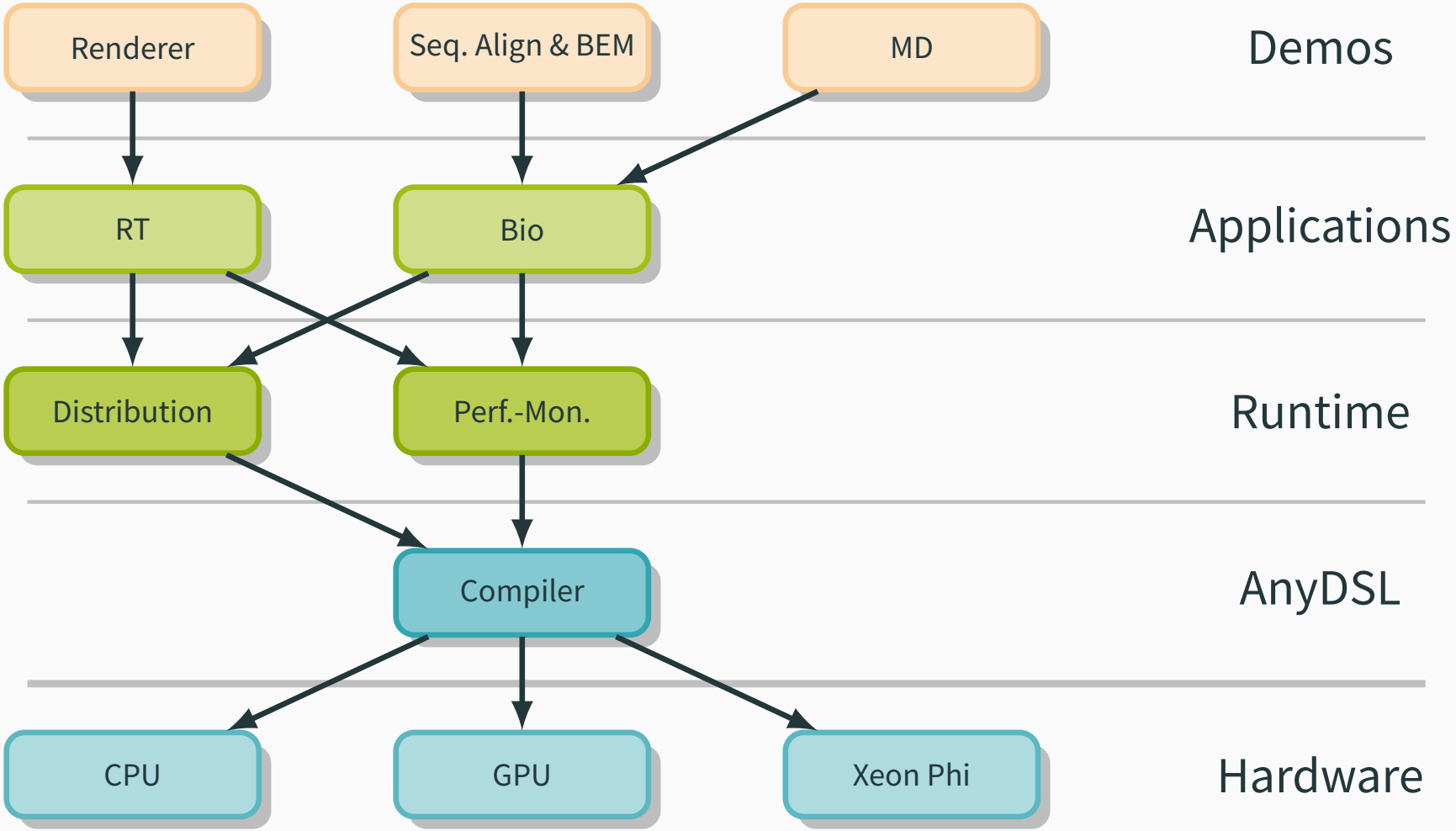
Metacca – Metaprogramming for Accelerators

Roland Leißa, UdS, DFKI, JGU, LSS, RRZE

UNIVERSITÄT DES SAARLANDES

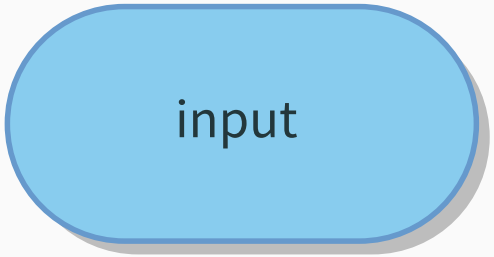
SIC Saarland Informatics Campus

Project Overview

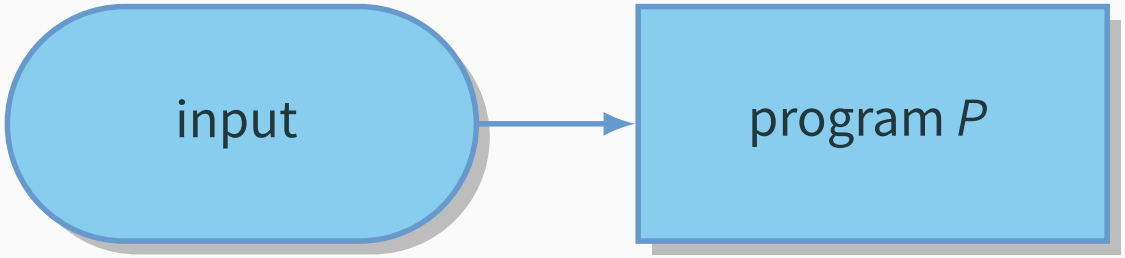


Partial Evaluation

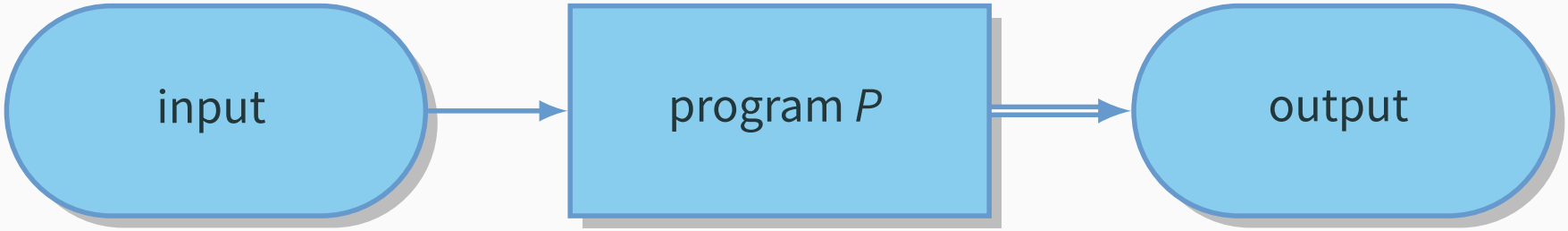
Partial Evaluation



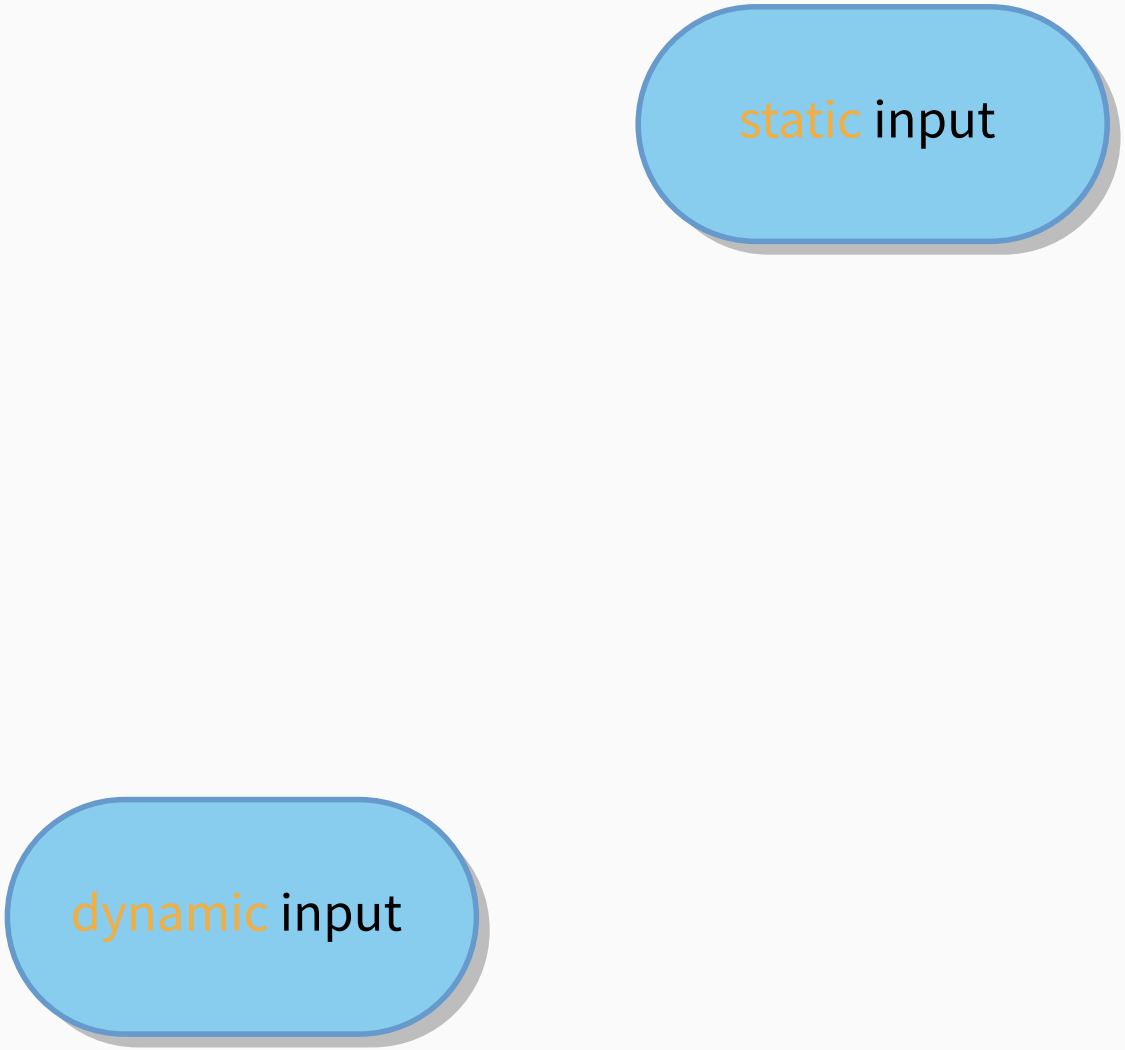
Partial Evaluation



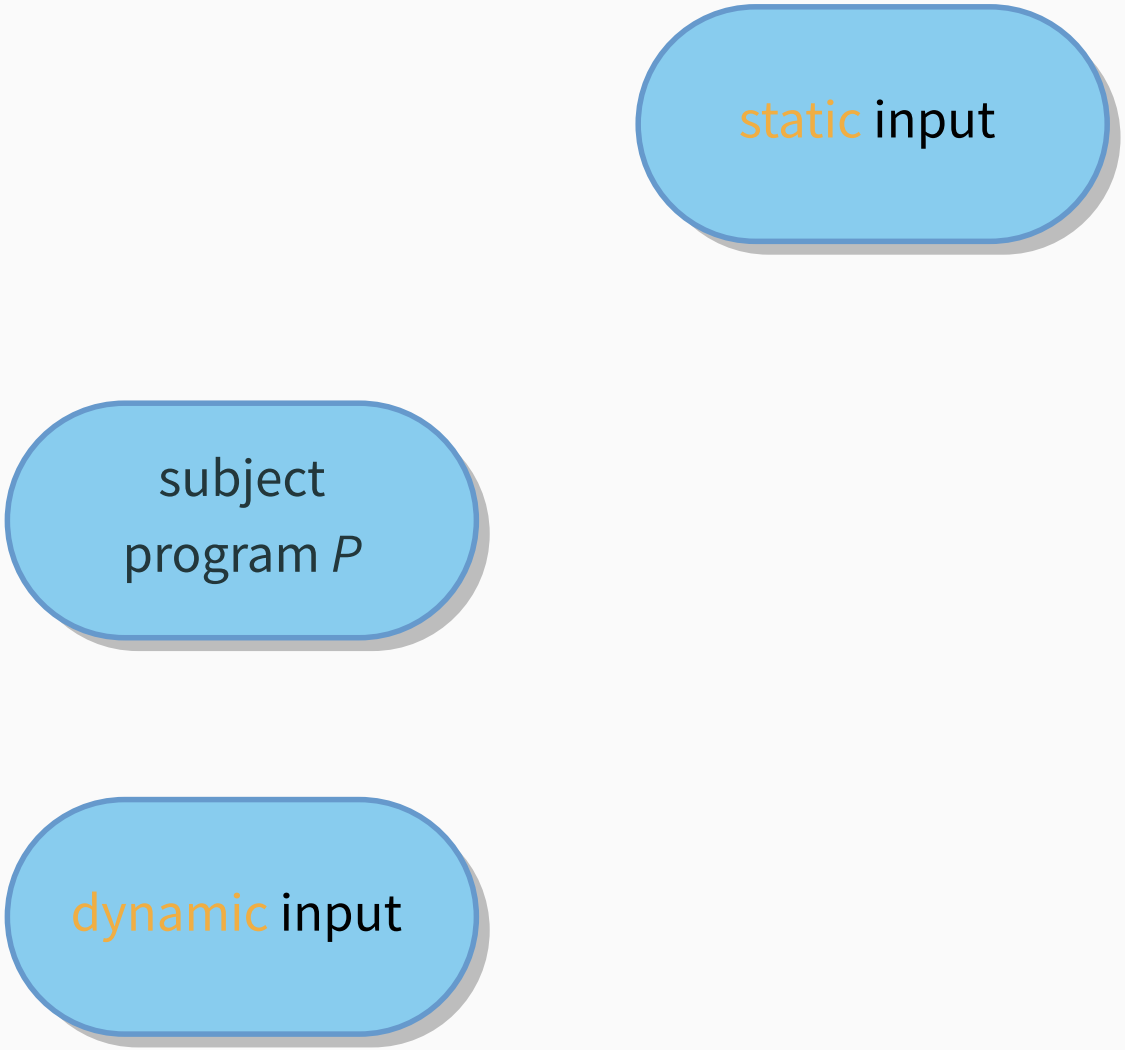
Partial Evaluation



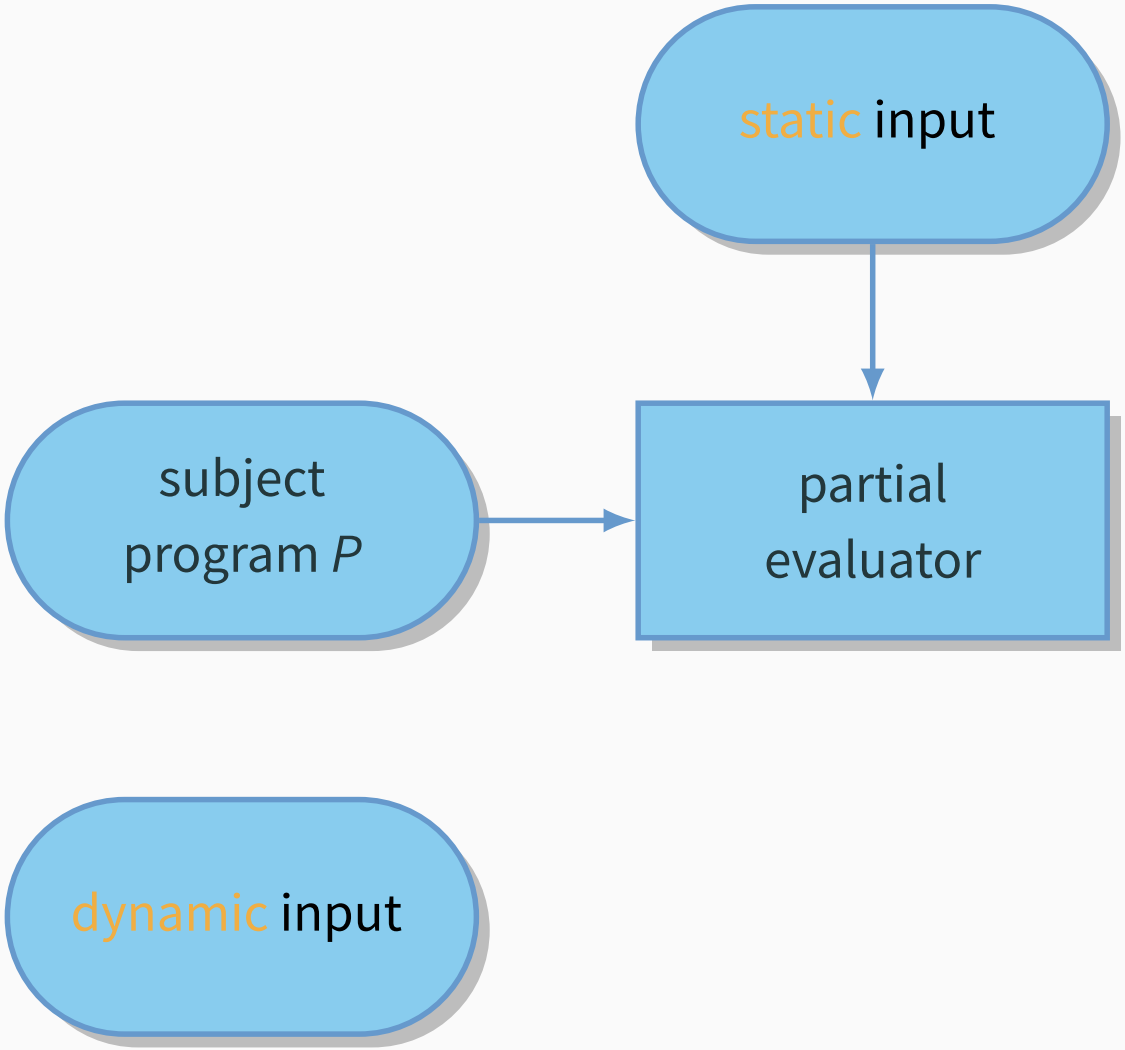
Partial Evaluation



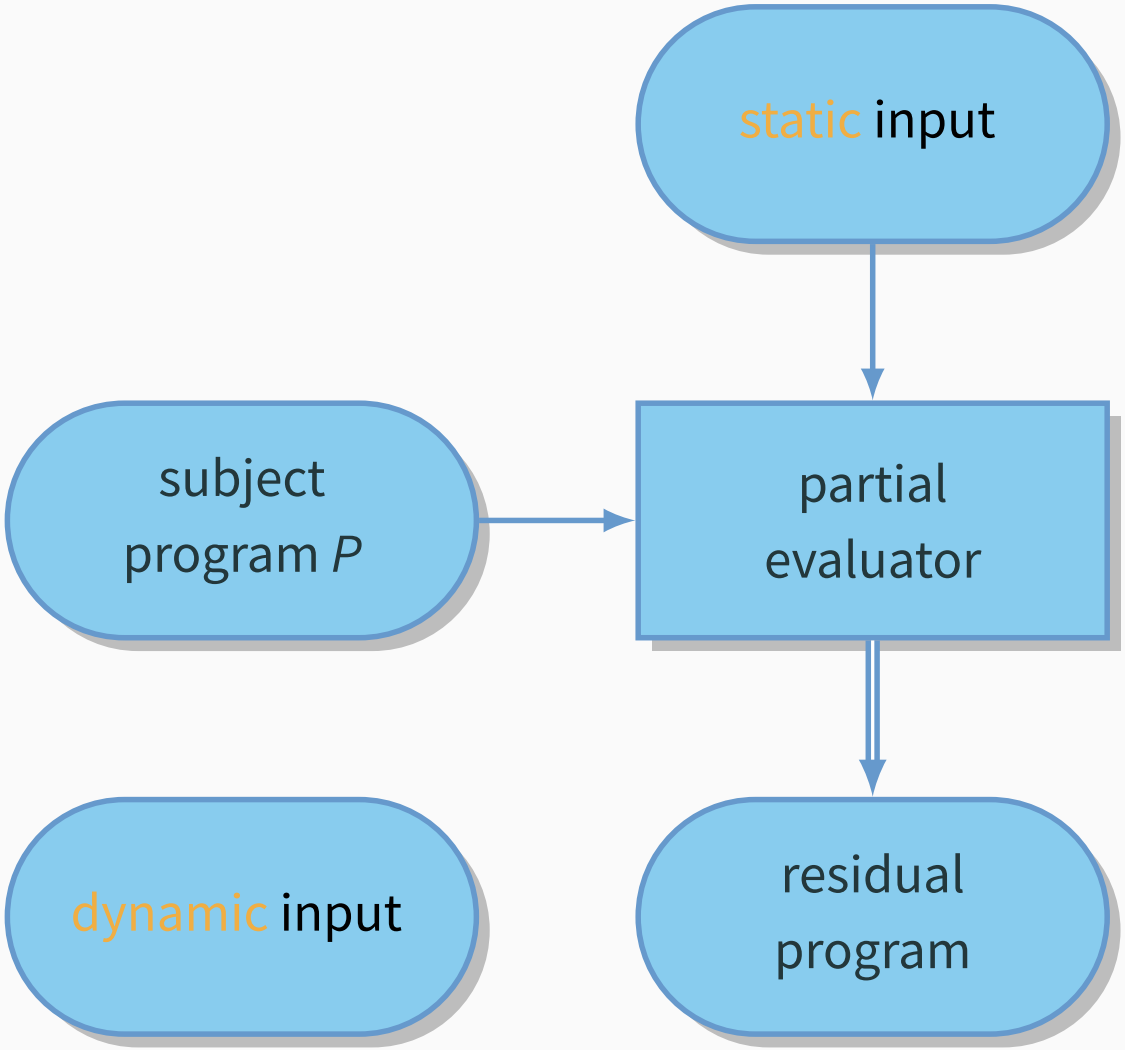
Partial Evaluation



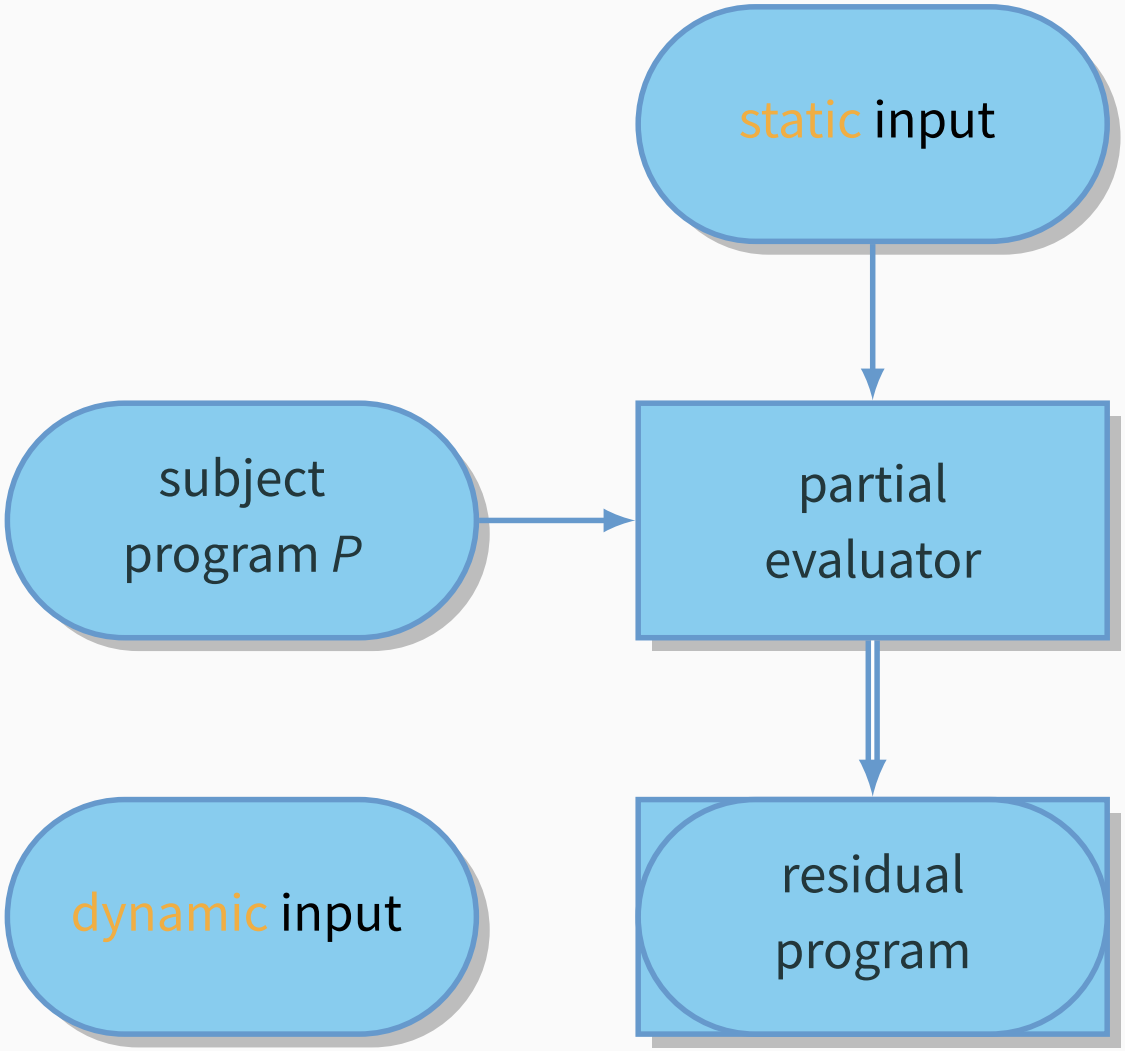
Partial Evaluation



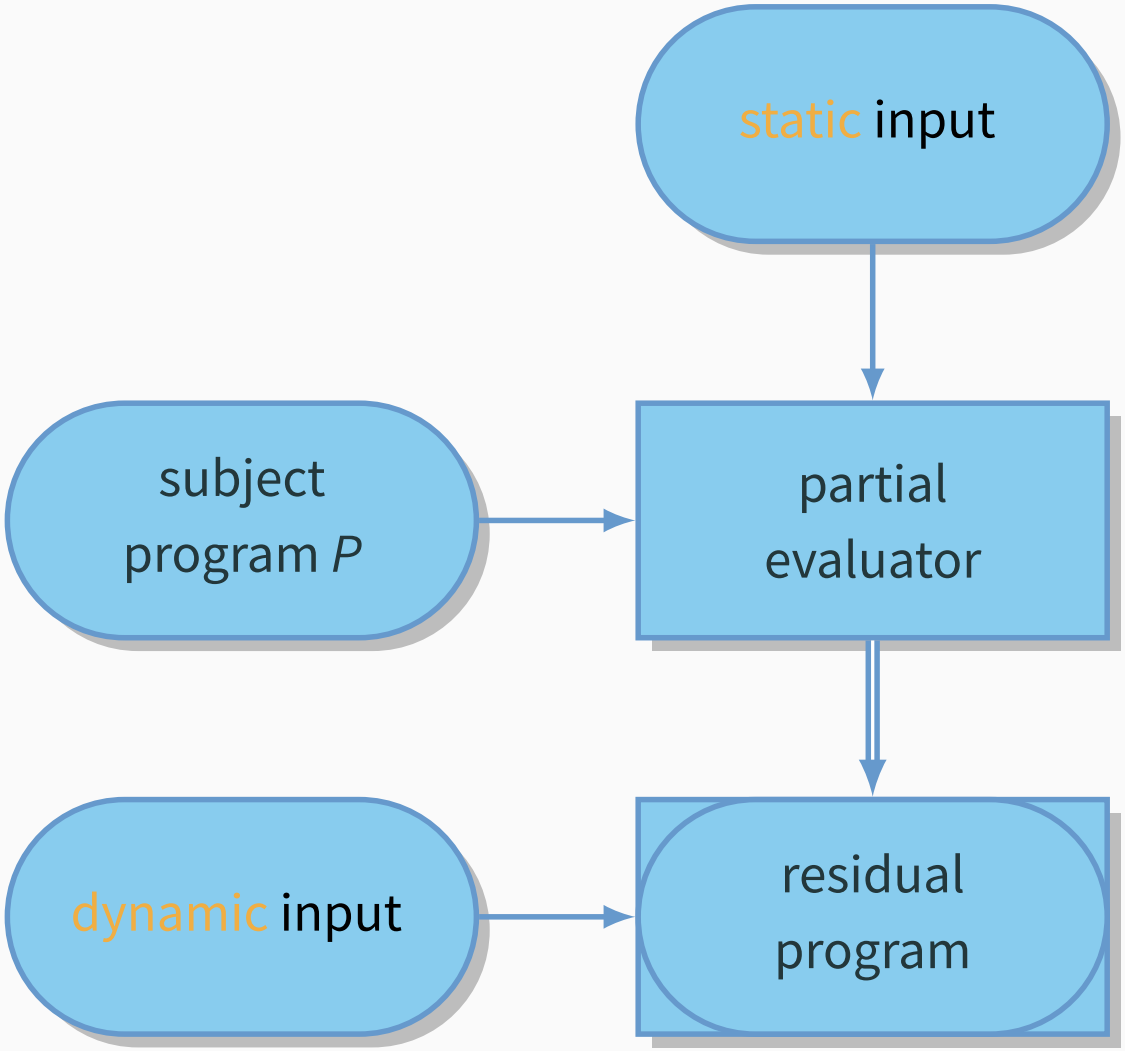
Partial Evaluation



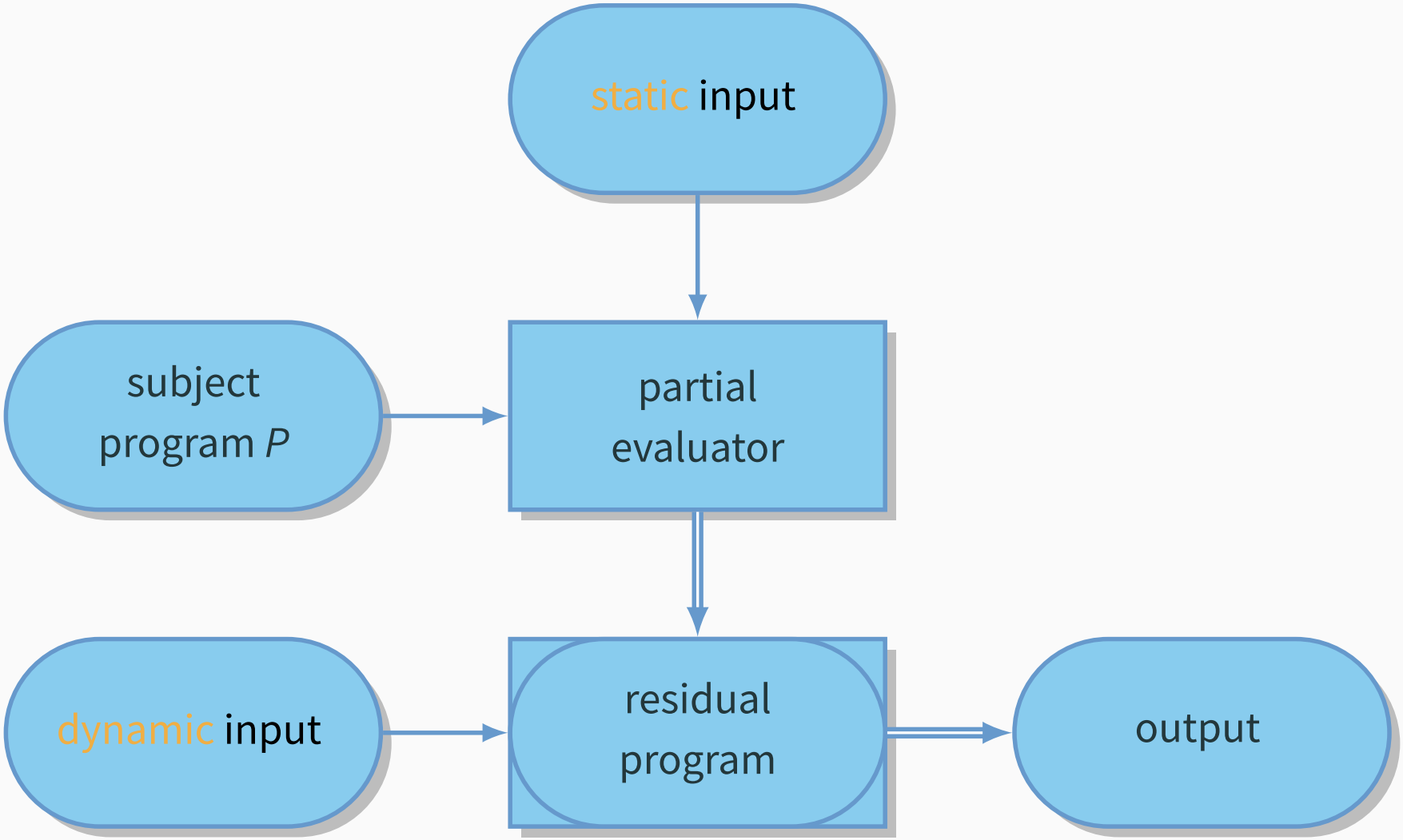
Partial Evaluation



Partial Evaluation



Partial Evaluation



Typical Use-Cases

- Higher-order Parameters:

  $\Rightarrow$ auto-annotated

Typical Use-Cases

- Higher-order Parameters:

  $\Rightarrow$ auto-annotated

- Small Wrappers:

```
fn @get_42() -> i32 { 42 }
```

Typical Use-Cases

- Higher-order Parameters:

  $\Rightarrow$ auto-annotated

- Small Wrappers:

```
fn @get_42() -> i32 { 42 }
```

- Recursive functions:

```
fn @(?n)pow(x: i32, n: i32) -> i32 { /*...*/ }
```

Typical Use-Cases

- Higher-order Parameters:

  $\Rightarrow$ auto-annotated

- Small Wrappers:

```
fn @get_42() -> i32 { 42 }
```

- Recursive functions:

```
fn @(?n)pow(x: i32, n: i32) -> i32 { /*...*/ }
```

Advantages over manual metaprogramming

- Polyvariance
- No clumsy quotations
- Well-typedness of the residuum

Target-specific Code Generation

Target-specific Code Generation

```
parallel(num_threads, a, b, |i| {
    body
});
```

Target-specific Code Generation

```
for i in parallel(num_threads, a, b) {
    body
}
```

Target-specific Code Generation

```
for i in vectorize(simd_width, a, b) {
    body
}
```

Target-specific Code Generation

```
for x, y in cuda(grid, block, a, b) {
    body
}
```

Target-specific Code Generation

```
for x, y in opencl(grid, block, a, b) {
    body
}
```

Target-specific Code Generation

```
for y in parallel(4, y_begin, y_end) {
    for x in vectorize(8, x_begin, x_end) {
        body
    }
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,       int,       fn(int)       -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)       -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

```
let bounds = (x_begin, y_begin, x_end, y_end);
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

```
let bounds = (x_begin, y_begin, x_end, y_end);
```

```
for x, y in combine(range, range)(bounds) {
    body
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

```
let bounds = (x_begin, y_begin, x_end, y_end);
```

```
let vec = @|num||a, b, body| vectorize(num, a, b, body);
let par = @|num||a, b, body|  parallel(num, a, b, body);

for x, y in combine(vec(8), par(4))(bounds) {
    body
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

```
let bounds = (x_begin, y_begin, x_end, y_end);
```

```
let vec = @|num||a, b, body| vectorize(num, a, b, body);
let par = @|num||a, b, body|  parallel(num, a, b, body);

for x, y in combine(vec(8), par(4))(bounds) {
    body
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

```
let bounds = (x_begin, y_begin, x_end, y_end);
```

```
let vec = @|num||a, b, body| vectorize(num, a, b, body);
let par = @|num||a, b, body|  parallel(num, a, b, body);

for x, y in combine(vec(8), par(4))(bounds) {
    body
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

```
let bounds = (x_begin, y_begin, x_end, y_end);
```

```
let vec = @|num||a, b, body| vectorize(num, a, b, body);
let par = @|num||a, b, body|  parallel(num, a, b, body);

for x, y in combine(vec(8), par(4))(bounds) {
    body
}
```

Program Transformations: Combining Loops

```
type Loop1D = fn(int,      int,      fn(int)      -> ()) -> ();
type Loop2D = fn(int, int, int, int, fn(int, int) -> ()) -> ();

fn combine(loop_x: Loop1D, loop_y: Loop1D) -> Loop2D {
    |x_begin, y_begin, x_end, y_end, body|
        loop_y(y_begin, y_end, |y|
            loop_x(x_begin, x_end, |x| body(x, y)))
}
```

```
let bounds = (x_begin, y_begin, x_end, y_end);
```

```
let vec = @|num||a, b, body| vectorize(num, a, b, body);
let par = @|num||a, b, body|  parallel(num, a, b, body);

for x, y in tile(512, 32, vec(8), par(4))(bounds) {
    body
}
```

Image Processing: User Code

```
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);
```

Image Processing: Implementation

```
type BinOp = fn(i32, i32) -> i32;

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)
}
```

Performance

Blur: Ours vs. Halide

CPU: +12%

GPU: +7%

Performance

Blur: Ours vs. Halide

CPU: +12%

GPU: +7%

Harris-Corner: Ours vs. Halide

CPU: +37%

GPU: +44%

Other Demos

Ray Traversal

Embree: -15% to +13%

Optix: -19% to -2%

ARM support

Other Demos

Ray Traversal

Embree: -15% to +13%

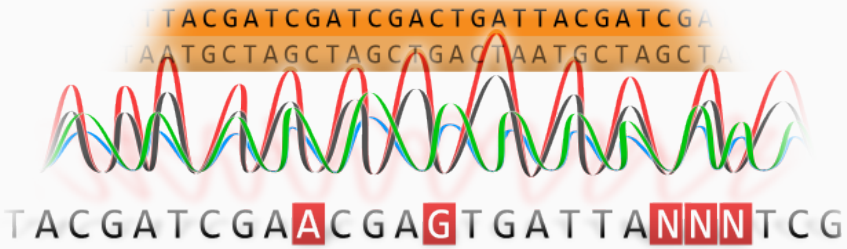
Optix: -19% to -2%

ARM support

Genome Sequence Alignment

SeqAn/Parasail: -3% to +11%

NVBIO: -3% to +10%



Renderer

AnyDSL Renderer

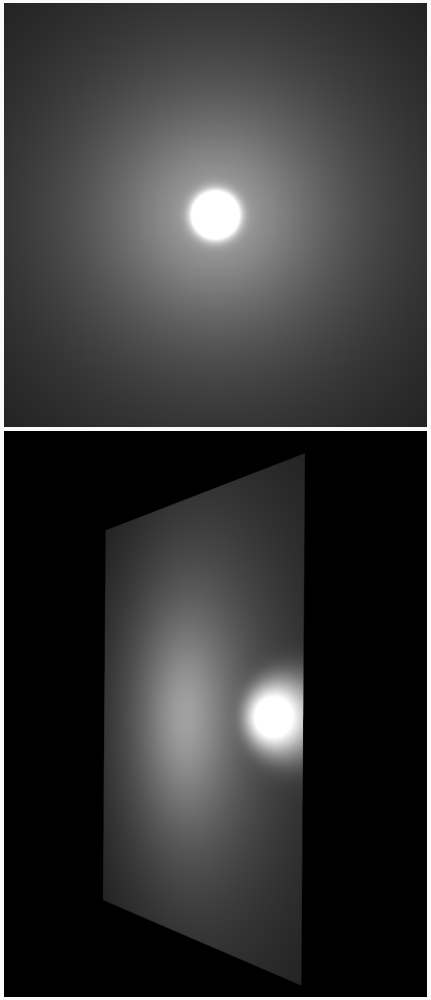
- Now a complete renderer (PT)
- Lights & Materials controlled by shaders
- CPU: Vectorized
- 2 GPU versions: Megakernel, Streaming

Diffuse Shader

```
fn @shader(math: Intrinsics,
           scene: Scene,
           ray: Ray,
           hit: Hit,
           surf: SurfaceElement
           ) -> Material {
  let kd = make_color(0.6f, 0.6f, 0.6f);
  let bsdf = make_diffuse_bsdf(surf, kd);
  make_material(bsdf)
}
```

Glossy Shader

```
fn @shader(math: Intrinsics,
           scene: Scene,
           ray: Ray,
           hit: Hit,
           surf: SurfaceElement
           ) -> Material {
  let kd = make_color(0.6f, 0.6f, 0.6f);
  let diff = make_diffuse_bsdf(surf, kd);
  let ks = make_color(0.5f, 0.5f, 0.5f);
  let ns = 96.0f;
  let spec = make_phong_bsdf(surf, ks, ns);
  let k = 0.6f / (0.5f + 0.6f);
  let bsdf = make_mix_bsdf(diff, spec, k);
  make_material(bsdf)
}
```



Textures

```
fn @shader(math: Intrinsics,
           scene: Scene,
           ray: Ray,
           hit: Hit,
           surf: SurfaceElement
           ) -> Material {
  let texture = make_texture(
      math,
      make_repeat_border(),
      make_bilinear_filter(),
      scene.images(0)
  );
  let uv = vec4_to_2(surf.attr(0));
  let kd = texture(uv);
  let bsdf = make_diffuse_bsdf(surf, kd);
  make_material(bsdf)
}
```

Textures

```
fn @shader(math: Intrinsics,
           scene: Scene,
           ray: Ray,
           hit: Hit,
           surf: SurfaceElement
           ) -> Material {
  let texture = make_texture(
      math,
      make_clamp_border(),
      make_bilinear_filter(),
      scene.images(0)
  );
  let uv = vec4_to_2(surf.attr(0));
  let kd = texture(uv);
  let bsdf = make_diffuse_bsdf(surf, kd);
  make_material(bsdf)
}
```

Textures

```
fn @shader(math: Intrinsics,
           scene: Scene,
           ray: Ray,
           hit: Hit,
           surf: SurfaceElement
           ) -> Material {
  let texture = make_texture(
      math,
      make_repeat_border(),
      make_nearest_filter(),
      scene.images(0)
  );
  let uv = vec4_to_2(surf.attr(0));
  let kd = texture(uv);
  let bsdf = make_diffuse_bsdf(surf, kd);
  make_material(bsdf)
}
```

Textures

```
fn @shader(math: Intrinsics,
           scene: Scene,
           ray: Ray,
           hit: Hit,
           surf: SurfaceElement
           ) -> Material {
  let texture = make_texture(
      math,
      make_repeat_border(),
      make_bilinear_filter(),
      scene.images(0)
  );
  let uv = vec4_to_2(surf.attr(0));
  let kd = texture(uv);
  let bsdf = make_diffuse_bsdf(surf, kd);
  make_material(bsdf)
}
```

Performance

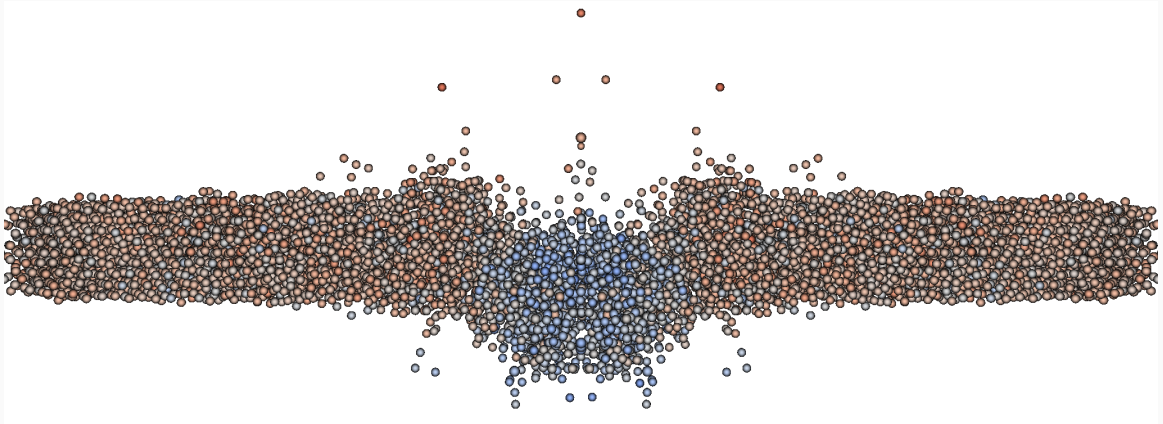
Preliminary results: 31 FPS on GPU, 9 FPS on CPU

AnyDSL FPGA Support

- Funded via IVCI
- In collaboration with Akif Ötzkan (FAU Erlangen-Nuremberg)
- C-based backends for Altera OpenCL and Vivado HLS
- Prototype mapping for stencil codes
- Good performance compared to Halide & Hipacc

Molecular Dynamics

Code Generation for Molecular dynamics



- Simulation of the trajectories of a large number of particles based on their interactions
- The computation of short-range interactions is an important use case in many simulations
- Most implementations employ a combination of **cell decomposition** and **neighbor lists**

Kernel Generation through Partial Evaluation

```
// example application
// the kernel is passed as an anonymous function here
execute(particles, |pi, ci, cluster_size| {
    // update particle pi in cluster ci
});
```

- By partially evaluating **execute** with respect to its second argument, code generation is triggered.
- All details about the target platform are hidden within its implementation
- To generate code for different platforms, different implementations must be provided

Kernel Generation on the CPU

```
fn execute(particles: Particles, kernel: Kernel) -> () {
    for ci in parallel(/*...*/) {
        // ...
        for i in vectorize(/*...*/) {
            kernel(/*...*/);
        }
    }
}
```

Kernel Generation on the GPU

```
fn execute(particles: Particles, kernel: Kernel) -> () {
    let acc = get_accelerator(device_id);
    let size = particles.number_of_clusters;
    let grid = (size * get_cluster_size(), 1, 1);
    let block = (get_cluster_size(), 1, 1);
    for bid, bdim, gid in acc.exec(grid, block) {
        let (gidx, _, _) = gid;
        let (bidx, _, _) = bid;
        let (bdimx, _, _) = bdim;
        kernel(gidx(), bidx(), bdimx());
    }
    acc.sync();
}
```

Single-Core Performance in FLOPS/cycle

Benchmark settings

- AnyDSL: LLVM version 5.0.1 with RV for vectorization, -O3, -march=native
- MiniMD: Intel C compiler version 18 with -O3, -xHost, -qopt-zmm-usage=high (SKL)
- Double-precision floating-point computations

Processor	AnyDSL	MiniMD
Skylake	5.816 (AVX512)	3.618 (AVX512)
Broadwell	2.928 (AVX2)	1.695 (AVX2)
Ivy Bridge	2.103 (AVX)	1.034 (AVX)

Acceleration on the GPU

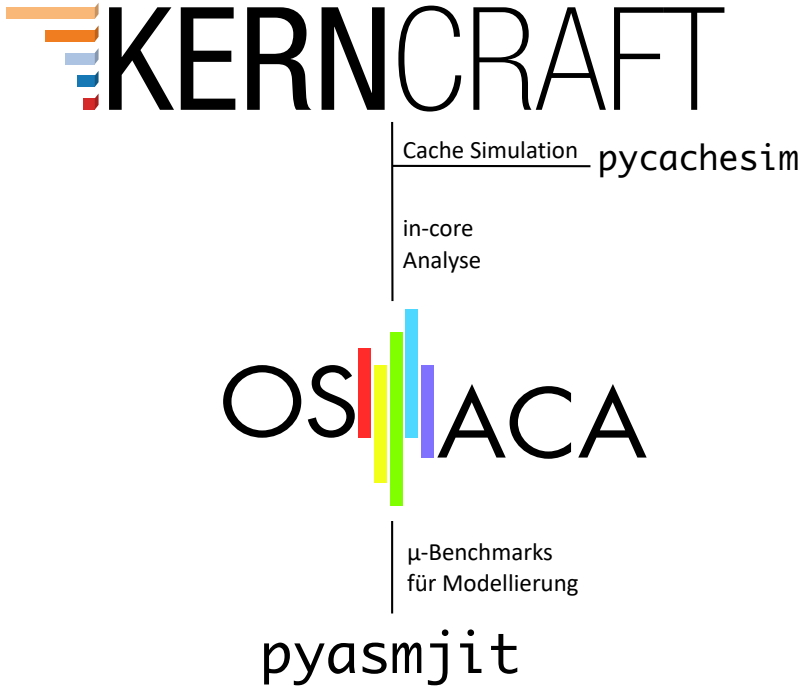
- CPU test platform: Intel Xeon E3-1275 v5 with four cores
- GPU test platform: NVIDIA GTX 1080, AnyDSL Backend: NVVM, Cluster size: 32
- Double-precision floating-point computations
- The generated GPU code runs around 5 times faster

Particles	AnyDSL (AVX2)	AnyDSL (GPU)
100 000	1044.39 ms	194.101 ms
500 000	4300.2 ms	826.627 ms
1 000 000	7652.97 ms	1684.18 ms
2 000 000	15014.7 ms	3294.85 ms

Performance Monitoring

RRZE: Aktuelle Arbeiten

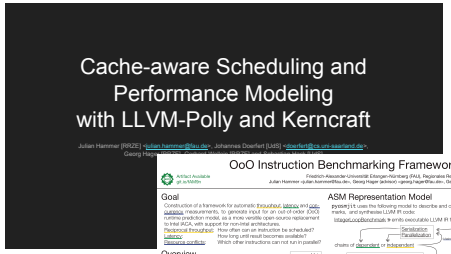
- `Kerncraft` & `pycachesim`: Laufzeitvorhersage für Kernels
- `OSACA`: In-core Laufzeitvorhersage für Assembler Basic Blocks
- `pyasmjit`: Micro-Benchmark Generierung für Modellierung
- Automatische Modellbildung, gemeinsam mit Univ. Saarlandes



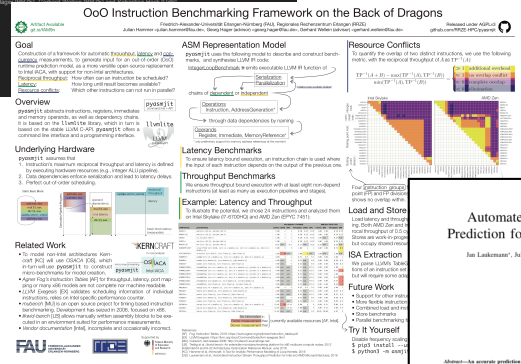
Julian Hammer <julian.hammer@fau.de>

RRZE: Aktuelle Beiträge

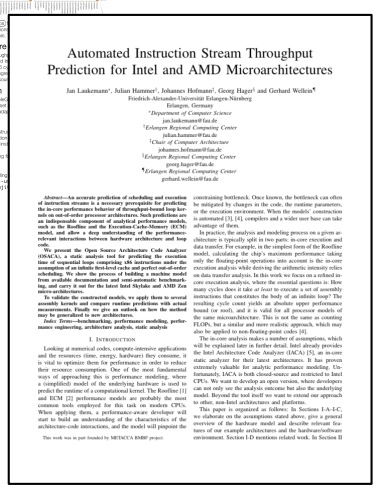
- Workshopbeitrag auf LLVM-CGO18 über automatisches Blocking in LLVM, gemeinsam mit Universität des Saarlandes



- Poster bei SC18 über pyasmjit



- Workshoppaper bei PMBS18 (in review) über OSACA



Conclusions

Conclusions

- Partial evaluation for zero-cost abstractions
- Target-specific code generation triggered via higher-order functions
- $\Rightarrow$ High-performance libraries that hide hardware peculiarities

Conclusions

- Partial evaluation for zero-cost abstractions
- Target-specific code generation triggered via higher-order functions
- $\Rightarrow$ High-performance libraries that hide hardware peculiarities

Thank you. Questions?