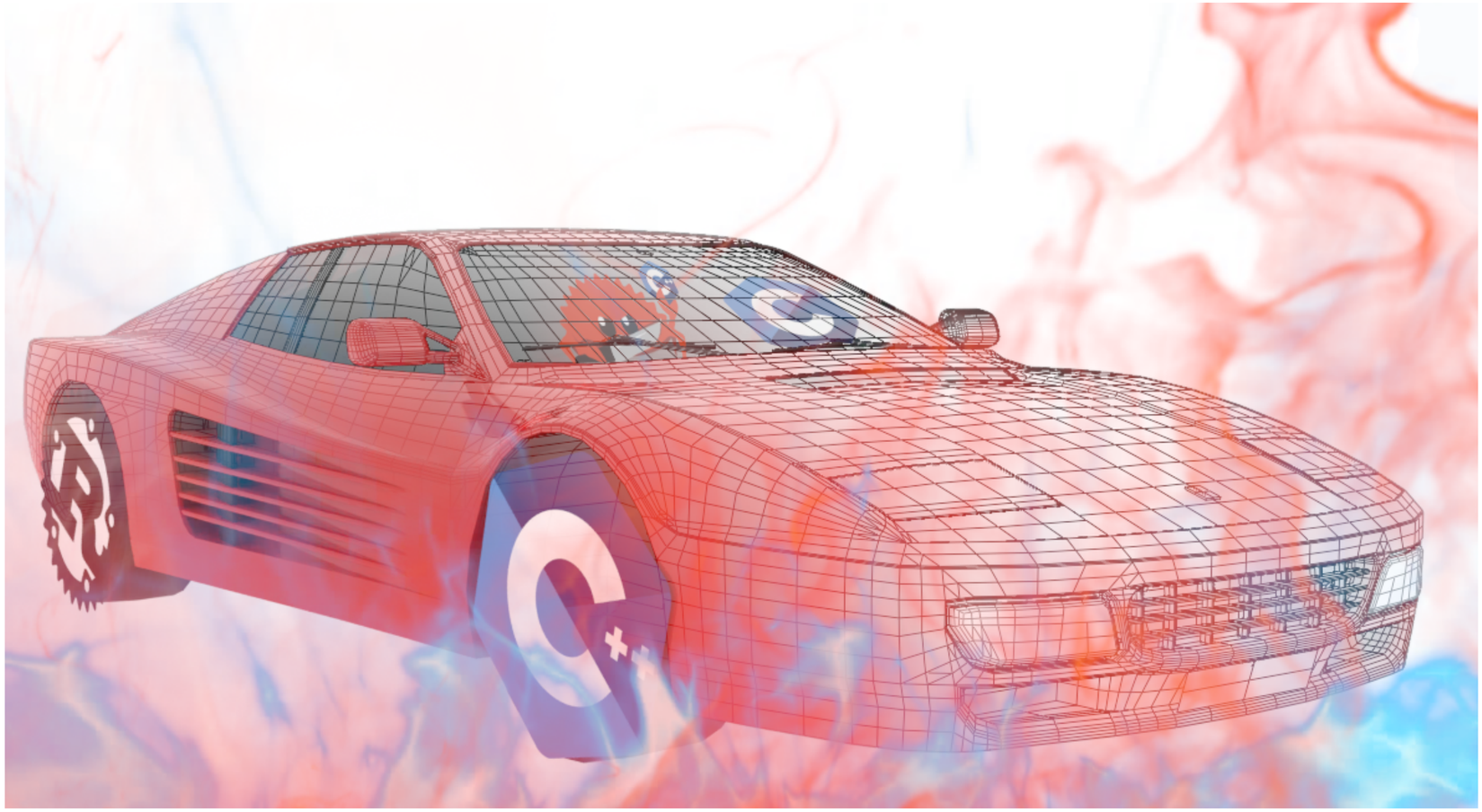
Effizientes Programmieren in C, C++ und Rust

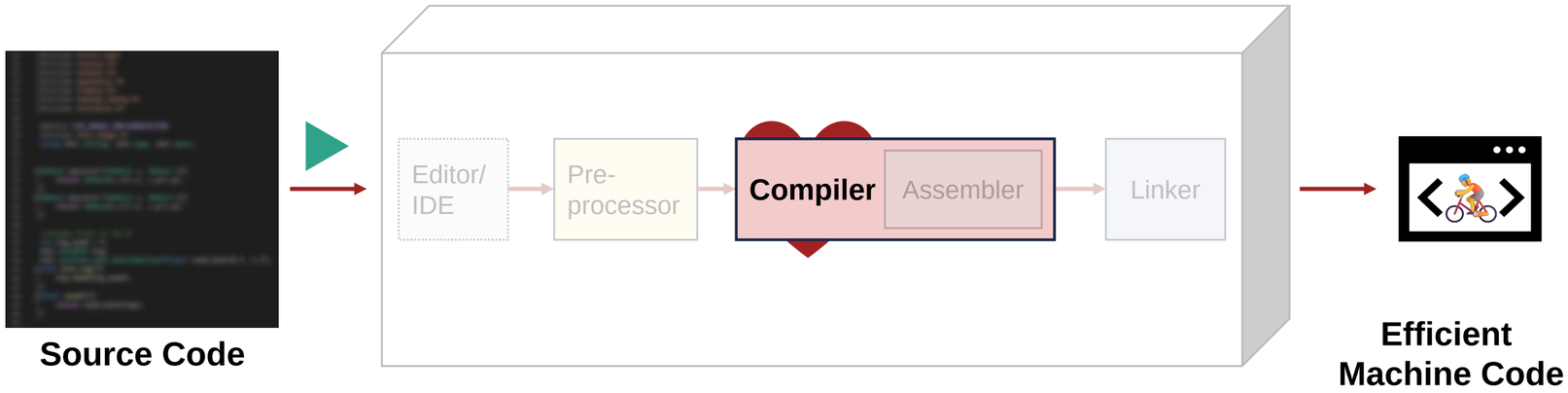
Compiler Optimizations and Undefined Behavior

Colin Bretl, Nikolai Maas, Peter Maucher | 24.10.2024



Peformance and the Compiler

- Our previous view of the compiler:



- **Claim:** Basic understanding of compiler optimizations is necessary to write truly efficient code
  - How does a compiler work?
  - Rough idea of important optimizations
  - Undefined behavior and abstract machine

Peformance and the Compiler

Two Extremes

- Write *everything by hand* (assembly?), don't rely on the compiler

Problems:

- Modern hardware/assembly is complicated with many architectures
- Compilers are better at instruction-level optimization than most people

- Write *as high-level as possible*, let the compiler do all optimizations

Problems:

- Compilers usually won't do large transformations
- The compiler has less information than you; *(at least without PGO)*

- Better: build intuition about optimizations that are reliable
- Stay high-level in general, verify efficiency of hot code and go low-level if needed
  - *Always* prefer a reasonable data layout and abstractions that are transparent to the compiler

Peformance and the Compiler

What do we expect of an optimizing compiler?

- Translation into "best possible" machine code — ???
- Elimination of redundant computations — reasonable (locally)
- Usage of information available at compile time — reasonable
  - *Constants, static type information, ...*
- Architecture-specific optimizations — reasonable
  - *Efficient ASM instructions, instruction reordering, vectorization, ...*
- Elimination of abstractions (e.g. function calls, classes) — reasonable
  - *Goal in C++ and Rust: zero-cost abstractions* 💸
- Usage of specific properties for "clever" optimizations — maybe
- Optimization of memory layout and allocations — maybe (locally)
- Selection of appropriate algorithm for each operation — problematic
- Automatic parallelization — problematic

Peformance and the Compiler

SURPRISINGLY POWERFUL?

```
uint32_t computeSum(uint32_t n) {
    uint32_t sum = 0;
    for (int j = 0; j <= n; ++j) {
        sum += j;
    }
    return sum;
}
```

(clang)

```
uint32_t computeSum(uint32_t n) {
    uint32_t sum = n - 1;
    sum *= n;
    sum >>= 2;
    sum += n;
    return sum;
}
```

SURPRISINGLY WEAK?

```
struct Complex {
    std::vector<int> v;
};

void consume(Complex c);

Complex fromComputation(int arg);
```

```
void passThrough() {
    consume(fromComputation(123));
}
```

```
void passThrough() {
    Complex x = fromComputation(123)
    consume(x);
    if ( !__is_empty(x.v) ) {
        __destruct(x.v);
    }
}
```

Peformance and the Compiler

SURPRISINGLY POWERFUL?

```
uint32_t computeSum(uint32_t n) {
    uint32_t sum = 0;
    for (int j = 0; j <= n; ++j) {
        sum += j;
    }
    return sum;
}
```

(clang)

```
uint32_t computeSum(uint32_t n) {
    uint32_t sum = n - 1;
    sum *= n;
    sum >>= 2;
    sum += n;
    return sum;
}
```

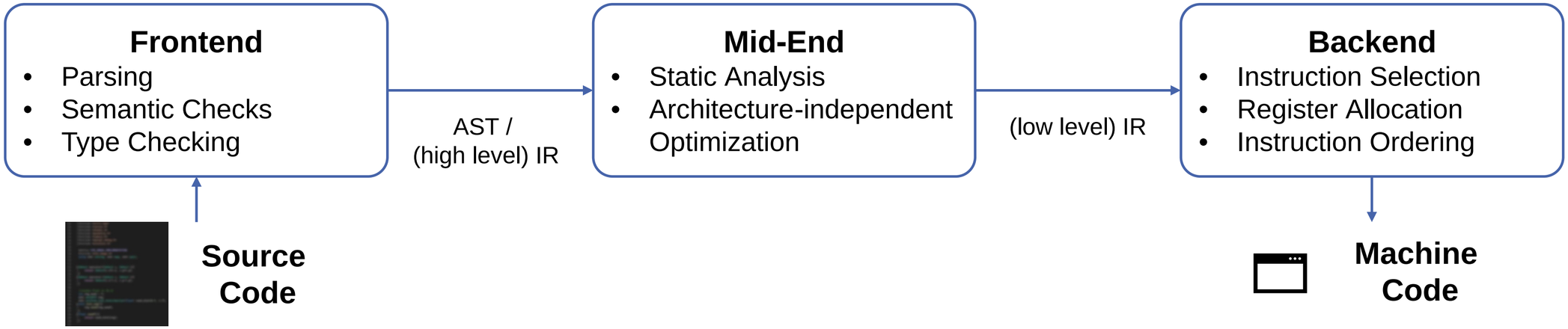
SURPRISINGLY WEAK?

```
struct Complex {
    std::vector<int> v;
};

void consume(Complex c);

Complex fromComputation(int arg);
```

```
void passThrough() {
    Complex x = fromComputation(123)
    consume(std::move(x));
}
```

```
void withTemporary() {
    Complex _slot1;
    Complex _slot2 =
        fromComputation(123);
    _slot1 = _slot2;
    _slot2 = 0;
    consume(_slot1);
    if ( !__is_empty(_slot1.v) ) {
        __destruct(_slot1.v);
    }
    if ( !__is_empty(_slot2.v) ) {
        __destruct(_slot2.v);
    }
}
```

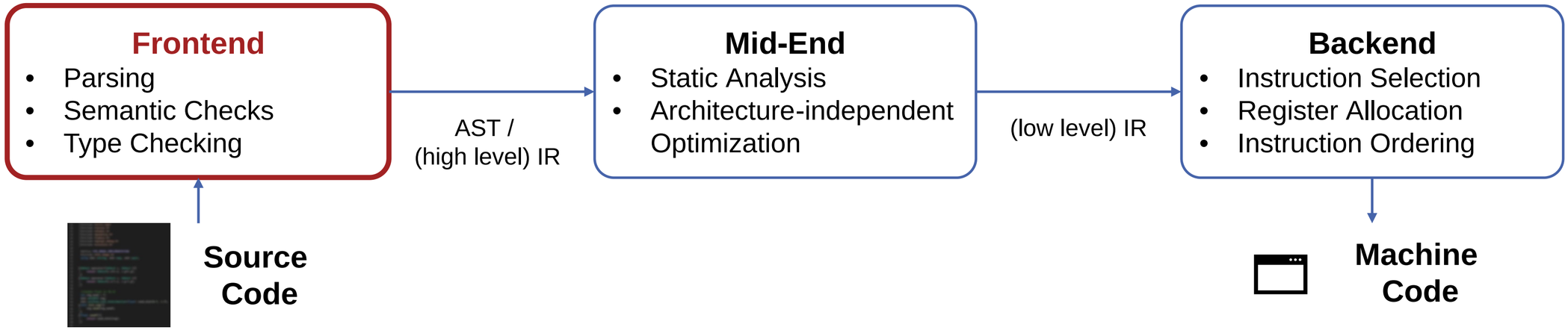
A Very Short Overview of Compiler Architecture



- Basic principle: apply series of transformation passes to the code
- Passes result in new representation or enhance the current representation

- AST: Abstract Syntax Tree
  - *Abstract tree-shaped representation of syntax (expressions, statements, ...)*
- IR: intermediate representation
  - *Compiler-internal language to represent code more efficiently*

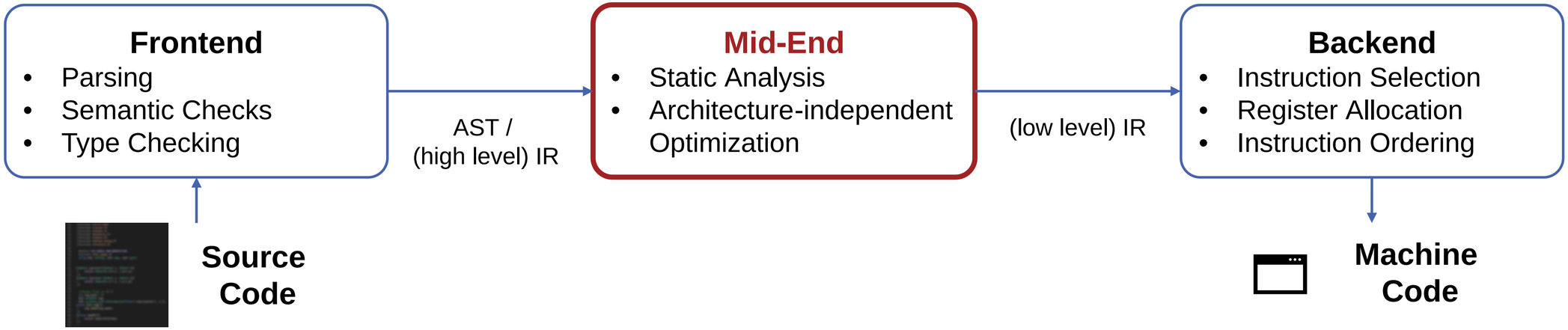
A Very Short Overview of Compiler Architecture



- Parsing: transform textual code into more efficient representation (AST)
- Check syntactic and semantic correctness...
- ...and provide useful error messages
  - *This is actually hard!*

- Frontend mostly irrelevant for optimization
  - Encode optimization-related information in IR (e.g. types)
  - Only in C++: monomorphization
  - Rust: monomorphization is delayed to the mid-end

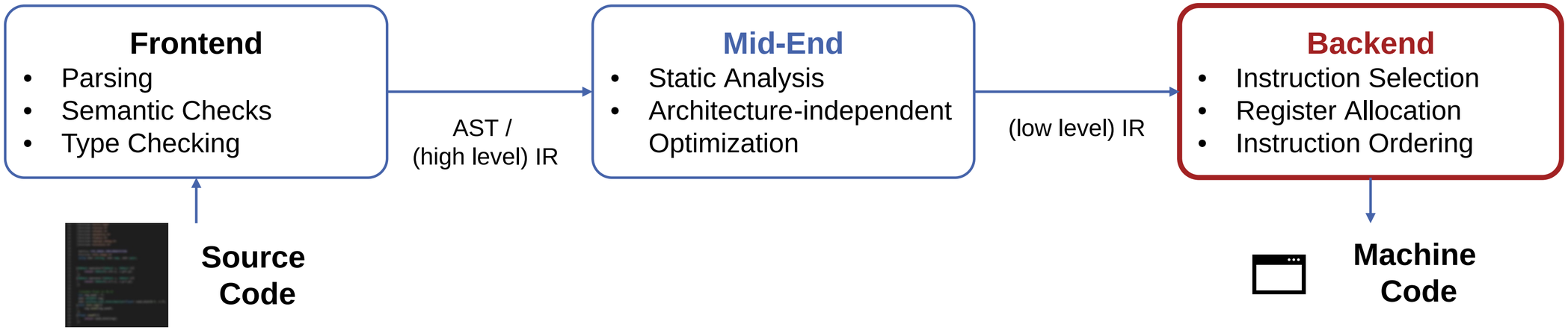
A Very Short Overview of Compiler Architecture



- Main task: optimization!
  - *Everything not architecture-dependent (e.g. all higher-level transformations)*
- Only in Rust: borrow checking and monomorphization

- Static analysis to determine properties useful for optimization (e.g. pureness) and, possibly, report warnings
  - *Can find errors like uninitialized variables, null-pointer dereferences and similar*

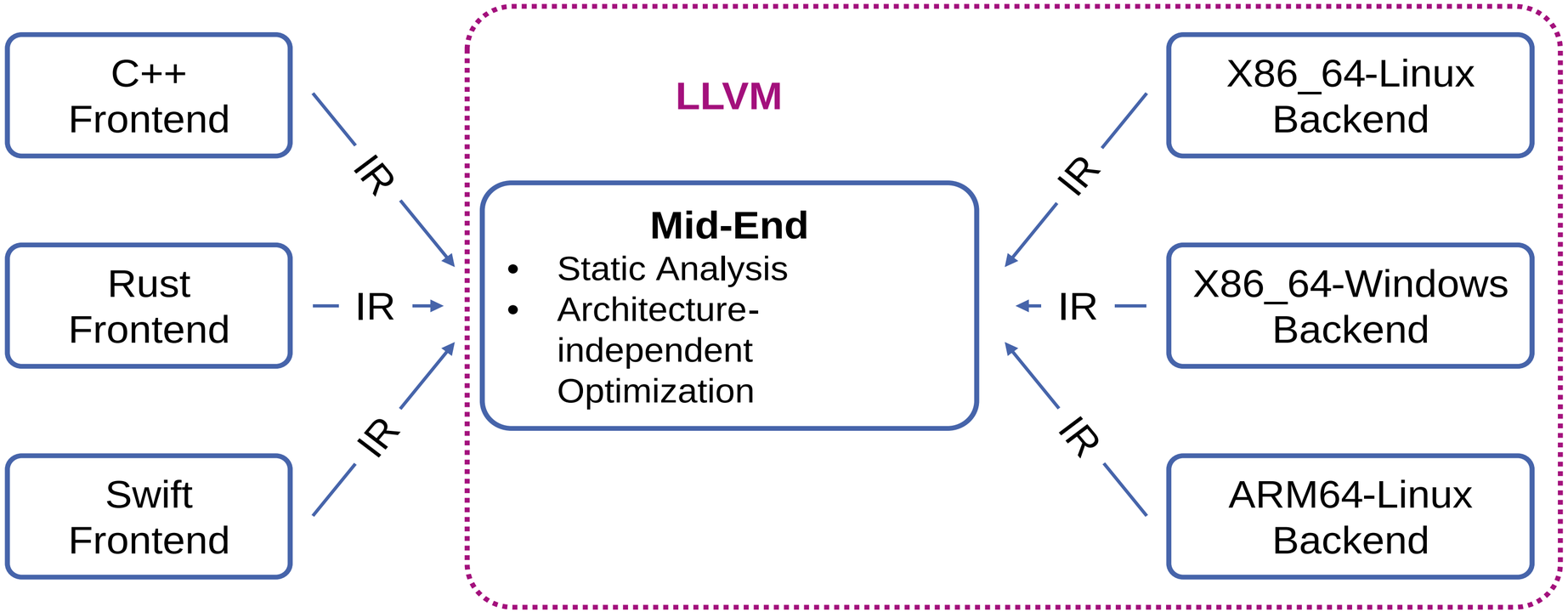
A Very Short Overview of Compiler Architecture



- Translate abstract code representation into concrete assembly code
- Heavily architecture-dependent
  - *Processor architecture affects available assembly instructions and registers*
  - *Operating system affects ABI as well as layout and setup of the executable*

- Select best available instructions (extreme example: SIMD)
- Keep data in registers instead of using the stack
- Minimize memory latencies via instruction ordering

A Very Short Overview of Compiler Architecture



- Modular architecture and intermediate representations allow reuse of mid-end
- Direct translation would require #languages × #architectures implementations

- **Example LLVM:** a reusable compiler backend (or rather, mid-end and backend) is main goal of the project

Basics of Optimization

What are optimizations allowed to do?

- C++ standard: as-if rule↖
  - Any optimization that does not change observable behavior↖ is allowed
- Additional circumstances where optimizations are allowed
  - Copy elision
  - Undefined behavior↗

What is observable behavior?

- Any kind of I/O
- Calling external functions
  (i.e. functions where the implementation is not known)
- Volatile reads and writes
  (irrelevant for us)

Basics of Optimization

Whole Program / Interprocedural Optimization (IPO)

- Works on the whole program at once (or at least multiple functions)
  - *Full access to any required information*
- In practice restricted to rather simple optimizations
  - *Any analysis with super-linear time is exceedingly expensive*
  - *Tracking the state of the whole program is hard*
- Includes inlining, dead code elimination and function analysis (e.g. pureness)

Local Optimization

- Restricted to a single function
  - *Note: In assembly code, almost everything is a function! (We have no classes here)*
- Simpler to implement and much faster
- Includes vast majority of practical optimizations

More on Local Optimization

- Everything outside the function is a blackbox
- The ABI (application binary interface) defines interactions with the outside world
  - *Specifically calling convention (i.e., how registers and stack are used for functions)*

- **Problem:** Any write to globals, heap memory or stack memory of callers might change observable behavior

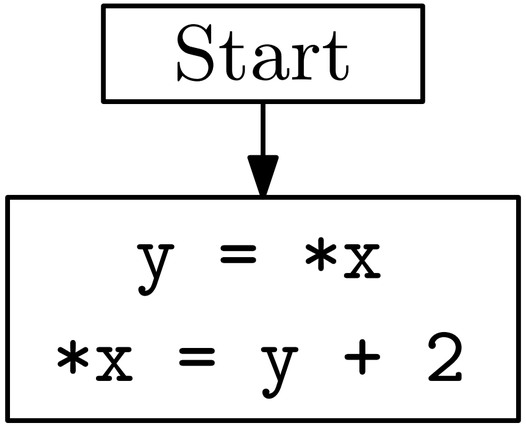
Side Effects

- Anything affecting observable behavior is called a side effect
  - *Also includes I/O, external function calls, ...*
- Pure means side effect free
- Side effects can not (easily) be removed or reordered
- Side effects usually depend on data and/or control flow

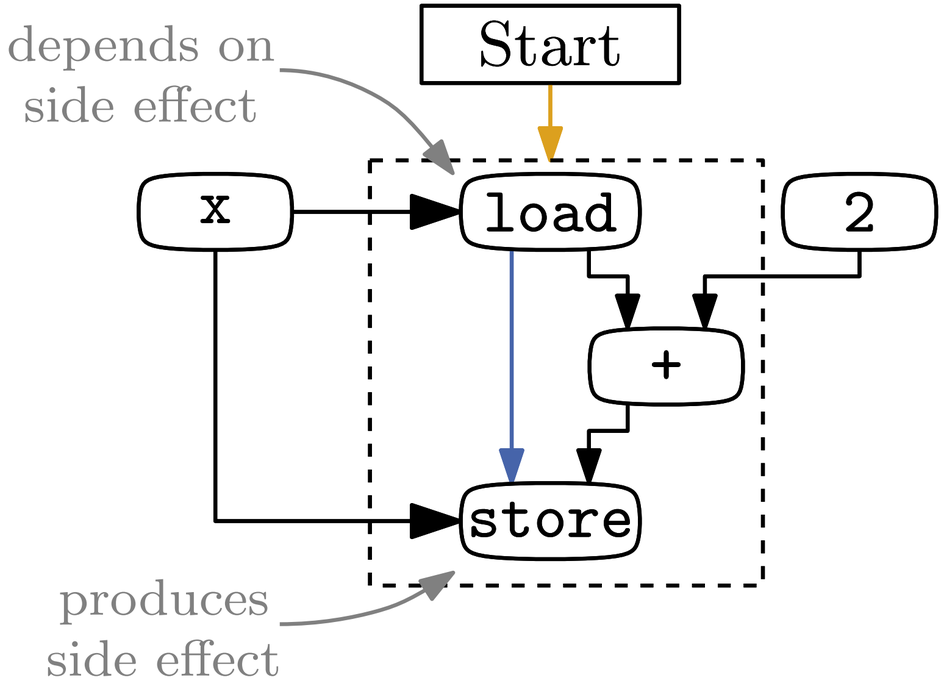
⇒ **Three restrictions:**

- Data dependencies
- Control flow dependencies
- Side effects

Control Flow Graph (CFG)



Abstracted CFG



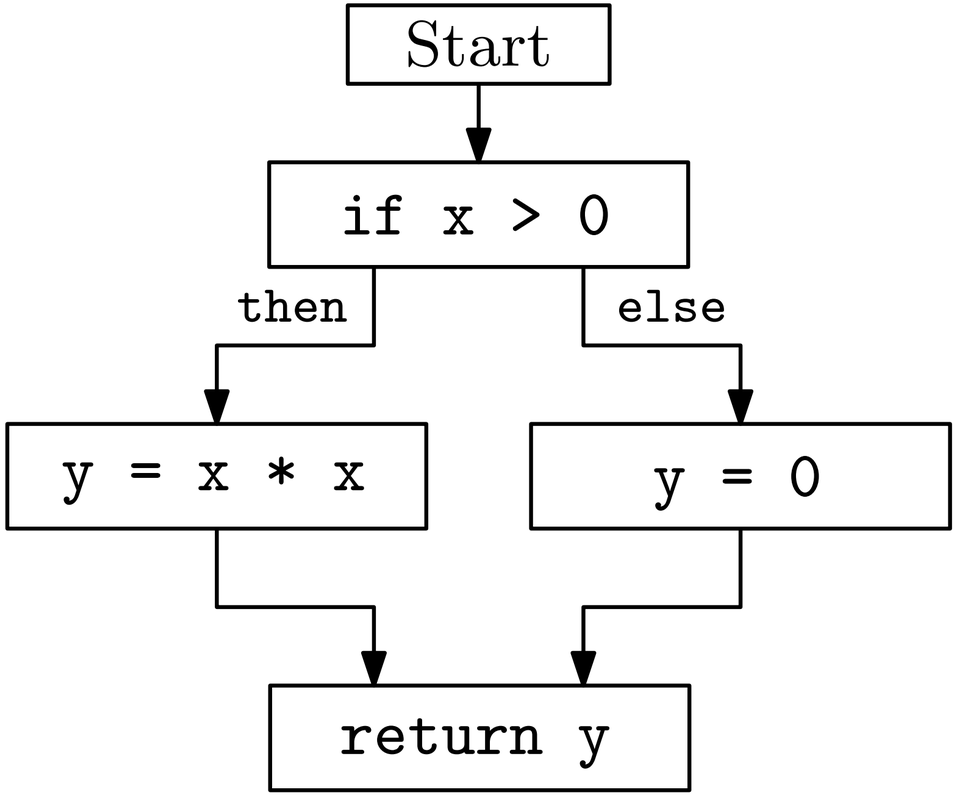
More Abstraction!

- We only need a **partial order*** (as implied by dependencies)
- Variables can be abstracted as an expression graph

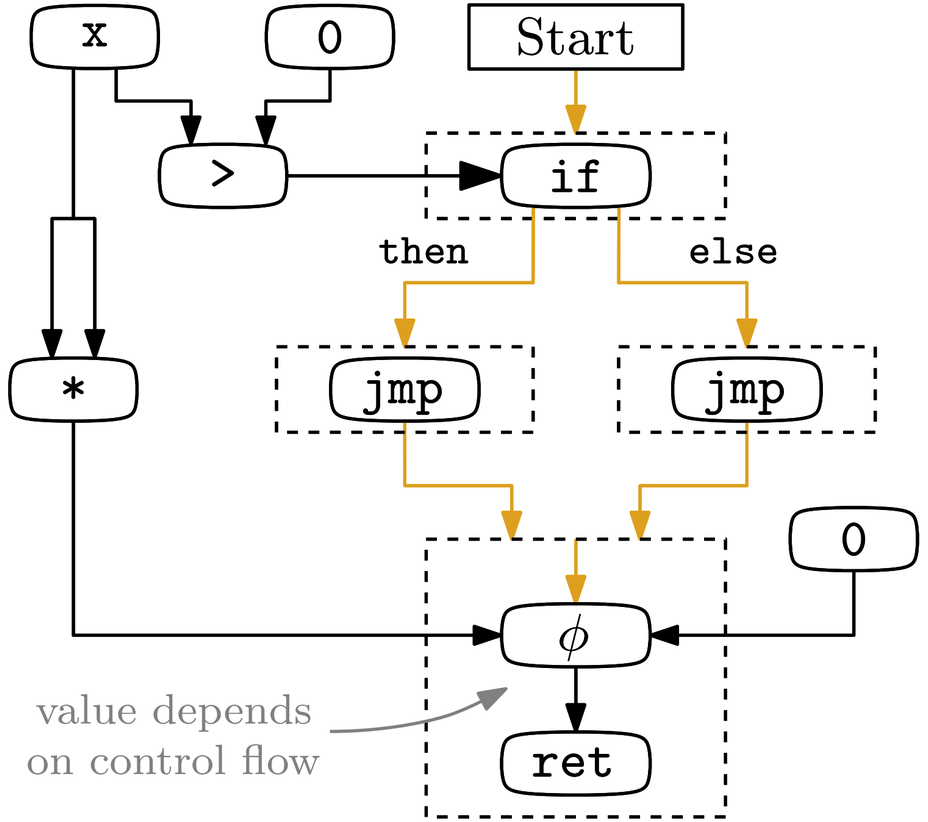
** not entirely an order, since loops create ... well, loops*

Basics of Optimization

Control Flow Graph (CFG)



Abstracted CFG*



** specifically, this representation uses SSA form*

Constant Propagation

- Replaces expressions with statically known value with their result

```
int32_t input = /*...*/ ;
size_t NUM_BITS = sizeof(int32_t * CHAR_BI
input = input >> (NUM_BITS / 2);
```

```
int32_t input = /*...*/ ;
input = input >> 16;
```

👍 Benefits

- Reduces computations necessary at runtime
- Enables elimination of branches and arithmetic optimizations

👎 Restrictions and Drawbacks

- Only for compile-time constants
- No drawbacks (?)

Dead Code Elimination

- Removes unreachable and dead (effectless) statements
- IPO part: removes functions that are never called

```
int unused = 10;  // no effect
if (false) {
    std::println("I'm dead"); // unreachab
}
unused = 42;
my_function(unused);
```

```
int unused = 42;
my_function(unused);
```

👍 Benefits

- Reduces binary size
- Removing branches might reveal more opportunities (constant propagation!)

👎 Restrictions and Drawbacks

- Only side-effect free statements can be dead
- No drawbacks (?)

Common Subexpression Elimination

- Deduplicates expressions that appear more than once

```
int input = /*...*/;
int a = 3 * (input + 5);
int b = (input + 5) / 2;
```

⇒

```
int input = /*...*/;
int common = input + 5;
int a = 3 * common;
int b = common / 2;
```

👍 Benefits

- Removes redundant computation

👎 Restrictions and Drawbacks

- Might need additional variable to store common value
  - Increases register pressure

⇒ Undo in backend ?

Loop Invariant Code Motion

- Pull redundant computations out of loops

```
int input = /*...*/;
for (int i = 0; i < n; ++i) {
    int t = 3 * input;
    my_function(i, t);
}
```

```
int input = /*...*/;
int t = 3 * input;
for (int i = 0; i < n; ++i) {
    my_function(i, t);
}
```

👍 Benefits

- Removes redundant computation
- $\mathcal{O}(n) \Rightarrow \mathcal{O}(1)$

👎 Restrictions and Drawbacks

- Pessimisation if loop is never entered!
- Expression must be loop invariant and side effect free

Loop Unrolling

- Pull redundant computations out of loops

```
int sum = 0;
for (int i = 0; i < 2 * n; ++i) {
    sum += array[i];
}
```

```
int sum = 0;
for (int i = 0; i < n; ++i) {
    sum += array[2 * i];
    sum += array[2 * i + 1];
}
```

👍 Benefits

- Check loop condition less often
- Enables instruction reordering
- Loops with constant number of iterations can be completely removed

👎 Restrictions and Drawbacks

- Can significantly increase binary size
- $m\times$ loop unrolling requires n multiple of m or special casing*

* *en.wikipedia.org/wiki/duff's_device*
(content warning: may cause brain damage)

Arithmetic Optimizations

- Exploit mathematical identities

```
int result = 10 + x - 1 + x;
```

```
int result = (x << 1) + 9;
```

```
uint32_t result = x * y;
result = result / x;
```

```
uint32_t result = y;
```

Overflow!

```
float result = 5.0 + x - 1.0;
```

```
float result = 4.0 + x;
```

Floats are not associative!

```
uint32_t x = y / 5;
```

```
uint64_t tmp = y * 3435973837;
uint32_t x = tmp >> 34;
```

Arithmetic Optimizations

- Exploit mathematical identities

```
int result = 10 + x - 1 + x;
```

⇒

```
int result = (x << 1) + 9;
```

```
uint32_t x = y / 5;
```

⇒

```
uint64_t tmp = y * 3435973837;
uint32_t x = tmp >> 34;
```

👍 Benefits

- Saves computation
- Replace expensive operations (division) with cheap ones

👎 Restrictions and Drawbacks

- Complicated by overflows and signed arithmetic
- Floats are not even associative...

⇒ `-ffast-math` for float optimizations that don't preserve semantic
(don't try this at home!)

Scalar Replacement of Aggregates

- Decompose `struct`/`class` into single members

```
struct Point { int x; int y; };
/* ... */
Point p {/* ... */};
int length = p.x * p.x + p.y * p.y;
```

```
int x = /* ... */;
int y = /* ... */;
int length = x * x + y * y;
```

👍 Benefits

- "Inlining" for data
- Use registers for operations on complex data types
- Simplifies further optimizations on the members

👎 Restrictions and Drawbacks

- Per default only possible for structs in the current stack frame
- Usually impossible at function call boundaries
  - *Except where part of the calling convention*

Load/Store Optimizations

- Remove unnecessary memory accesses

```
int x = *ptr;
*ptr = 3 * x;
// *ptr is not accessed
x = *ptr;
*ptr = x + y;
```

```
int x = *ptr;
// *ptr is not accessed
*ptr = 3 * x + y;
```

👍 Benefits

- Memory accesses are much more expensive than most other operations
- A lot of code is memory-bound
- ⇒ Can achieve significant speedups 🚴

👎 Restrictions and Drawbacks

- Must ensure there are no concurrent accesses to the memory location
- Non-local memory is mostly a blackbox

Inlining

- Replace function call with content of function

```
int square(int x) {
    return x * x;
}

int square_2d(int x, int y) {
    return square(x) + square(y);
}
```

```
int square(int x) {
    return x * x;
}

int square_2d(int x, int y) {
    return x * x + y * y;
}
```

- Similar effect to actual copy & paste
- Removes function call overhead
  - *(often the less important part)*
- $\Rightarrow$ The smaller the function, the greater the effect!

- Allows to apply further optimizations using the additional context
  - Important case: constant arguments
- Can increase binary size *(many call sites)*
- Can decrease binary size *(more efficient assembly)*

Inlining

The king of optimizations?

- Eliminates cost of small functions
- Primary enabler for almost all other optimizations
- Who needs IPO (interprocedural optimization) if we can just inline?
- Main workhorse for *eliminating abstraction overhead*
  - Refactoring into smaller functions has no performance penalty

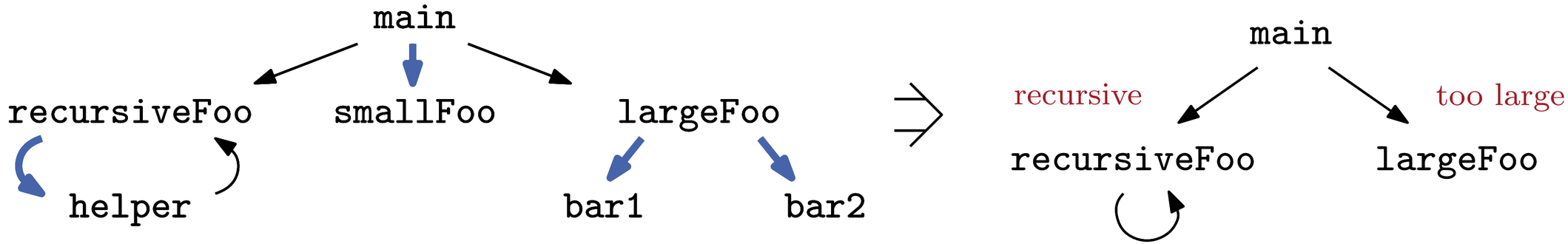
As the saying goes:

LLVM's inlining heuristic is "yes".

Criticism

- In theory, general IPO is more powerful than inlining
- Potential to massively increase binary size
- Inlining heuristics are fickle while having massive impact

Inlining



- Works on the call graph
- Simple strategy: bottom-up traversal
- More advanced strategy: evaluate and prioritize

Critical Aspect: Inlining Heuristic

- Decides whether a specific call is inlined
- Some criteria:
  - Callee size
  - Number of callers
  - Call site hot or cold?
  - Constant arguments?
  - Recursions?
  - Manual annotations?
  - "Yes"

Inlining

- Combining multiple functions greatly increases scope of local optimizations
- Pseudo-IPO: local optimizations get a more global view
- Exploit properties of the call site, e.g. constant parameters

- Extremely important in practice (zero cost abstractions!)
- Use manual annotations where (really) necessary!

  ⇒ `__attribute__((always_inline))`

  ⇒ `#[inline(always)]`

👍 Benefits

- Removes function call overhead
- Enables many other optimizations
- Might reduce binary size

👎 Restrictions and Drawbacks

- Often increases binary size
- Might increase register pressure
  (stress test for the register allocator!)
- Only possible for *static dispatch*
- Unreadable profiling and debug output

Code Generation - Instruction Selection

- Central step in backend: translate abstract representation into assembly
  - *For the moment, assume infinite registers*
- Use architecture-dependent efficient instructions! $\Rightarrow$ `-march=native`

EXAMPLE: `lea`

- *(= load effective address)*
- `x = x + 4*y + 10` translates to
  `lea @0, [@0 + 4*@1 + 10]`
- Supports adding two values, multiplying one with 2, 4 or 8 and adding a constant
- Address mode for index computation

EXAMPLE: `cmov`

- *(= conditional move)*
- Moves a value from one register to another if a condition is met (comparison-based)
- Allows to eliminate branches!

EXAMPLE: AUTO-VECTORIZATION

- Operate on multiple values at once (e.g. AVX-512 instructions operate on 512 bits)
- Massive potential speedup, but needs data "chunks", no branches, correct associativity

Code Generation - Instruction Ordering

- Optimize the order of the generated instructions
- Restricted by dependencies between instructions
- Hide latencies by initiating memory accesses as early as possible
- Can improve ILP (instruction level parallelism)

Code Generation - Register Allocation

- Previously: infinite virtual registers
- Register Allocation: map virtual registers to actual registers...
- ...or stack slots, if not enough registers are available

The Importance of Registers

- Despite an L1 cache-hit, accessing the stack might incur a 10x slowdown
  - Often needs a read and write!
  - Register access is virtually free

Restrictions

- Optimal register allocation is NP-hard
- At function call boundaries, calling convention must be obeyed
  - Registers are shared between functions

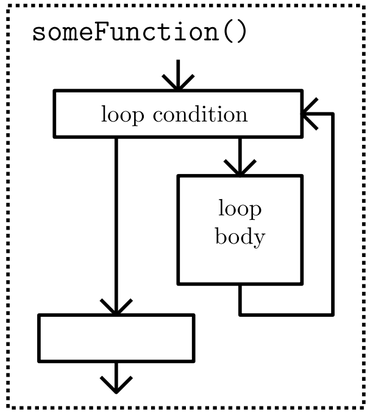
    $\Rightarrow$ must be saved!
  - Often, some of the arguments are passed on the stack

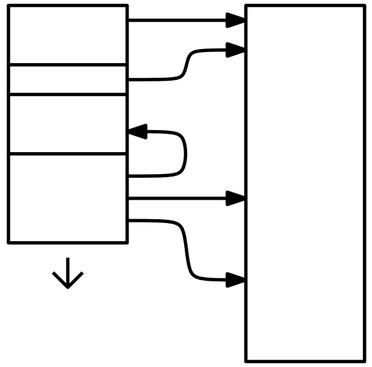
Understanding the Compiler

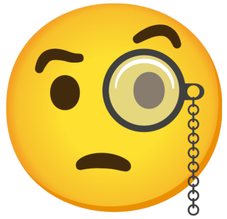
- **Goal:** understand capabilities and limitations of a typical AOT compiler

A High-Level Model of Compilers

- Two fundamental layers: **instructions** and **data**





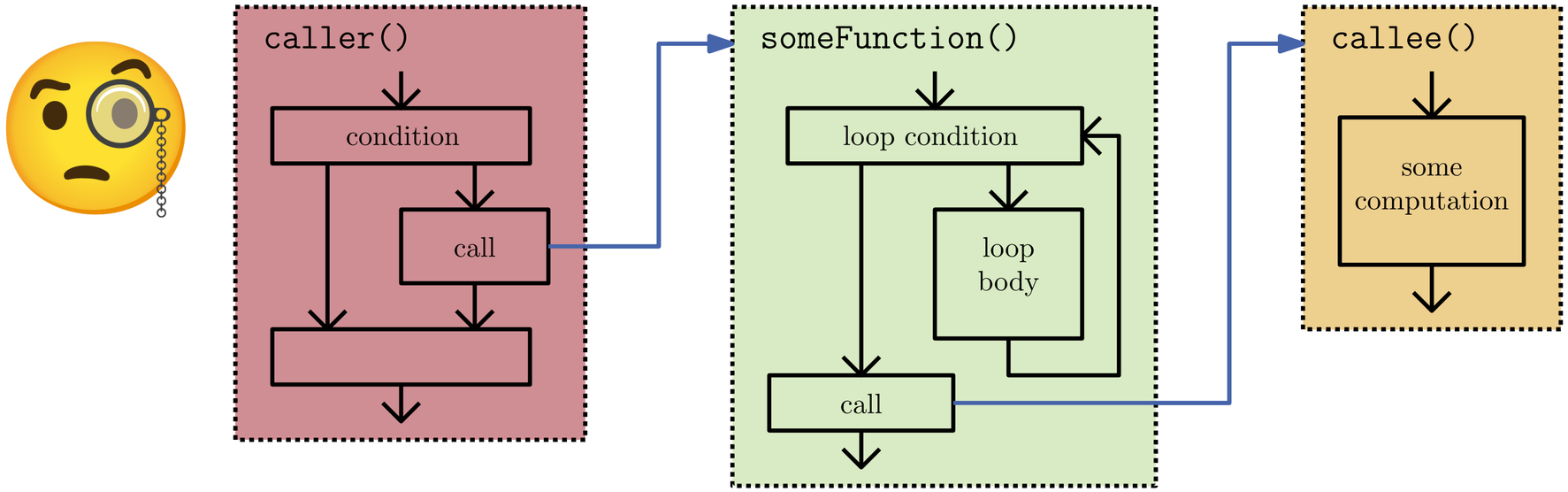


Compiler

- Compilers rely on local optimization
  - Compilers are "short-sighted":

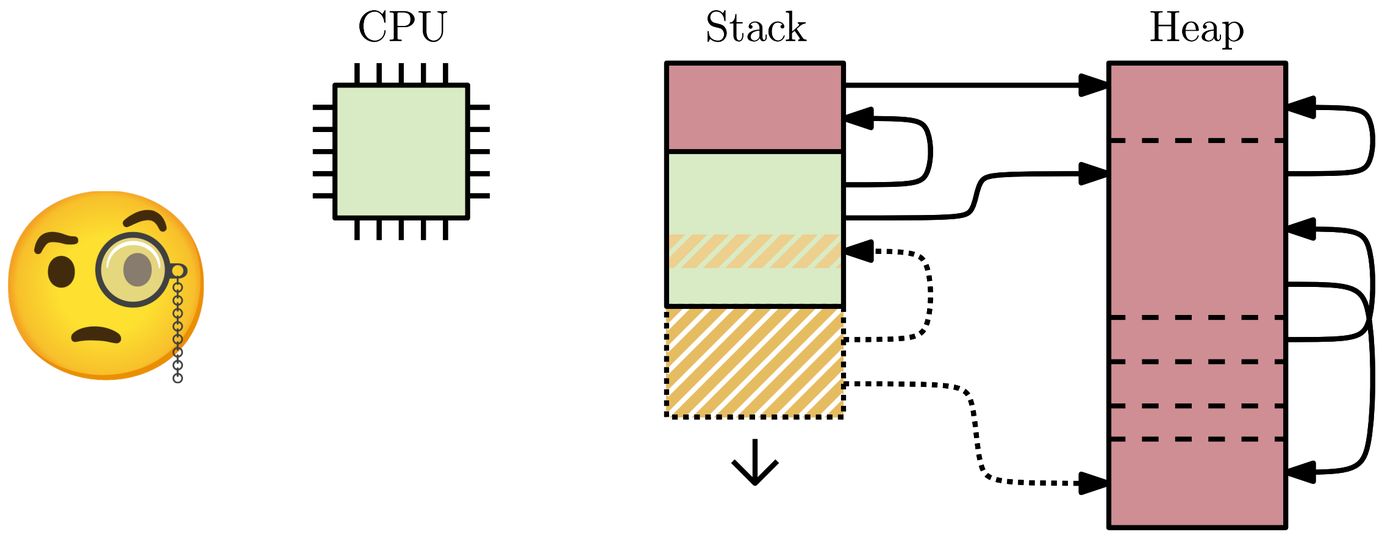
    Limited knowledge about both instructions and data
- Compilers must preserve semantics

The Instruction Layer



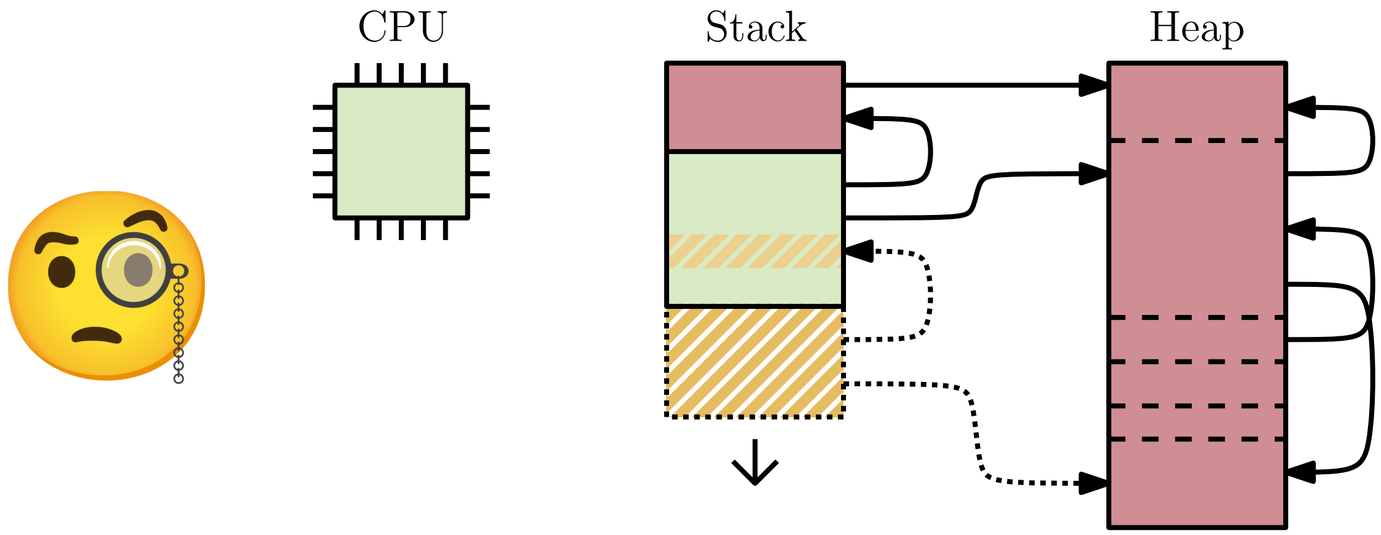
- Perfect knowledge about code of current function
- Caller is a complete blackbox
- Some information about callees (e.g. pureness, size, ...)
  - *Only in case of static dispatch*
- *Inlining* of callees merges functions and provides full information

The Data Layer



- Perfect knowledge about register-allocated values
- Good knowledge about stack frame of current function
  - *Callees might have pointers into the frame*
  - *Reasoning about arbitrary memory is harder than reasoning about numbers*
- As before: caller stack frames are a blackbox, some information about callee stack frames
- The heap is for the most part a big, black box
  - *Can be accessed by any part of the program*

The Data Layer



- Inlining merges stack frames
- Scalar replacement of aggregates transforms data to register-allocated values
  - *Restricted by call boundaries and aliasing*

What about the heap?

- The heap consists of allocations
  $\Rightarrow$ associate pointer with its allocation, try to track allocations
- Use the type of the pointees to differentiate between pointers

A Summary of Compiler Limitations

Compilers usually don't know...

- ... much about any memory that is indirectly referenced
- ... anything about properties of the input data *(see profile guided optimization)*
- ... invariants of your data types
- ... what the program is intended to do

Compilers usually won't...

- ... change anything about the **data layout**
- ... remove all your unnecessary memory accesses
- ... optimize across non-inlined function call boundaries
- ... optimize across compilation units without enabling LTO

$\Rightarrow$ Pick locations for manual optimization accordingly

Changing Observable Behavior

Is this a valid optimization?

```
int32_t addAndGet(Point* p,
                  const int32_t* val) {
    p->x += *val;
    p->y += *val;
    return *val;
}
```

```
struct Point { int32_t x; int32_t y; };
/* ... * /
int32_t addAndGet(Point* p,
                  const int32_t* val) {
    int tmp = *val;
    p->x += tmp;
    p->y += tmp;
    return tmp;
}
```

No!

```
Point p { 1, 2 };
int z = addAndGet(&p, &p.x);

// z = 2, p.y = 4 vs. z = 1, p.y = 3
```

- Compilers won't optimize this
- Aliasing is a problem not only for safety, but also performance!

Changing Observable Behavior

Let's try again!

```
int64_t addAndGet(Point* p,
                  const int64_t* val) {
    p->x += *val;
    p->y += *val;
    return *val;
}
```

⇒

```
struct Point { int32_t x; int32_t y; };
/* ... * /
int64_t addAndGet(Point* p,
                  const int64_t* val) {
    int tmp = *val;
    p->x += tmp;
    p->y += tmp;
    return tmp;
}
```

But we could do ugly things

```
Point p { 1, 2 };
int z = addAndGet(&p, (int64_t*)&p);

// ???
```

Actually:

- Compilers gladly optimize this code

  ⇒ "Counterexample" has different output in debug and release mode
- But this changes observable behavior?

  ⇒ Yes, but this is okay. Because of **undefined behavior!**

Undefined Behavior (UB)

An Origin Story

Once a upon a time, the proud tribe of C++ programmers lived by its code, which was impure yet powerful, expressive and efficient at the same time.

And their life would have been happy if it wasn't for their archenemy, the ancient tribe of assembly programmers.

Though of lesser power, the assembly programmers never ceased to mock and taunt the C++ tribe, since the efficiency of the C++ code was not perfect yet.

Facing this plight, the leaders of the C++ tribe (known as the standard committee) foresaw that salvation could be found only by a single means:
A pact with the mythical beings of the Great Temple of Compiler Technology...

Undefined Behavior (UB)

Oh dark goddess of performance, **what sacrifice** do you require of us
so that we match the efficiency of these filthy assembly programmers?

Know that such great gift is not given easily. I shall conjure a long list of seemingly **arbitrary rules** and you shall be bound to these rules by blood and oath. And if you faithfully abide the rules without the tiniest of flaws, I shall grant to you the divine efficiency of the super-computer.

Oh dark goddess, what if we commit a sin against one of the rules?

If you violate only **a single letter** of the rules, know that your soul shall be mine forever and all your works shall be burnt to the ground and thrown into the abyss of eternal darkness.

Undefined Behavior (UB)

- Undefined behavior is an assumption the compiler might use for optimization
  - *Example: pointers point to an object of the according type*

A More Precise Definition of UB

- The standard defines certain operations as undefined
- Let E be a specific execution of a program P
- If E contains an undefined operation x, E may behave in any arbitrary way

Remarks

- Undefined behavior is a property of E and not of P
- Common usage: "the program P exhibits UB"
  $\Rightarrow$ Usually means "some executions of P contain UB"
- The behavior of E is undefined even before x is executed

Undefined Behavior (UB)

From the C++ Standard:

> "However, if any such execution contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation)."

https://port70.net/%7Ensz/c/c%2B%2B/c%2B%2B11_n3337.txt

Undefined Behavior (UB)

Example: Signed Integer Overflow

```
int redundant(int x, int y) {
    return y * x / y;
}
```

```
int redundant(int x, int y) {
    return x;
}
```

```
int saturatingInc(int x) {
    // check for overflow
    if (x >= 0 && x + 1 <= 0) {
        return x;
    }
    return x + 1;
}
```

```
int saturatingInc(int x) {
    return x + 1;
}
```

- Are these optimizations reasonable?
- Both are allowed for signed integers due to UB and not allowed for unsigned integers

Undefined Behavior (UB)

Example: Time Traveling

```
int* ptr = /* ... */;
if (ptr != nullptr) {
    performSomeAction(ptr);
} else {
    logError();
    printf("Erronous value: %d", *ptr);
}
```

```
int* ptr = /* ... */;
performSomeAction(ptr);
```

- Null pointer dereference in second branch introduces UB
- Result of optimization: logError() is never called

  $\Rightarrow$ UB might change behavior before the offending operation
- Not actually done by gcc/clang, but legal

Undefined Behavior (UB)

Example: Wiping your hard drive

```c
static int (*fn_ptr)();

int eraseAll() {
    return system("rm -rf /");
}

void neverCalled() {
    fn_ptr = eraseAll;
}

int main() {
    return fn_ptr();
}
```

```c
int main() {
    return system("rm -rf /");
}
```

- UB can have arbitrary effects (here: resurrecting unreachable code)
- Nasal demons:

> "[with UB] demons may fly out of your nose"

Undefined Behavior (UB)

Further Examples (C++)

- Out-of-bounds access
- Use-after-free
- Operating on uninitialized data
- Modifying a constant or string literal
- Creating an enum from an invalid integer value
- Data races
- Modification of a scalar via a side-effect with indeterminate order relative to another access*
- ...

** that this one holds even for single-threaded code is actually insane: it makes `f(i, ++i)` UB (compilers luckily don't seem to exploit it)*

The Abstract Machine

If E contains an undefined operation x, then E may behave in any arbitrary way.

- Our definition is still not very precise
- How to define which operations are undefined?
  - *Programs run on actual hardware (producing different results)*
  - *Whether an operation is undefined can depend in complicated ways on its input*

$\Rightarrow$ We need abstract but precise terms for these definitions

Solution:

- The Standard uses the abstract machine to define correct program execution
  - *Think of an imaginary virtual machine for executing C++*
- A compiled program must have the same observable behavior as if running it on the abstract machine

$\Rightarrow$ Regarding correctness, programs are written for the abstract machine and not actual hardware

The Abstract Machine

- C++ has as whole zoo of "not-clearly-defined" behavior

IMPLEMENTATION-DEFINED BEHAVIOR

- Depends on the compiler
  (but the behavior is well-defined)
  - *"Parameter" of the abstract machine*
  - *Example: size of basic integer types*
- Must by documented by the compiler

UNSPECIFIED BEHAVIOR

- The abstract machine permits a set of acceptable behavior
  - *⇒ Abstract machine is non-deterministic with regard to these possible behaviors*
  - *Example: evaluation order of function arguments*

UNDEFINED BEHAVIOR

- Observable behavior may diverge from the abstract machine

ILL-FORMED, NO DIAGNOSTIC REQUIRED

- Basically undefined behavior for the whole program (not only specific executions)

The Abstract Machine

Memory (C++)

- No assumptions about memory/pointer layout (e.g. no mention of stack or heap)
- Memory contains objects represented as contiguous byte sequence
  - Objects have an associated address, unique except for first member
  - Each object has an (implicit) lifetime
- Bytes can have $2^{N\_BITS} + 1$ values:
  $\{0, \ldots, 2^{N\_BITS} - 1\} \cup \{\text{indeterminate}\}$
  (corresponds to uninitialized)
- Writing to indeterminate memory is okay, reading is almost always UB

Implications

- Memory is not just data
  (in the astract machine)
- Object lifetimes and locations, indeterminate values

Comparison to Rust

- Lifetimes have explicit types
- No unique address
- Actually, no concept of objects at all
  (instead, borrows and allocations have lifetimes)
- Borrow stack↗

Aliasing and Pointer Provenance

- If memory is not "just memory", what about pointers?

The Optimization Perspective

- We want to enable load/store optimizations
  - *This is actually the whole motivation for the "object" concept in the abstract machine*
- Requires to prove that pointers are distinct (no aliasing)

⇒ Use "reasonable" assumptions:

- Pointers don't leave their allocation
- Pointers with different types don't alias (exception: `void*` and `char*`)
- Objects are initialized before use

```
int64_t addAndGet(Point* p,
                  const int64_t* val) {
    p->x += *val;
    p->y += *val;
    return *val;
}
```

```
int64_t addAndGet(Point* p,
                  const int64_t* val) {
    int tmp = *val;
    p->x += tmp;
    p->y += tmp;
    return tmp;
}
```

Aliasing and Pointer Provenance

- Pointers don't leave their allocation*
- Pointers with different types don't alias
- Objects are initialized before use

Pointers have Provenance!

- These assumptions need precise semantics
- Each pointer has "ghostly" runtime metadata that doesn't actually exist (= provenance)
  - C++: allocation, type
  - Rust: allocation, mutability, borrow tag

⇒ Operations not permitted by a pointers provenance usually cause UB (e.g. pointer arithmetic, cast + access)

** yes, compilers special case `malloc` and `free` to track allocations*

What does this mean?

1. For the type system
2. For the hardware
3. For the compiler/abstract machine

Undefined Behavior: Blessing or Curse?

👎 The Case Against UB 👎

- Historically, UB was like unspecified behavior: allowing simpler compiler implementations for different architectures via diverging behavior*
  - *Compilers used only simple optimizations and the translation to assembly was "obvious"*
- Now, behavior is mostly standardized and optimizations are more powerful

  $\Rightarrow$ Meaning of UB shifted over time
- UB often introduces bugs although the programmers intent is clearly different

- Claim 1: Allowing crazy optimizations was not the original intent of the standard.
- Claim 2: The bugs introduced by UB outweigh the performance benefits.
- ***Bonus:*** *UB makes manual optimizations harder and thus is even detrimental to performance.*

** how true is this? I can't find a better source than Wikipedia*

Undefined Behavior: Blessing or Curse?

👍 The Case For UB 👍

- The original intent of the standard might not be entirely clear...
- ...but the current intent is!
- Removing redundant memory accesses is virtually impossible without UB
- Not all abstractions translate well to mordern hardware

  ⇒ Requires complicated optimizations to generate efficient machine code
- Writing "optimal" code with a less optimizing compiler ranges from cumbersome to near-impossible
  - *Remark: even modern compilers produce sub-optimal code in many cases*

- Answer 1: Allowing optimizations is definitively the current intent of the standard.
- Answer 2: For modern platforms, UB is necessary to achieve optimal performance.

Undefined Behavior: Blessing or Curse?

👎 The Case Against UB 👎

- Allowing crazy optimizations was not the original intent of the standard.
- The bugs introduced by UB outweigh the performance benefits.

👍 The Case For UB 👍

- Allowing optimizations is definitively the current intent of the standard.
- For modern platforms, UB is necessary to achieve optimal performance.

Conclusion

- Trade-off between performance and vulnerability to bugs (correctness)
- C++ highly prioritizes performance
  - Only trumped by backwards compatibility

Summary

Optimizations

- Understanding compilers is important for knowing where to optimize
- Main restriction: observable behavior

  $\Rightarrow$ Optimizations are typically local and don't change data layout
- Static dispatch and less indirection help
- Code can be tweaked to unlock optimizations
  - *Example: power of two step-width in loops allows unrolling and maybe auto-vectorization*
- Inlining is extremely important as abstraction eliminator

Undefined Behavior

- UB is an assumption made to allow optimizations
- UB is a necessary evil to achieve near-optimal performance
- C++ code runs on the abstract machine *(except it doesn't)*
- Pointers are not integers!