y)))
}
```

Example: Image Processing in Impala

```
fn tile(xs: i32, ys: i32, loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
  |xa, ya, xb, yb, f|
    loop_y(0, (yb-ya)/ys, |ly|
      range(ly*ys+ya, (ly+1)*ys+ya, |ry|
        range(0, (xb-xa)/xs, |rx|
          loop_x(rx*xs+xa, (rx+1)*xs+xa, |lx| f(lx, ry)))))
}

fn tile_cuda(xs: i32, ys: i32) -> Loop2D {
  |xa, ya, xb, yb, f| {
    let (grid, block)  = ((xb - xa, yb - ya, 1), (xs, ys, 1));
    cuda(grid, block, || f(cuda_gid_x(), cuda_gid_y()))
  }
}
```

Other DSLs

- Sequence Alignment (U Mainz):
  - within ~15% of hand-tuned expert code: NVBIO, SeqAn
  - ~6 man months development
  - GPU, CPU code from same code base

- Ray Tracing (DFKI)
  - on par with NVIDIA OptiX, Intel Embree
  - ~6 man months development
  - GPU, CPU code from same code base

Summary

- Provide implementation of DSL constructs as (higher-order) library functions (tagless interpreter)
- Partial evaluation produces a prg that “looks” as if it had been created by a DSL compiler
- Three case studies from different domains:
  all implemented as libs, no compiler work needed, perf on par
- **PE enables you to implement large parts of a DSL compiler**
  without having to write that compiler