

FUNCTIONAL PROGRAMMING FOR PARALLEL AND CONCURRENT SYSTEMS

-

THEORY AND PRACTICE

GABRIELE KELLER
SOFTWARE TECHNOLOGY GROUP
UTRECHT UNIVERSITY



RESEARCH

Functional Programming

Probabilistic Programming

Verification & Testing

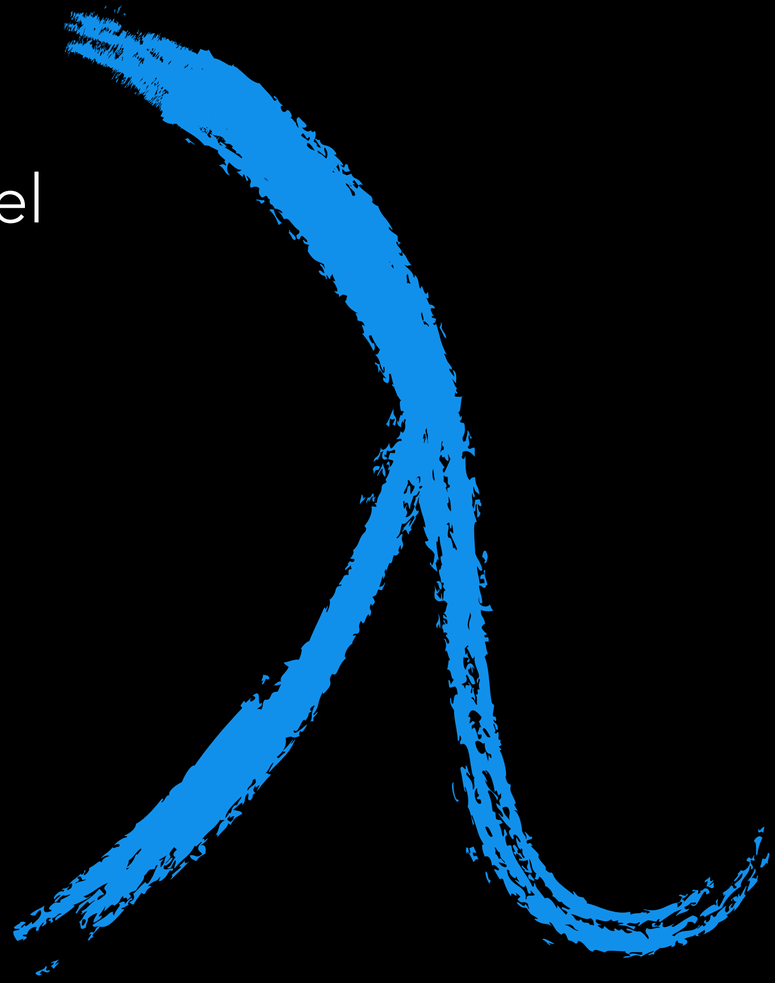
Automatic Differentiation

Parallel Programming

PL and Hardware boundary

FUNCTIONAL PROGRAMMING FOR CONCURRENT AND PARALLEL PROGRAMMING

- Functional programming has been touted for parallel and concurrent computing from early on
- Programs are functions which could be evaluated in parallel, not sequential instructions
- In reality, the implicit parallelism is not sufficient



FUNCTIONAL PROGRAMMING FOR CONCURRENT AND PARALLEL PROGRAMMING

Success stories of explicit concurrent & parallel programming:

Software Transactional Memory

Parallel Array Programming



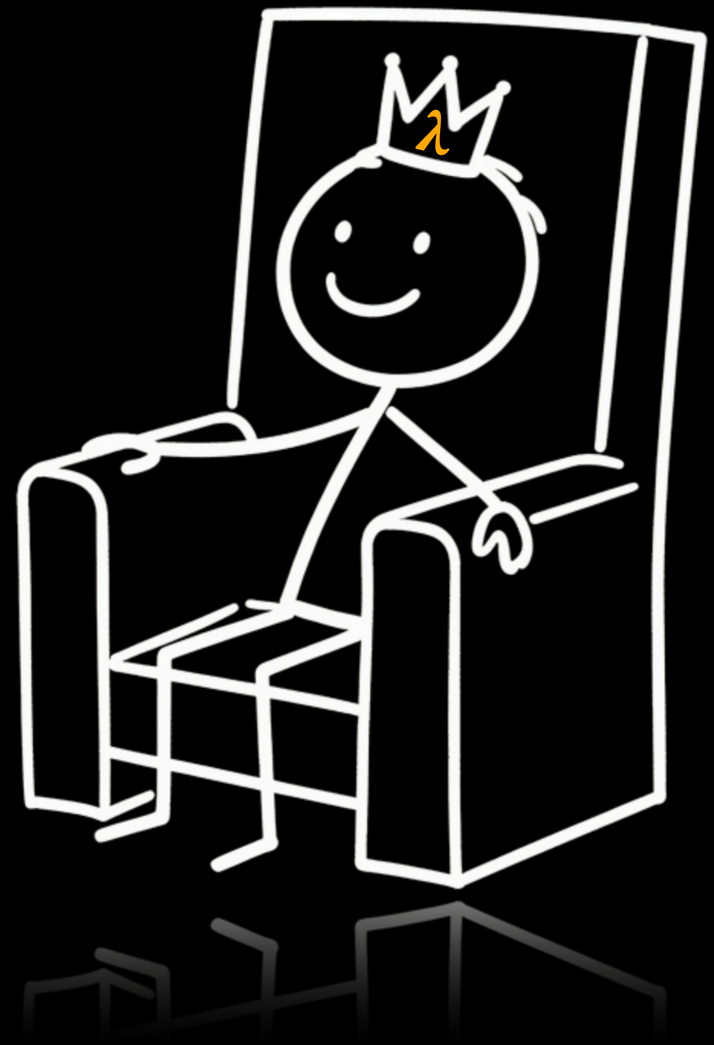
WHAT IS (PURE) FUNCTIONAL PROGRAMMING

- What are the characteristics of functional programming?



FUNCTIONAL PROGRAMMING

- Higher-order functions are not a distinctive feature of FP anymore
  - they are supported in most mainstream languages
- However, in FP higher-order functions are not just another possibility, but often the preferred method
- This influences the design of our code!



ONLY PURE FUNCTIONS!

- Pure functions have no implicit observable side-effects
  - they can be written in (almost) any language, but purity is usually not enforced
  - it’s important how natural/ergonomic is it to write pure functions in the language



NO IMPLICIT, OBSERVABLE SIDE EFFECTS

- It’s about observable side effects!
  - even the evaluation of a simple expression `(3 + 2)` has side effects, but they are not observable by the program
- No observable side effect means any function or operation applied to the same arguments will always yield the same result

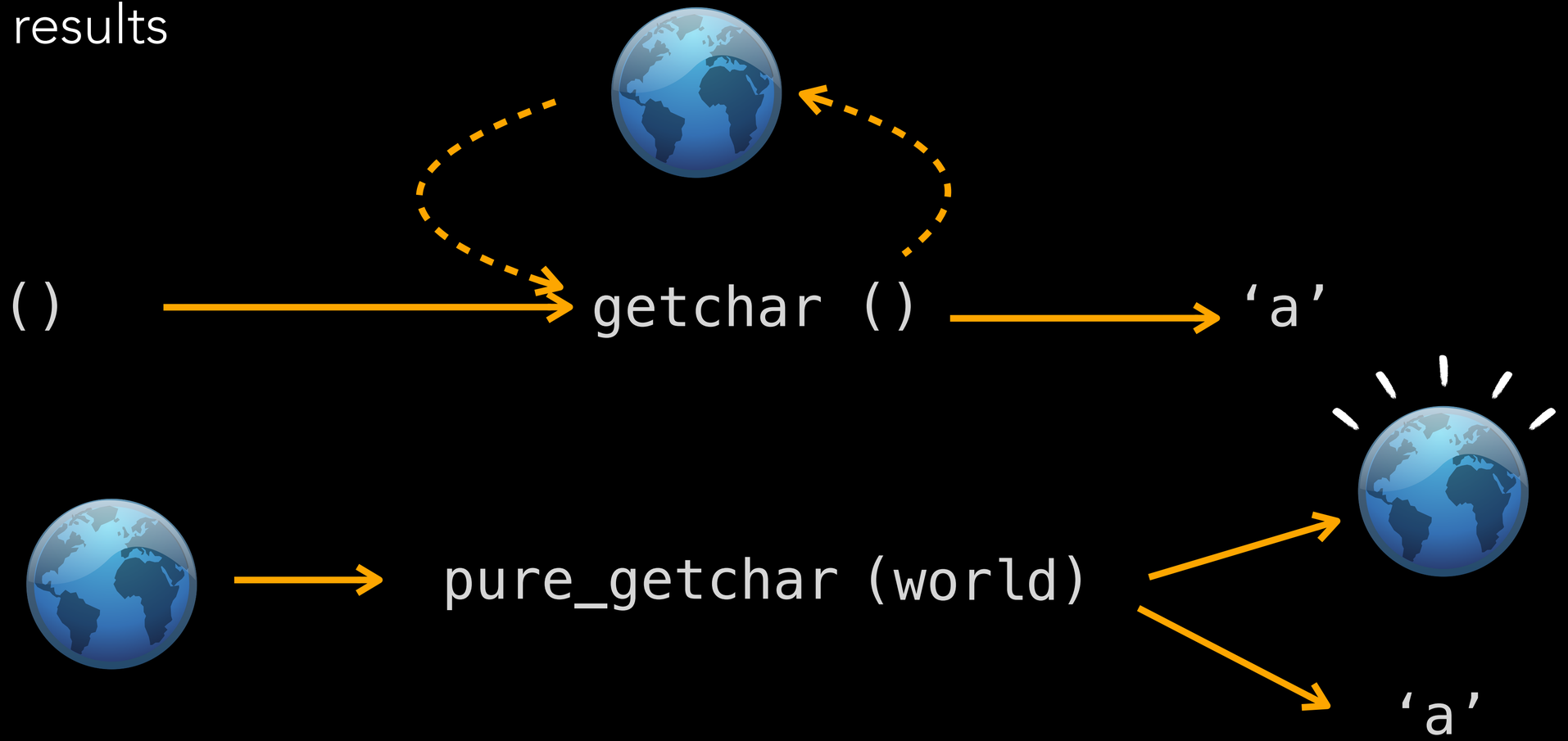
```
reverseString “GOTO!”
x + y
a[i]
getchar ()
```

NO IMPLICIT, OBSERVABLE SIDE EFFECTS

- Any useful program has observable side effects
  - without any I/O effects, the execution of a program would be pointless
- Destructive updates of data structures are necessary for the efficiency

NO IMPLICIT, OBSERVABLE SIDE EFFECTS

- Pure functions: make the side effect explicit in the type of the arguments and results



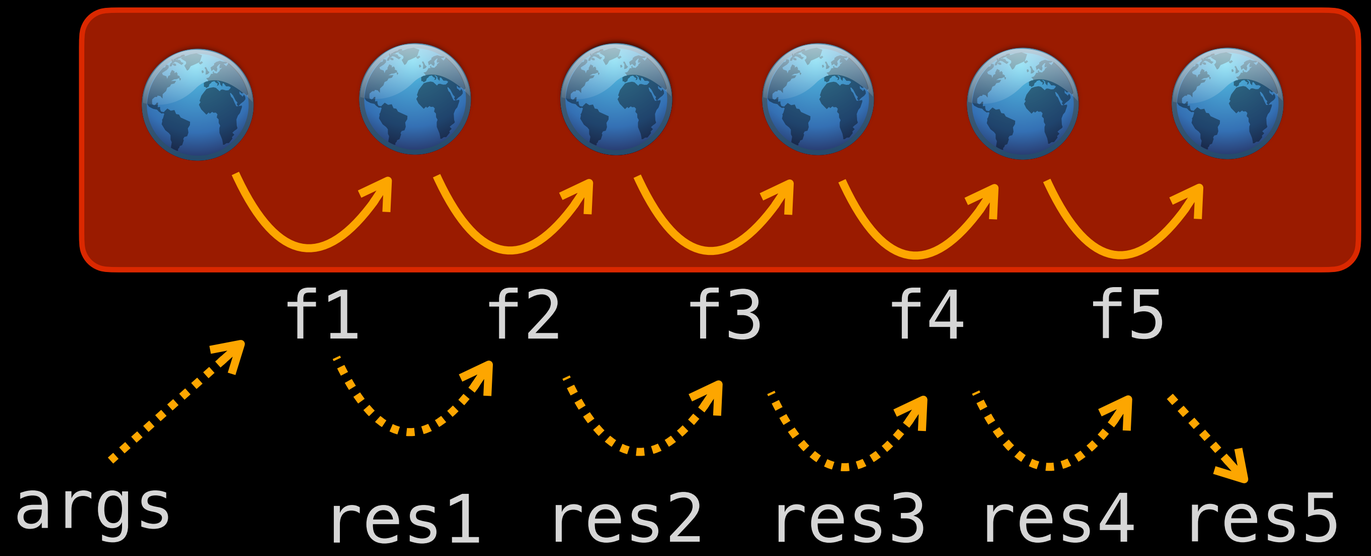
```
void add () {
  a = read_int ();
  b = read_int ();
  print (to_string (a + b));
}

add (world) =
  let
    (world1, a) = read_int world
    (world2, b) = read_int world1
  in print (to_string (a+b)) world2
```

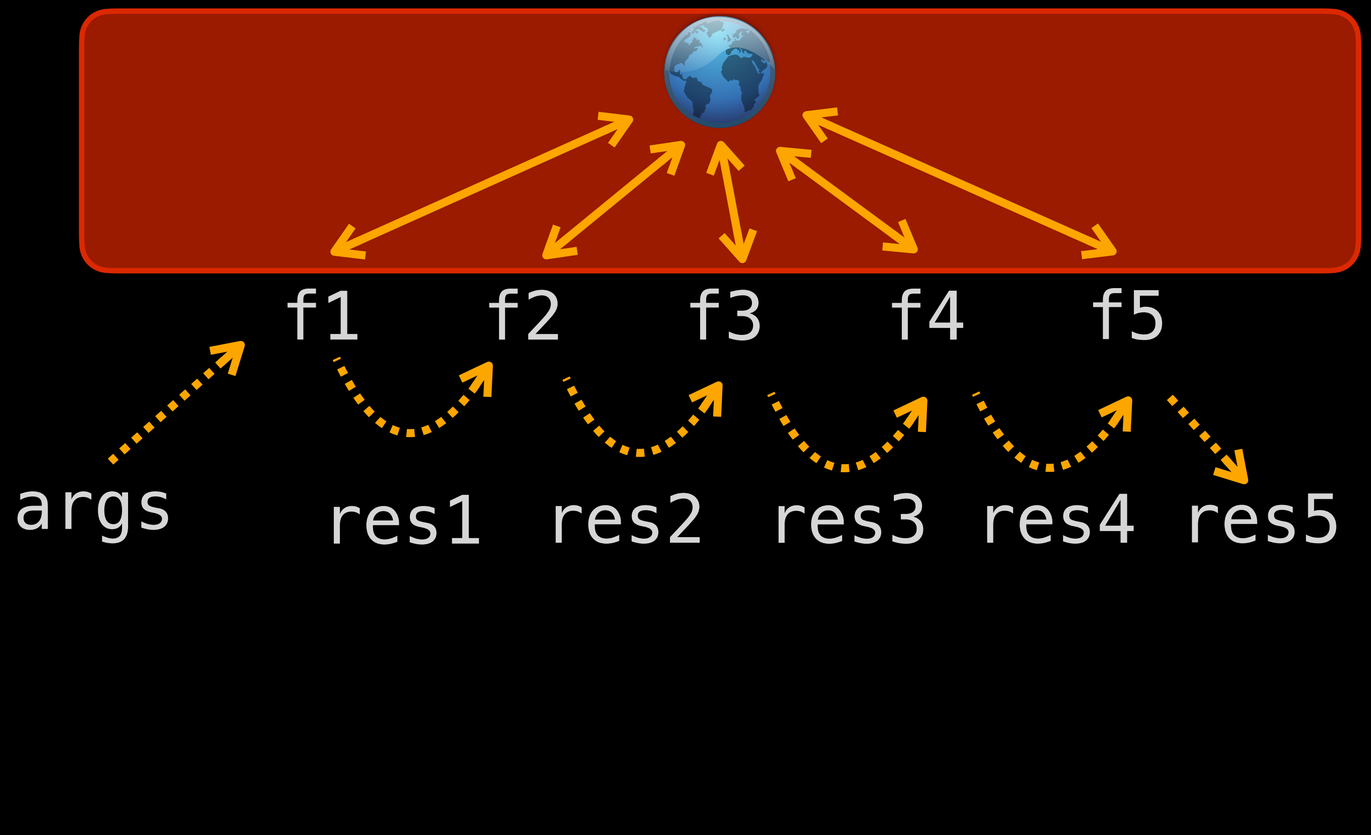
```
argType -> World -> (World, resType1)

resType1 -> World -> (World, resType2)
```

necessary to make sure the ‘world’ is passed through correctly!



We can implement is via destructive updates behind the scene!

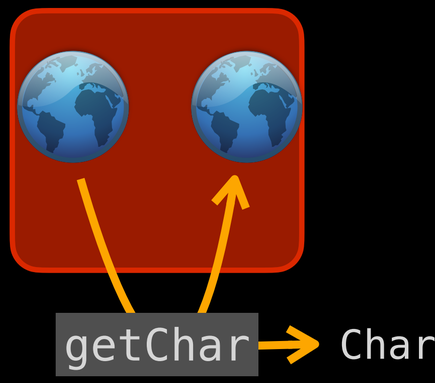


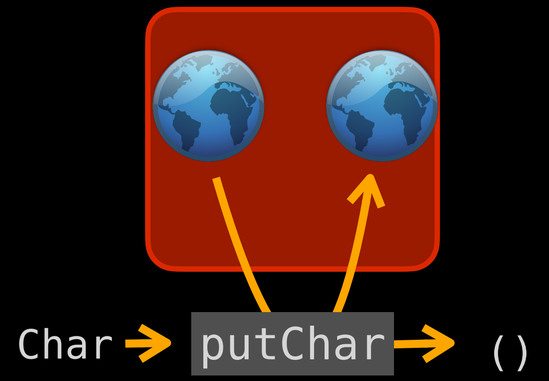
HASKELL

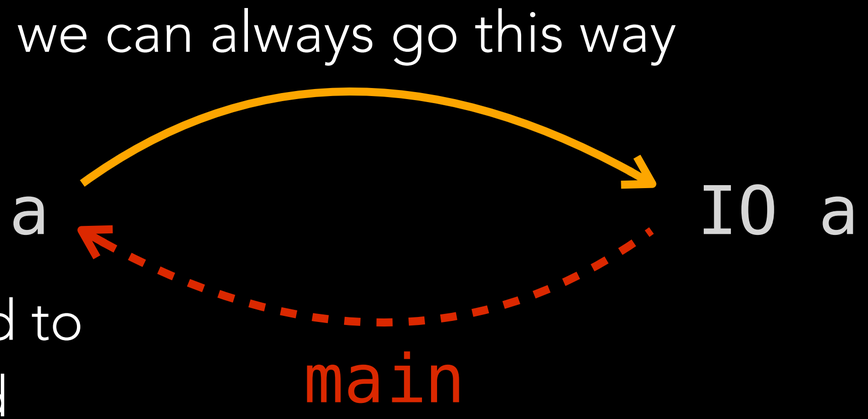
- Hiding the world in an abstract data type:

```
World-> (World, resType) = IO resType

getChar:: IO Char
putChar:: Char -> IO ()
```





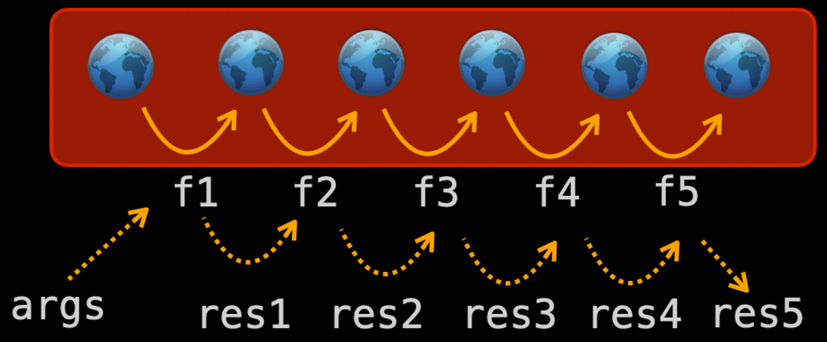


to go back, we need to
apply it to the world

HASKELL

```
World-> (World, resType) = IO resType
```

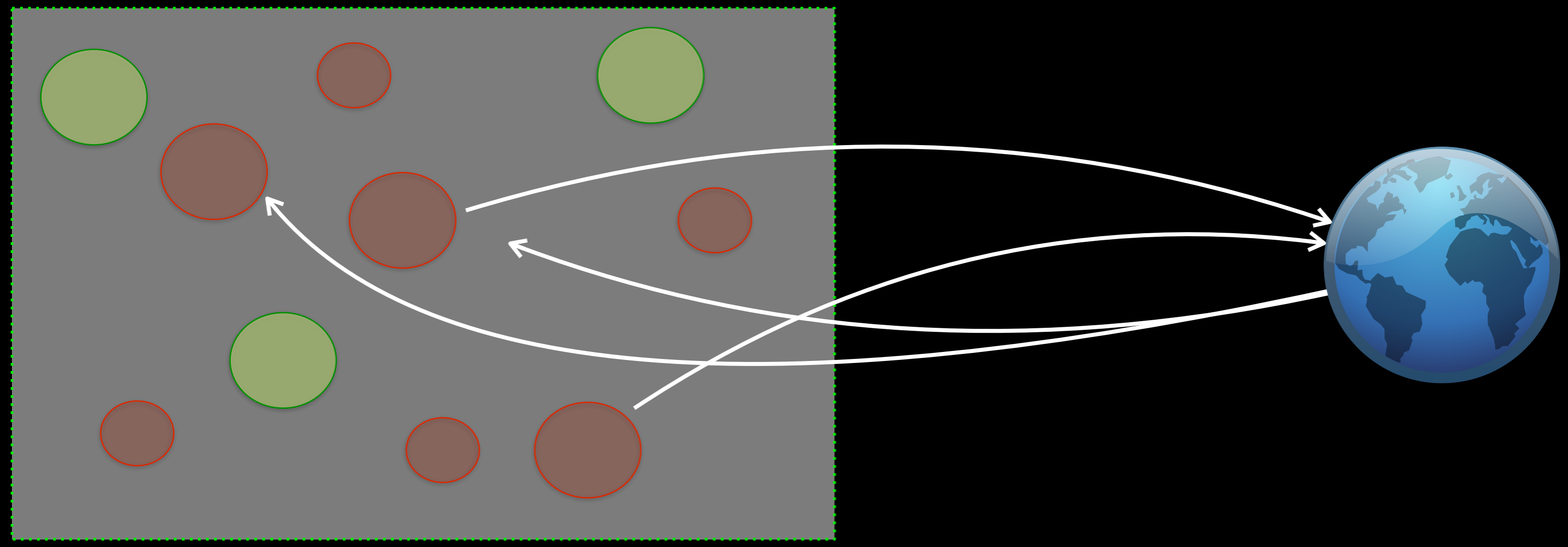
```
f args =  do
  res1 <- f1 args
  res2 <- f2 args res1
  res3 <- f3 args res1 res2
  …
```

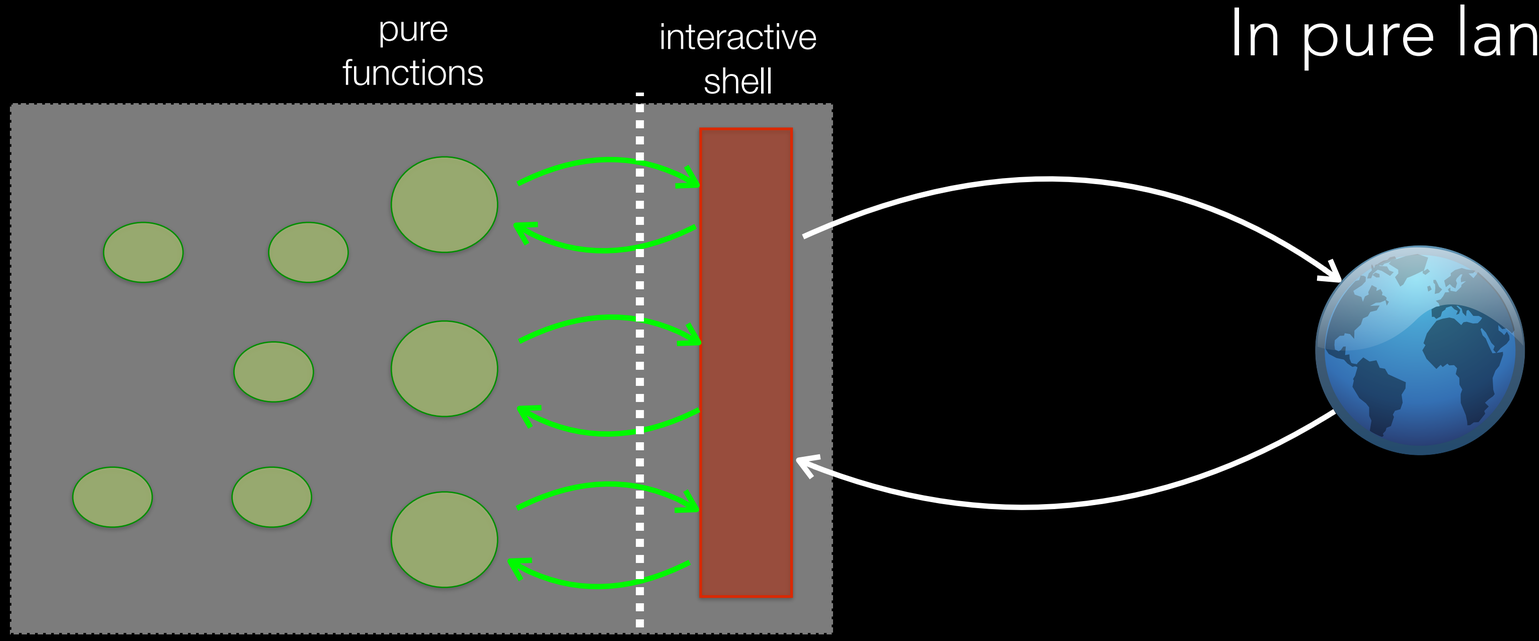



```
addNumbers =  do
  x <- read_int
  y <- read_int
  print (to_string (x + y))
```

- Pure functional languages differ in
  - how they enforce the constraints of threading the state through computations
  - how they try to make is convenient to use
- but the basic idea of threading the state is the same!

Modelling the state in the type has an effect on the design!





In pure languages

NOW LET’S TALK ABOUT CONCURRENCY!

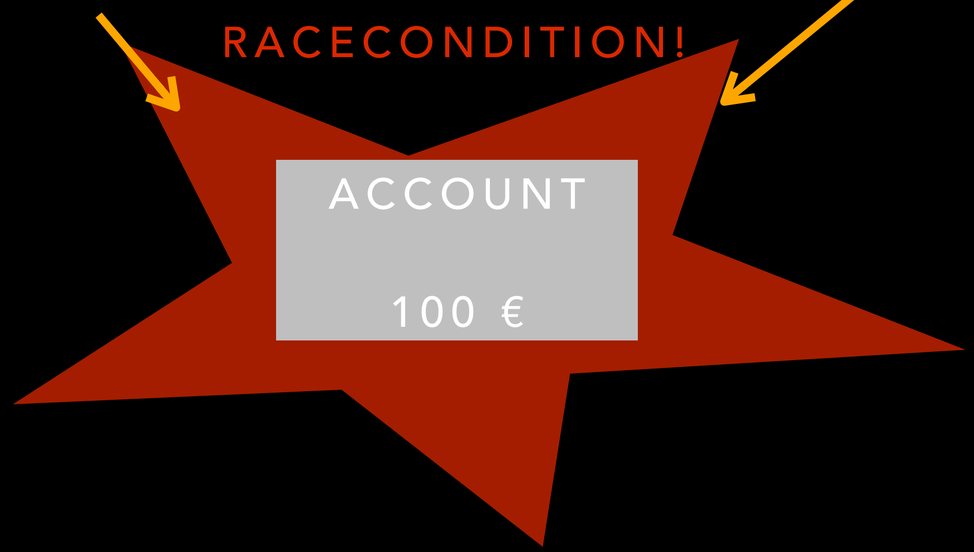
SOFTWARE TRANSACTIONAL MEMORY

CONCURRENT ACCESS TO SHARED STATE VIA LOCKS

Withdraw from an account (shared state): check balance, withdraw if it doesn't cause the account to go negative

```
Thread 1:
withDraw account amount1:
  check balance,
    if ok, deduct amount
```

```
Thread 2:
withDraw account amount1:
  check balance,
    if ok, deduct amount
```



CONCURRENT ACCESS TO SHARED STATE

Withdraw from an account (shared state): check balance, withdraw if it doesn't cause the account to go negative

```
Thread 1:
withDraw account amount1:
  lock account
  check balance,
    if ok, deduct amount
  unlock account
```

```
Thread 2:
withDraw account amount1:
  lock account
  check balance,
    if ok, deduct amount
  unlock account
```



CONCURRENT ACCESS TO SHARED STATE

Transfer from a `account1` to `account2`: check balance of `account1`, withdraw if it doesn't cause the account to go negative, and add amount to `account2`:

```
transfer (account1, account2, amount):
    withDraw account1 amount
    withDraw account2 (-amount)
```



CONCURRENT ACCESS TO SHARED STATE

- Locking has many problems:
  - error prone (deadlocks, forgetting to take out locks,…)
  - not modular
  - pessimistic

SOFTWARE TRANSACTIONAL MEMORY

- Critical sections which depend on shared state:
  - read from shared state (without locking)
  - calculate updates
  - in one atomic operation:
    - check if state variables are still the same
      - if so, apply updates
      - otherwise, roll back (possibly retry)
- has been around since the mid 90's

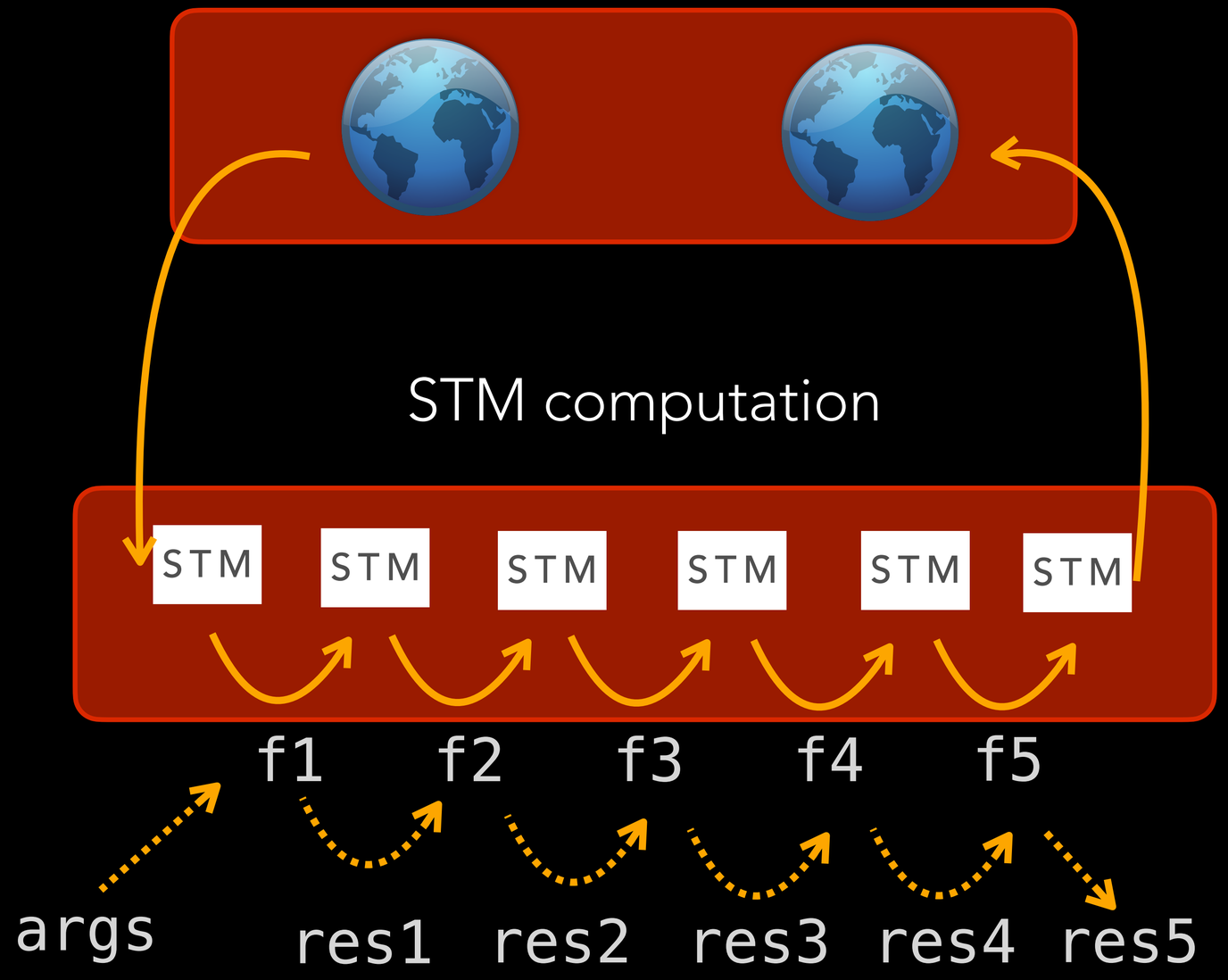
may only have effects we can control!

```
transfer amount account1 account2:
  withDraw account1 amount:
    check balance account1
    if ok, deduct amount
  withDraw account2 (-amount)
    check balance account2
    if ok, deduct (-amount)
```



```
account1.balance ?= 100
account2.balance ?= 50

account1.balance -> 70
account2.balance -> 80
```



checking and applying the changes:

```
atomic:: STM resT -> IO resT
```

combinators:

```
retry:: STM a -> STM a
```

…

We can express such a computations as computation over a controlled shared state:

```
Stm -> (Stm, resType)
        =
    STM resType
```

STM IMPLEMENTATIONS

- STM provides a high-level, powerful programming model for safe access to shared state
- STM implementations for almost all common programming languages
- Mainly took hold in Haskell, and Clojure
- Controlling side-effects is essential for STM!
    - controlled by the type system of Haskell
    - STM integrates well with the typical design of Haskell applications
- Combinators are in Haskell a natural way to build complex behaviour

PARALLELISM



ALL ABOUT PERFORMANCE

DATA PARALLEL COMPUTING



- Applying operations in parallel to large collections of data, execute on multicore, GPUs
- Usually, large (multi-dimensional) arrays
- With GPUs, multi-core machines, also on commodity hardware

?



FUNCTIONAL ARRAY PROGRAMMING?

- Array programming doesn't have to be index based!

```
sum = 0;
for (i = 0; i < N; i++) {
  brr[0] = 0;
  for (j = 0; j < M; j++) {
   brr[i] += arr[i][j];
  }
}
```

```
brr = map sum arr
```

In parallel array programming, we apply operations collectively, not on individual elements!

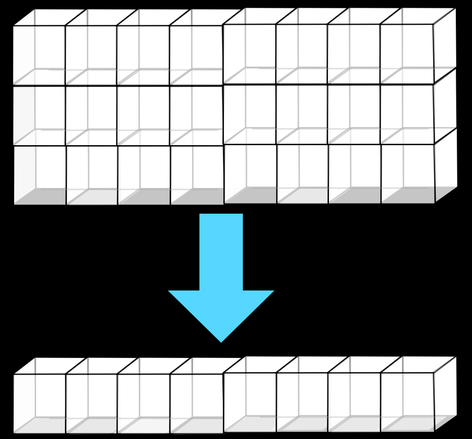
Higher-order functions (combinators) are a natural way to express these computations

Destructive updates often not necessary (or even helpful) for performance

Index based:

compiler has to figure out data dependencies and memory access patterns:

```
for (i = 0; i < N; i++) {
  brr[0] = 0;
  for (j = 0; j < M; j++) {
   brr[i] += arr[i][j];
  }
}
```

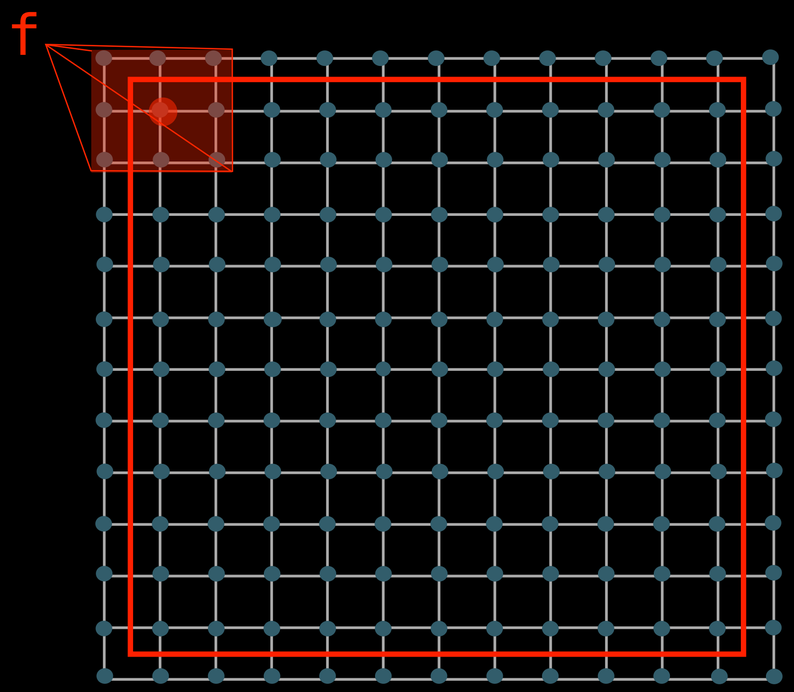


Combinator based:

access patterns, parallelism and dependencies often statically known:

```
brr = map sum arr
```

EXAMPLE: STENCIL COMPUTATIONS

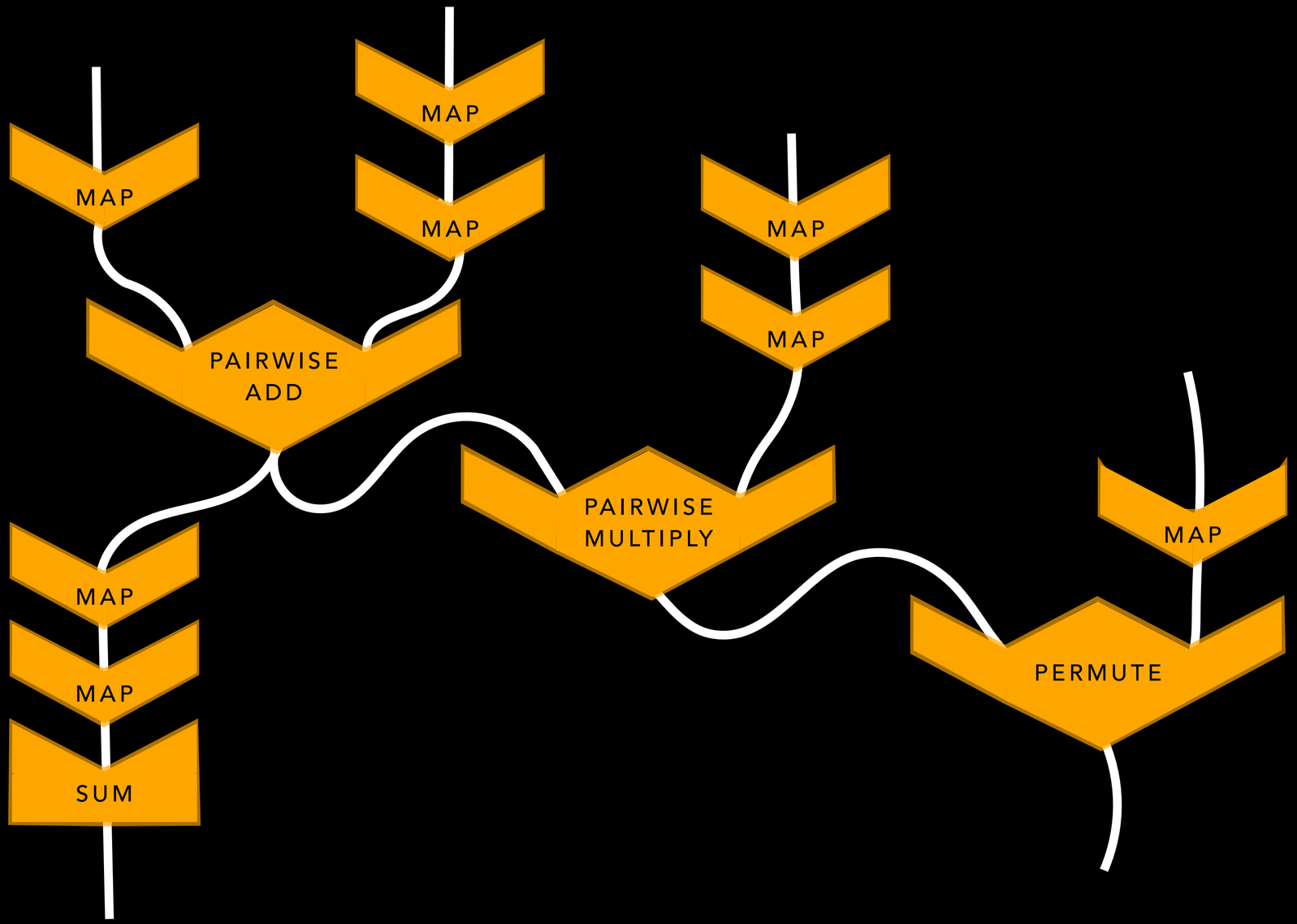



```
stencil 3x3 f clamp
```

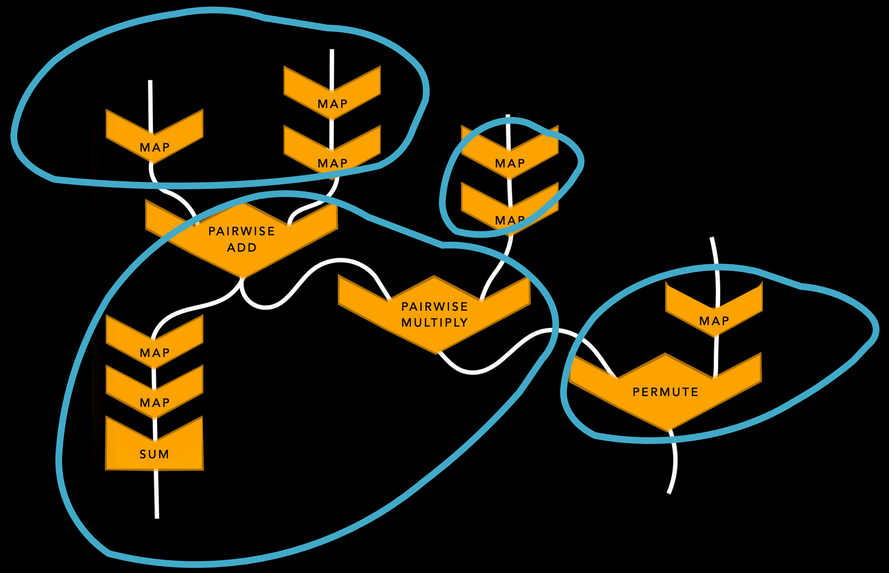
less code for the programmer to write

more information for the compiler

PROGRAMS BUILT FROM CONSTRUCTORS



NEED TO BE TRANSLATED TO LOOPS/KERNELS



finding the right clustering is a hard problem

```
template <int BLOCK_M_, int BLOCK_N_, int HEAD_DIM_, int NUM_WARPS>
__global__ __launch_bounds__(NUM_WARPS * WARP_SIZE, STREAMS_PER_SM)
void fmha_fwd_kernel(const __half* __restrict__ q,   // [B,H,M,D] row-major
                     const __half* __restrict__ k,   // [B,H,N,D] row-major
                     const __half* __restrict__ v,   // [B,H,N,D] row-major
                     float* __restrict__ o,          // [B,H,M,D] row-major
                     const float scale,
                     const int B, const int H,
                     const int M, const int N) {
    constexpr int SMEM_Q_BYTES = BLOCK_M_ * HEAD_DIM_ * sizeof(__half);
    constexpr int SMEM_K_BYTES = BLOCK_N_ * HEAD_DIM_ * sizeof(__half);
    constexpr int SMEM_V_BYTES = SMEM_K_BYTES;
    constexpr int SMEM_RED_BYTES = (BLOCK_M_ / 4) * sizeof(float);

                                                                  4_K_BYTES + SMEM_V_BYTES) + SMEM_RED_BYTES];
```

```
// ---- online softmax accumulators (one per mma row fragment) ----
float m_i[2] = { -INFINITY, -INFINITY };
float l_i[2] = { 0.f, 0.f };
float acc[2][4] = {};
(void)acc;

// ---- KV pipeline prologue ----------------------------------------
constexpr int KV_ELEMS_PER_STAGE = BLOCK_N_ * HEAD_DIM_;
const long kv_base = ((long)(b * H + h)) * N * HEAD_DIM_;
const int kv_threads = KV_ELEMS_PER_STAGE / 8;

#pragma unroll
for (int s = 0; s < STAGES; ++s) {
    const int n_row0 = s * BLOCK_N_;
    #pragma unroll 2
    for (int idx = tid; idx < kv_threads; idx += blockDim.x) {
        const int r = (idx * 8) / HEAD_DIM_;
        const int c = (idx * 8) % HEAD_DIM_;
        const int gr = n_row0 + r;
        if (gr < N) {
            cp_async_16(smem_k[s] + swizzle_smem_index(r, c, HEAD_DIM_),
                        k + kv_base + (long)gr * HEAD_DIM_ + c);
            cp_async_16(smem_v[s] + swizzle_smem_index(r, c, HEAD_DIM_),
                        v + kv_base + (long)gr * HEAD_DIM_ + c);
        }
    }
    cp_async_commit_group();
}
```

```
// ---- main loop over KV tiles ----------------------------------------
const int num_kv_tiles = (N + BLOCK_N_ - 1) / BLOCK_N_;
for (int t = 0; t < num_kv_tiles; ++t) {
    const int stage = t % STAGES;
    const int n_row0 = t * BLOCK_N_;

    if (t + STAGES < num_kv_tiles) {
        const int pre_row0 = (t + STAGES) * BLOCK_N_;
        #pragma unroll 2
        for (int idx = tid; idx < kv_threads; idx += blockDim.x) {
            const int r = (idx * 8) / HEAD_DIM_;
            const int c = (idx * 8) % HEAD_DIM_;
            const int gr = pre_row0 + r;
            if (gr < N) {
                const int ps = (t + STAGES) % STAGES;
                cp_async_16(smem_k[ps] + swizzle_smem_index(r, c, HEAD_DIM_),
                            k + kv_base + (long)gr * HEAD_DIM_ + c);
                cp_async_16(smem_v[ps] + swizzle_smem_index(r, c, HEAD_DIM_),
                            v + kv_base + (long)gr * HEAD_DIM_ + c);
            }
        }
        cp_async_commit_group();
    }
    cp_async_wait_group(STAGES - 1);
    __syncthreads();

    // ---- S = Q K^T via mma.sync, per 16x16 subtiles ----
    float s_frag[4];
    float m_new[2];
    #pragma unroll
    for (int kt = 0; kt < BLOCK_N_ / 16; ++kt) {
        mma_m16n8k16_frag(s_frag,
                          smem_q + (warp_id * 16) * HEAD_DIM_,
                          smem_k[stage] + kt * 16 * HEAD_DIM_,
                          s_frag /* dummy-C; real path uses zero-init */);
        #pragma unroll
        for (int e = 0; e < 4; ++e) s_frag[e] *= scale;
    }

    // ----
```

optimal solution may look very different than hand-written code!

EXAMPLE: SCANS

```
brr[0] = 0;
for (i = 1; i < N; i++) {
  brr[i] = brr[i-1] + arr[i-1];
}
```

- Implementing scans on GPUs efficiently is non trivial
- requires multiple loops, architecture specific

FUNCTIONAL ARRAY LIBRARIES/LANGUAGES

- Functional array languages:
  - APL
  - Futhark (https://futhark-lang.org/)
  - SaC (https://www.sac-home.org/start)
  - Accelerate (https://hackage.haskell.org/package/accelerate)
- Comprehensive survey showed
  - competitive performance compared to hand-optimised code
  - significantly easier to write
  - portable

FUTURE DIRECTIONS

- With Accelerate, we're working on
  - more backends (currently LLVM for multi-core, NVidia GPUs)
  - extending to heterogeneous systems, with automatic verification
    - combines concurrency with data-parallelism
    - automatic verification requires well-defined semantics