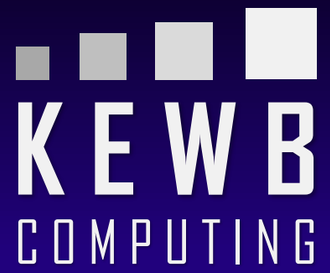
Back to Basics: The Abstract Machine

Bob Steagall

CppCon 2020



Overview/Goals

KEWB COMPUTING

- Describe abstract machines in general
- Describe the C++ abstract machine specifically
  - Language goals that drive its design
  - Role in program development and execution
  - Important definitions and characteristics
  - Important components of the abstract machine, and their relationships
- Provide a useful overview of the C++ abstract machine

What is an Abstract Machine?

"Abstract machines are machines *because they permit step-by-step execution of programs; they are* abstract *because they omit the many details of real (hardware) machines.*

Abstract machines provide an intermediate language stage for compilation. They bridge the gap between the high level of a programming language and the low level of a real machine. *The instructions of an abstract machine are tailored to the particular operations required to implement operations of a specific source language or class of languages."* (my emphasis)

- Stephan Diehl, Pieter Hartel, Peter Sestoft
Future Generation Computer Systems 16 (2000)

What is an Abstract Machine?

KEWB COMPUTING

*"**Programming language specifications** (not just C and C++, all high-level programming language specifications), **define the languages in terms of an abstract machine, which, in this usage, is the simplest imaginary computer capable of executing a program in the source language** (or a family of languages, as in the case of the JVM)."* (my emphasis)

- Sergey Zubkow, www.quora.com (2015)

What is an Abstract Machine?

KEWB COMPUTING

*"**A processor design which is not intended to be implemented as hardware, but which is the notional executor of a particular intermediate language (abstract machine language) used in a compiler or interpreter.** An abstract machine has an instruction set, a register set and a model of memory. It may provide instructions which are closer to the language being compiled than any physical computer or it may be used to make the language implementation easier to port to other platforms."* (my emphasis)

- Free Online Dictionary of Computing

What is the C++ Abstract Machine?

“The C++ abstract machine is a portable abstraction of your operating system, kernel and hardware.

The abstract machine is the intermediary between your C++ program and the system that it is run on.”

- Bryce Adelstein Lelbach, Core C++ 2019
The C++ Execution Model

Abstract Machines - Rationale

- We want to accomplish complex information processing tasks
- We want to write programs (in some human-readable language) to help us accomplish these complex tasks
  - Human-readable, for productivity
- We want tools that robustly and reliably transform our programs into a form that can execute on one or more computing systems
  - In C++, these tools are primarily the compiler and the linker

Abstract Machines - Rationale

- Wide variety of computing platforms
  - General-purpose CPUs
    - Intel, AMD, ARM, MIPS, Power, SPARC, Alpha, etc.
  - GPUs and specialized co-processors
    - NVidia, AMD, Intel, etc.
  - Embedded processors
    - TI, Atmel, Intel, Freescale, FPGAs, etc.
- Simply impractical (impossible?) to create a separate language and set of tools for every computing platform

Abstract Machines - Rationale

- We want tools to help us manage complexity
- When we write our programs, we (mostly) don't want to be concerned with the details of the underlying physical hardware
- We want programming languages capable of representing abstractions relevant to the problem domain at hand
- We want programming languages that transform our programs to executable form on one or more computing platforms

Abstract Machines - Rationale

- To solve problems like this, we humans like to use **abstraction**

- And so we define programming languages in terms of **abstract machines**
  - C, C++, Java, Python, Perl, Awk, Algol, and many more…

- What drives the design of the C++ abstract machine?

Language Goals

- C++ should support: (from wg21.link/p2137r0)
  - Performance-critical software
  - Software and language evolution
  - Code that is simple and easy to read, understand, and write
  - Practical safety guarantees and testing mechanism
  - Fast and scalable development
  - Current hardware architectures, OS platforms, and environments as they evolve

Language Goals

- C++ should support: (from wg21.link/p2137r0)
  - **Performance-critical software**
    - **Give the programmer control over every aspect of performance**
  - Software and language evolution
  - Code that is simple and easy to read, understand, and write
  - Practical safety guarantees and testing mechanism
  - Fast and scalable development
  - Current hardware architectures, OS platforms, and environments as they evolve

Language Goals

- C++ should support: (from wg21.link/p2137r0)
  - **Performance-critical software**
    - **Give the programmer control over every aspect of performance**
    - **Code should perform predictably**
  - Software and language evolution
  - Code that is simple and easy to read, understand, and write
  - Practical safety guarantees and testing mechanism
  - Fast and scalable development
  - Current hardware architectures, OS platforms, and environments as they evolve

Language Goals

- C++ should support: (from wg21.link/p2137r0)
  - **Performance-critical software**
    - **Give the programmer control over every aspect of performance**
    - **Code should perform predictably**
    - **Leave no room for a lower-level language**
  - Software and language evolution
  - Code that is simple and easy to read, understand, and write
  - Practical safety guarantees and testing mechanism
  - Fast and scalable development
  - Current hardware architectures, OS platforms, and environments as they evolve

Language Goals

KEWB COMPUTING

- C++ should support: (from wg21.link/p2137r0)
  - **Performance-critical software**
    - **Give the programmer control over every aspect of performance**
    - **Code should perform predictably**
    - **Leave no room for a lower-level language**
  - Software and language evolution
  - Code that is simple and easy to read, understand, and write
  - Practical safety guarantees and testing mechanism
  - Fast and scalable development
  - **Current hardware architectures, OS platforms, and environments as they evolve**

How Does C++ Meet These Goals?*

KEWB COMPUTING

(*for this discussion, the performance- and hardware-related goals)

- C++ has no layers of abstraction or execution between it and the hardware
  - No interpreters or virtual machines
  - Allows high-quality code generation by compiler back end
  - There is no room for another language between C++ and the hardware
- C++ maps types and operations directly onto hardware
  - Fundamental C++ types (e.g., char, int, double) map directly into memory entities
  - C++ pointers, references, arrays map directly to hardware addressing capabilities
  - Modern hardware supports useful arithmetic/logical operations on those entities

How Allows C++ to Meet These Goals?*

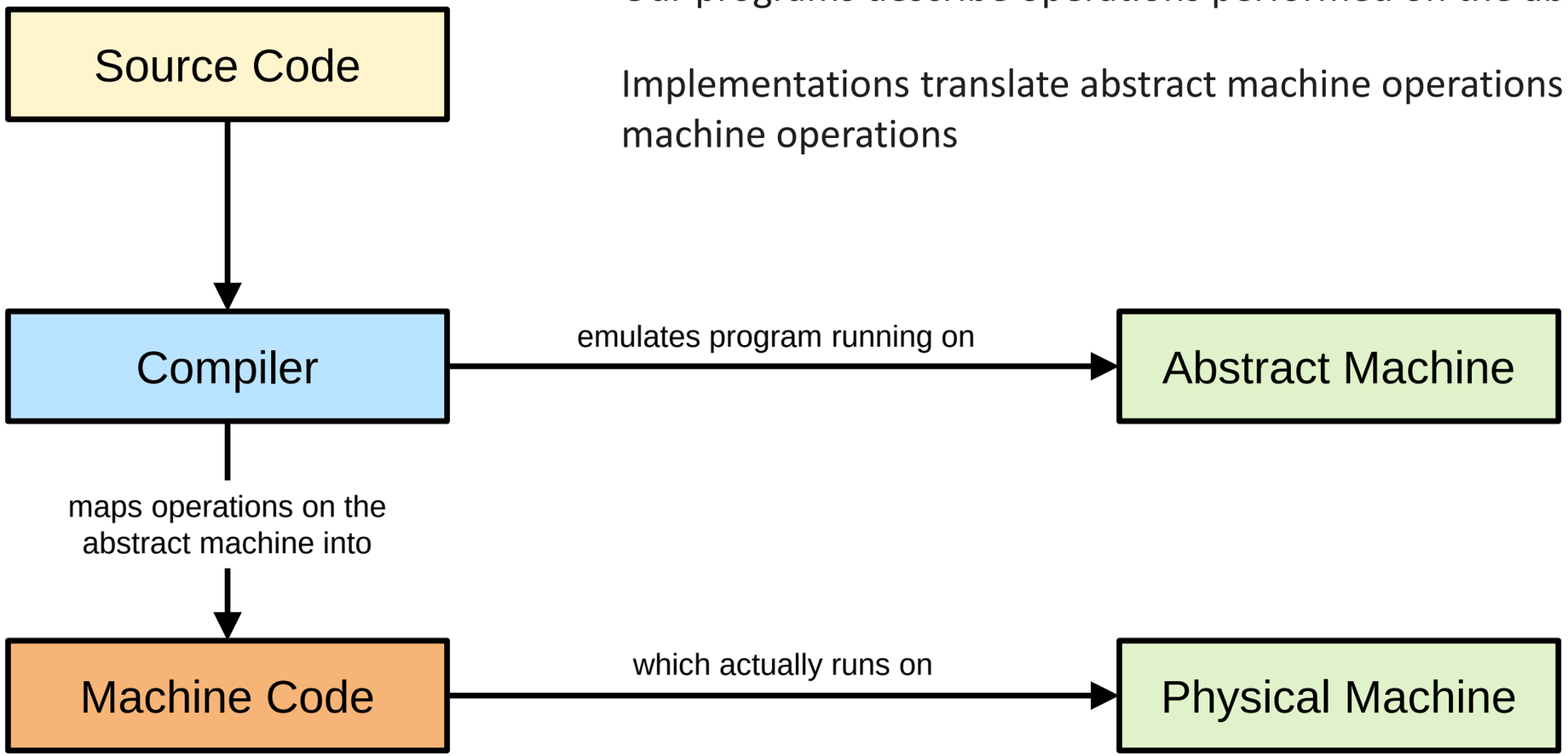
(*for this discussion, the performance- and hardware-related goals)

C++ defines how programs work in terms of an abstract machine deliberately defined to be “close to the hardware”

The Abstract Machine

Our programs describe operations performed on the abstract machine

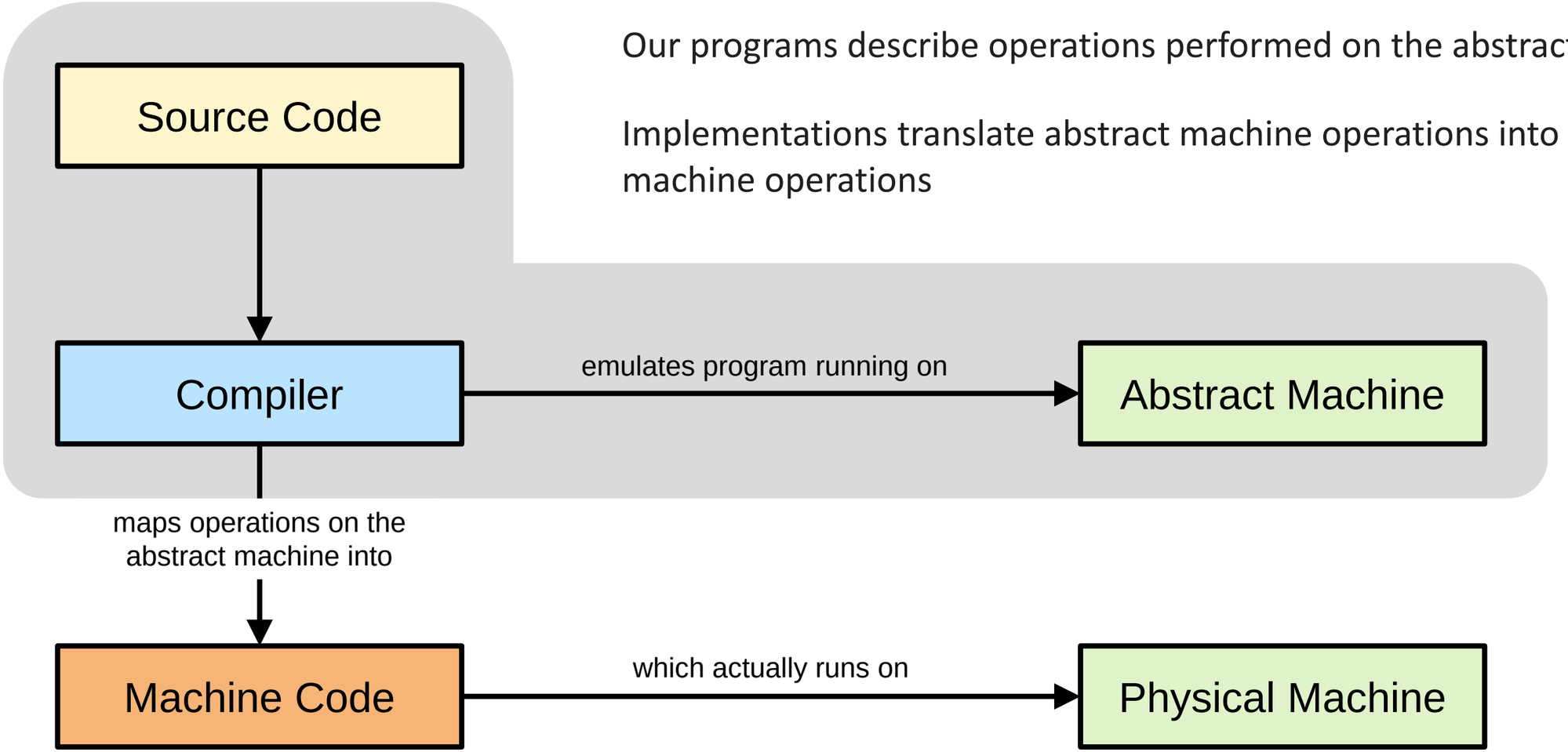
Implementations translate abstract machine operations into physical machine operations



The Abstract Machine – the Programmer's Concern

Our programs describe operations performed on the abstract machine

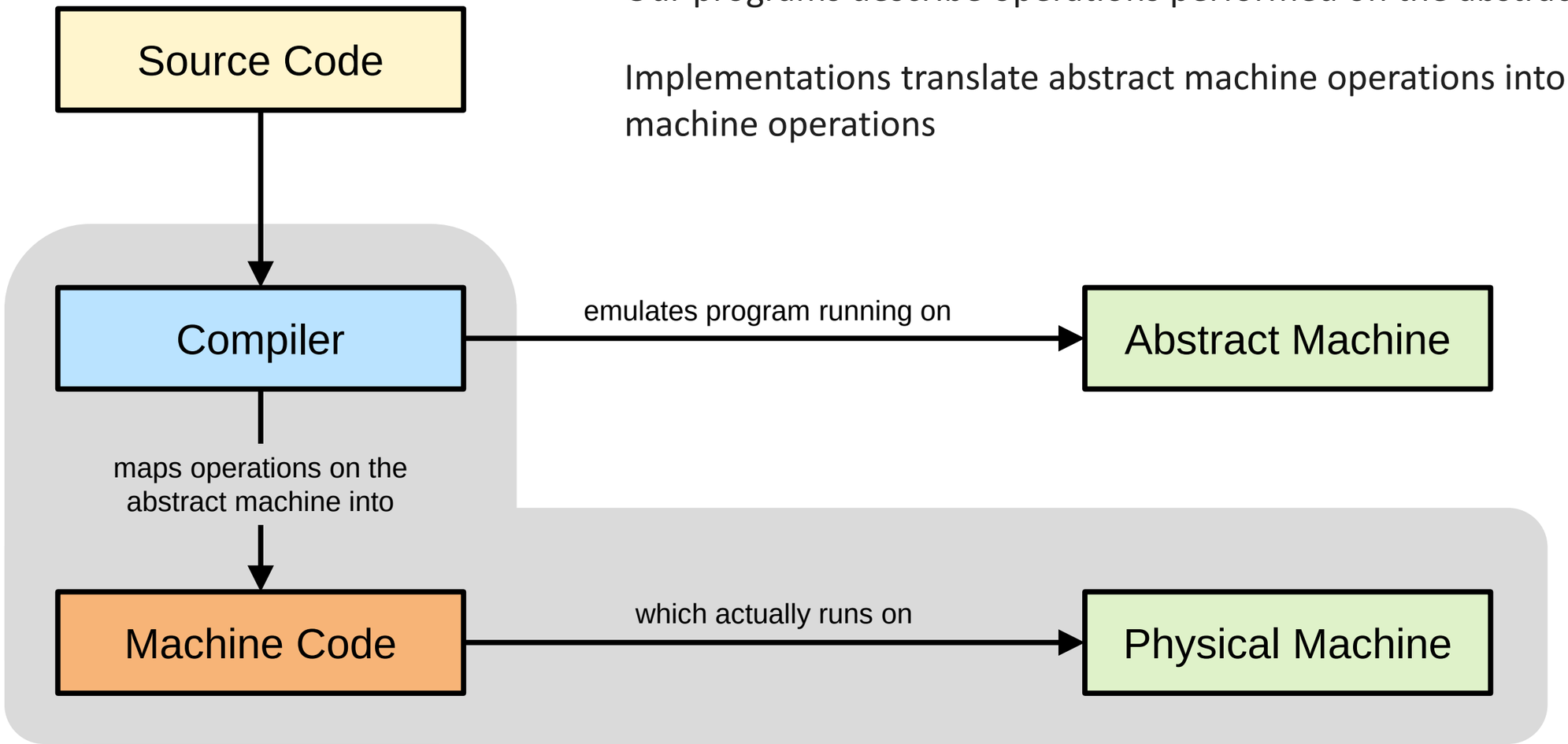
Implementations translate abstract machine operations into physical machine operations



The Abstract Machine – the Compiler's Concern

Our programs describe operations performed on the abstract machine

Implementations translate abstract machine operations into physical machine operations



The Abstract Machine – Central Thesis

When we write C++ code, we are writing to the C++ abstract machine

Defining the Abstract Machine

- The abstract machine is defined in the C++ standard [intro.abstract (4.1)]:

  - (4.1.1) *The semantic descriptions in this document define a* ***parameterized nondeterministic abstract machine****. This document places no requirement on the structure of conforming* ***implementations****. In particular, they need not copy or emulate the structure of the abstract machine. Rather,* ***conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below****.*[4]”
    (my emphasis)

Characteristics of the Abstract Machine

- **Implementation**
- **Parametrized**
- **Non-deterministic**

- Implementations must emulate the **observable behavior** of the abstract machine

Characteristics of the Abstract Machine

- **Implementation**
  - Tool(s) to verify a program on the abstract machine and generate an executable image (compiler, linker)
- **Parametrized**
- **Non-deterministic**

- Implementations must emulate the **observable behavior** of the abstract machine

Defining the Abstract Machine

- The abstract machine is defined in the C++ standard (cont.):
  - (4.1.1) *The semantic descriptions in this document define a* ***parameterized nondeterministic abstract machine****. This document places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather,* ***conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below****.[4]"*
    (my emphasis)

  - (4) *"This provision is sometimes called the* ***"as-if" rule****, because an implementation is free to disregard any requirement of this document* ***as long as the result is as if the requirement had been obeyed****, as far as can be determined from the observable behavior of the program. For instance,* ***an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced****."*
    (my emphasis)

Characteristics of the Abstract Machine

- **Implementation**
  - Tool(s) to verify a program on the abstract machine and generate an executable image (compiler, linker)
- **Parametrized**
- **Non-deterministic**

- Implementations must emulate the **observable behavior** of the abstract machine
- **Expressions** with non-observable **side effects** may be ignored

Defining the Abstract Machine

- The abstract machine is defined in the C++ standard (cont.):

  […]

  - (4.1.2) *Certain aspects and operations of the abstract machine are described in this document as* ***implementation-defined*** *[…].* ***These constitute the parameters of the abstract machine****. Each implementation shall include documentation describing its characteristics and behavior in these respects. […].* (my emphasis)

Characteristics of the Abstract Machine

- **Implementation**
  - Tool(s) to verify a program on the abstract machine and generate an executable image (compiler, linker)
- **Parametrized**
  - Implementation-defined (and documented) allowable behaviors of the abstract machine
- **Non-deterministic**

- Implementations must emulate the **observable behavior** of the abstract machine
- **Expressions** with non-observable **side effects** may be ignored

Defining the Abstract Machine

- The abstract machine is defined in the C++ standard (cont.):

  […]

  - (4.1.3). *Certain other aspects and operations of the abstract machine are described in this document as unspecified […].* ***Where possible, this document defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution for a given program and a given input****.*
    (my emphasis)

Characteristics of the Abstract Machine

- **Implementation**
  - Tool(s) to verify a program on the abstract machine and generate an executable image (compiler, linker)
- **Parametrized**
  - Implementation-defined (and documented) allowable behaviors of the abstract machine
- **Non-deterministic**
  - Unspecified (undocumented) set of allowable behaviors of the abstract machine

- Implementations must emulate the **observable behavior** of the abstract machine
- **Expressions** with non-observable **side effects** may be ignored

Defining the Abstract Machine

- The abstract machine is defined in the C++ standard (cont.):

  […]

  - (4.1.4) *Certain other operations are described in this document as* ***undefined*** *(for example, the effect of attempting to modify a const object). [Note: This document imposes no requirements on the behavior of programs that contain undefined behavior. —end note]* (my emphasis)

Characteristics of the Abstract Machine

- **Implementation**
  - Tool(s) to verify a program on the abstract machine and generate an executable image (compiler, linker)
- **Parametrized**
  - Implementation-defined (and documented) allowable behaviors of the abstract machine
- **Non-deterministic**
  - Unspecified (undocumented) set of allowable behaviors of the abstract machine
- Operations can be **undefined**
  - No requirements on the behavior of undefined operations – anything can happen

- Implementations must emulate the **observable behavior** of the abstract machine
- **Expressions** with non-observable **side effects** may be ignored

Defining the Abstract Machine

- The abstract machine is defined in the C++ standard (cont.):

  […]

  - (4.1.5) *A conforming implementation executing a well-formed program* ***shall produce the same observable behavior as one of the possible executions of the corresponding instance of the abstract machine with the same program and the same input****. However, if any such execution contains an undefined operation, this document places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).* (my emphasis)

Characteristics of the Abstract Machine

- **Implementation**
  - Tool(s) to verify a program on the abstract machine and generate an executable image (compiler, linker)
- **Parametrized**
  - Implementation-defined (and documented) allowable behaviors of the abstract machine
- **Non-deterministic**
  - Unspecified (undocumented) set of allowable behaviors of the abstract machine
- Operations can be **undefined**
  - No requirements on the behavior of undefined operations – anything can happen
- Implementations must emulate the **observable behavior** of the abstract machine
  - Many possible valid execution paths (observable behaviors) could occur in the abstract machine
  - Implementation must match one of them
- **Expressions** with non-observable **side effects** may be ignored

Defining the Abstract Machine

- The abstract machine is defined in the C++ standard (cont.):

  […]

  - (4.1.6) *The least requirements on a conforming implementation are:*
    - *Accesses through volatile glvalues are evaluated strictly according to the rules of the abstract machine.*
    - *At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.*
    - *The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.*

  These collectively are referred to as the observable behavior of the program.

  (emphasis mine)

Characteristics of the Abstract Machine

- **Implementation**
  - Tool(s) to verify a program on the abstract machine and generate an executable image (compiler, linker)
- **Parametrized**
  - Implementation-defined (and documented) allowable behaviors of the abstract machine
- **Non-deterministic**
  - Unspecified (undocumented) set of allowable behaviors of the abstract machine
- Operations can be **undefined**
  - No requirements on the behavior of undefined operations – anything can happen
- Implementations must emulate the **observable behavior** of the abstract machine
  - *Observable behavior* - execution state information exchanged over time with outside entities
  - Many possible valid execution paths (observable behaviors) could occur in the abstract machine
  - Implementation must match one of them
- **Expressions** with non-observable **side effects** may be ignored

Defining the Abstract Machine

- The abstract machine is defined in the C++ standard (cont.):

  […]

  - [intro.execution](6.9.1.7) *Reading an object designated by a volatile glvalue […], modifying an object, calling a library I/O function, or calling a function that does any of those operations are all* ***side effects, which are changes in the state of the execution environment****.*
    (my emphasis)

  […]

  - [expr.pre](7.1) *[…] An expression is a sequence of operators and operands that specifies a computation.* ***An expression can result in a value and can cause side effects****.*
    (my emphasis)

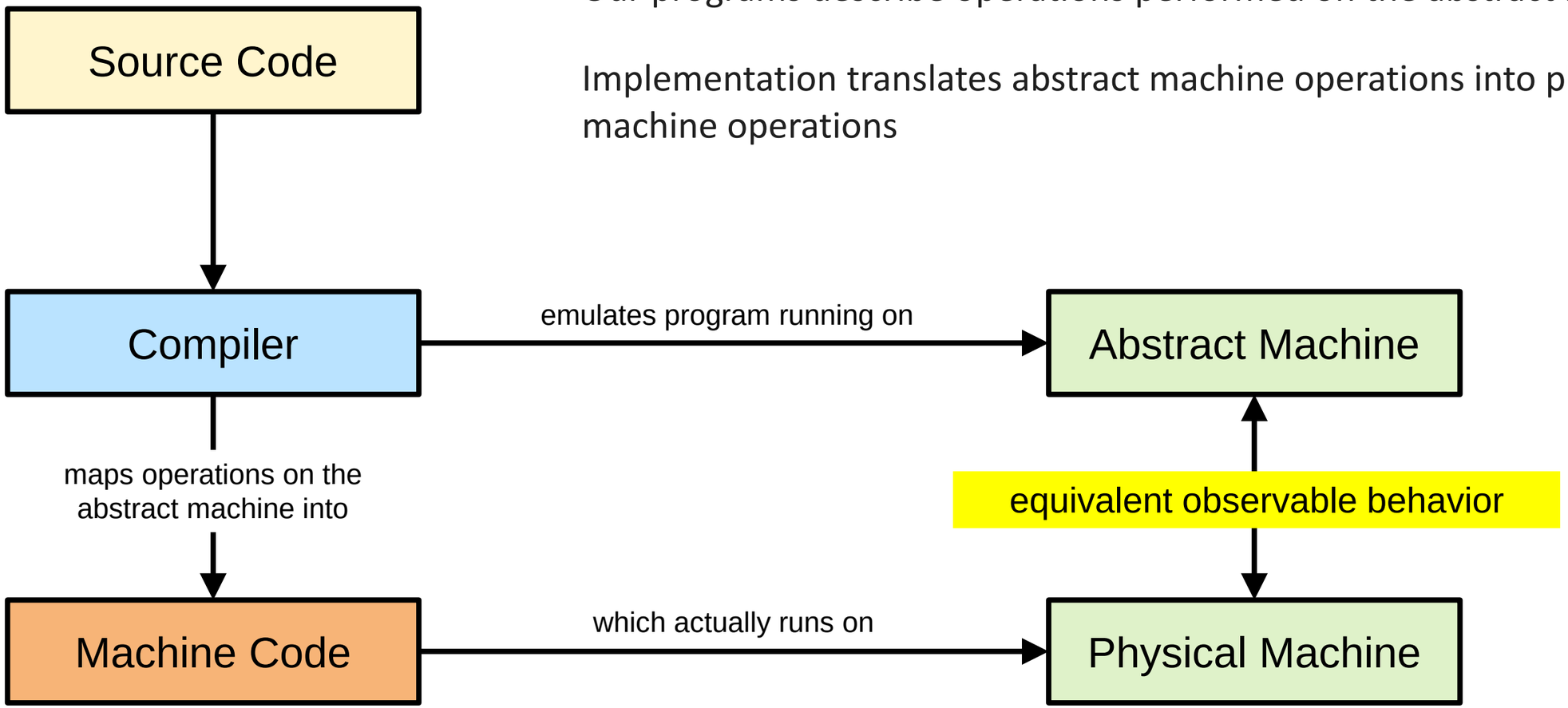
Characteristics of the Abstract Machine

- **Implementation**
  - Tool(s) to verify a program on the abstract machine and generate an executable image (compiler, linker)
- **Parametrized**
  - Implementation-defined (and documented) allowable behaviors of the abstract machine
- **Non-deterministic**
  - Unspecified (undocumented) set of allowable behaviors of the abstract machine
- Operations can be **undefined**
  - No requirements on the behavior of undefined operations – anything can happen
- Implementations must emulate the **observable behavior** of the abstract machine
  - *Observable behavior* - execution state information exchanged over time with outside entities
  - Many possible valid execution paths (observable behaviors) could occur in the abstract machine
  - Implementation must match one of them
- **Expressions** with non-observable **side effects** may be ignored
  - *Expression* - a sequence of operators and operands that specifies a computation
  - *Side effects* - changes in the program's execution state

The Abstract Machine

Our programs describe operations performed on the abstract machine

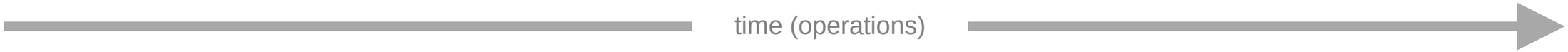
Implementation translates abstract machine operations into physical machine operations



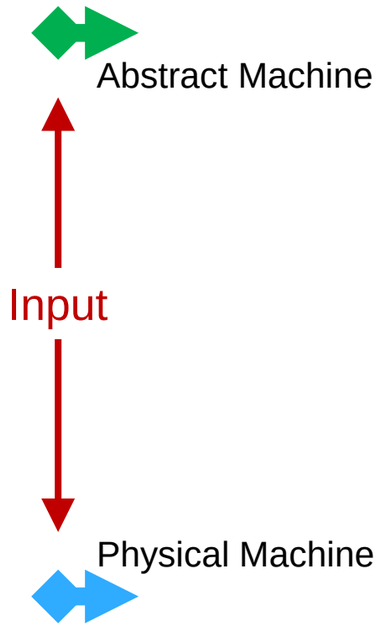
A Program in Action

Abstract Machine

Physical Machine

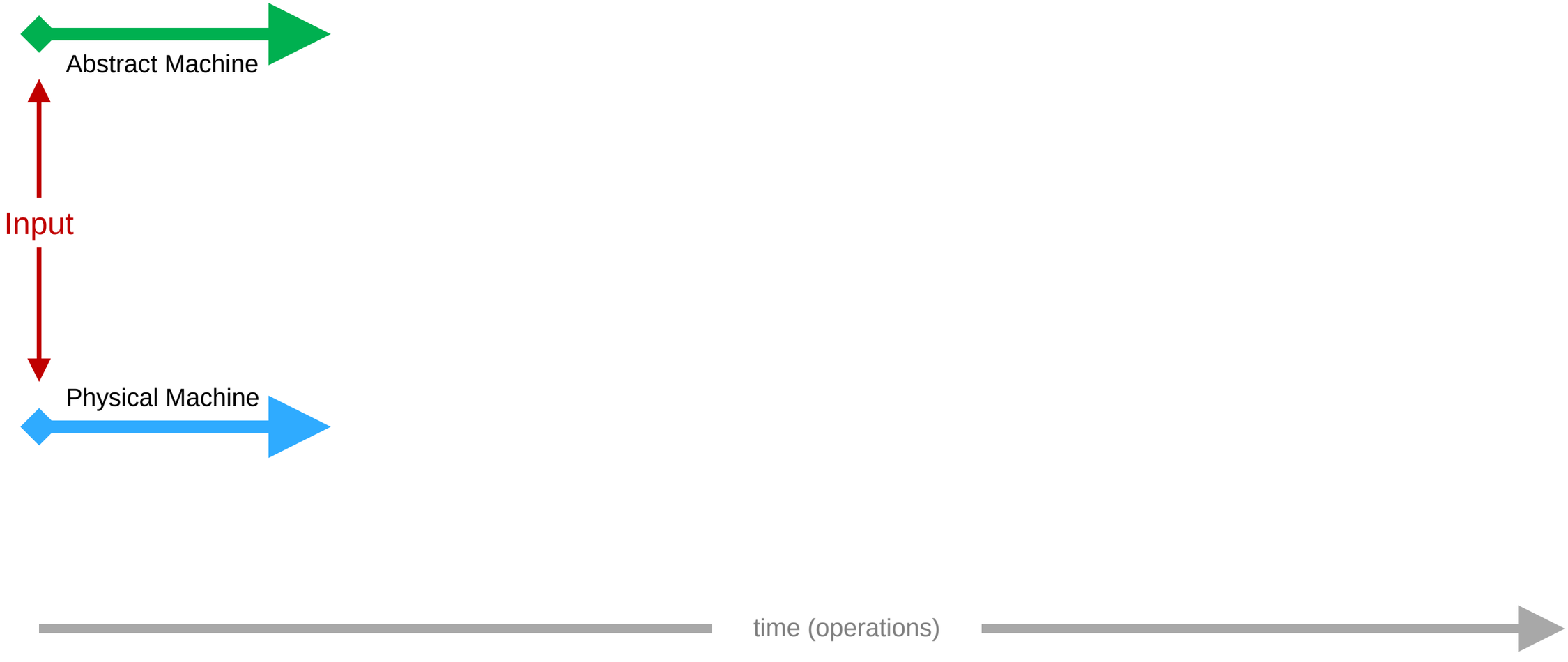


A Program in Action

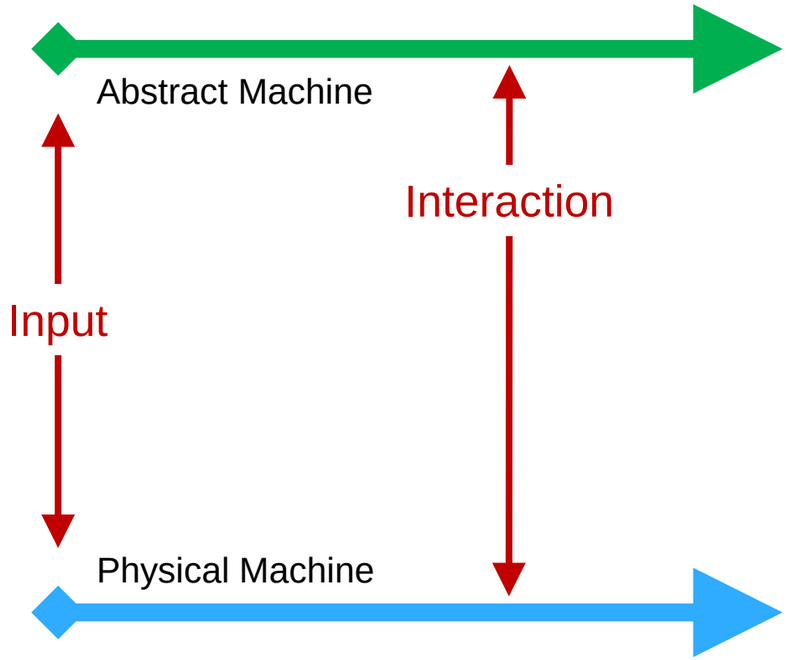


time (operations)

A Program in Action

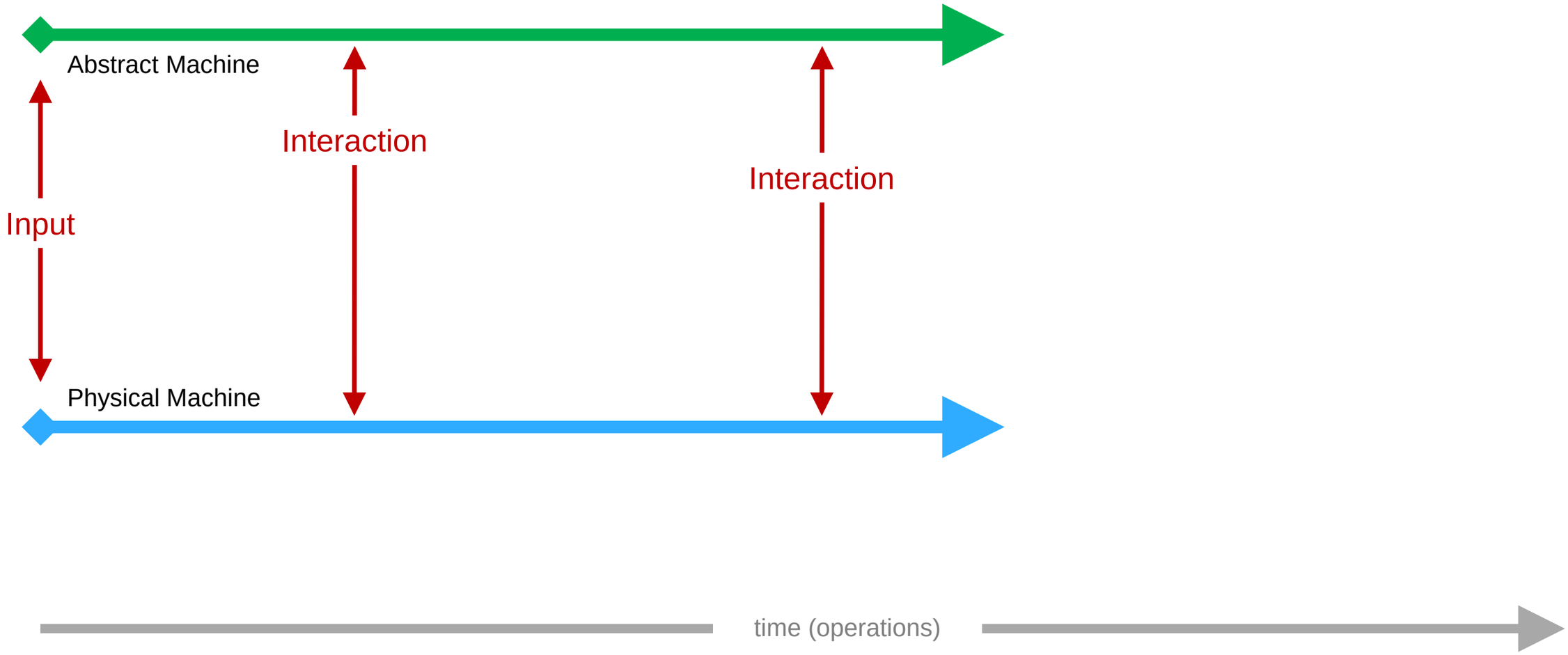


A Program in Action

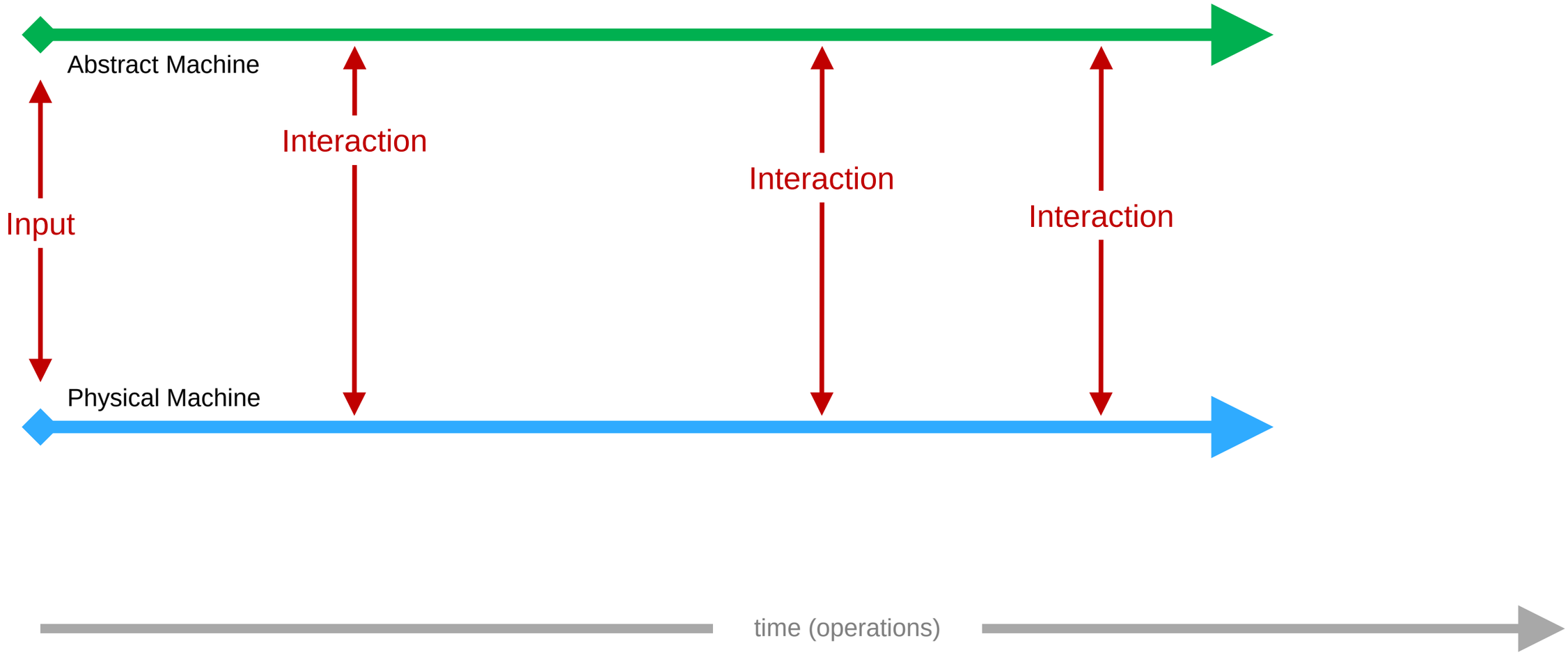


time (operations)

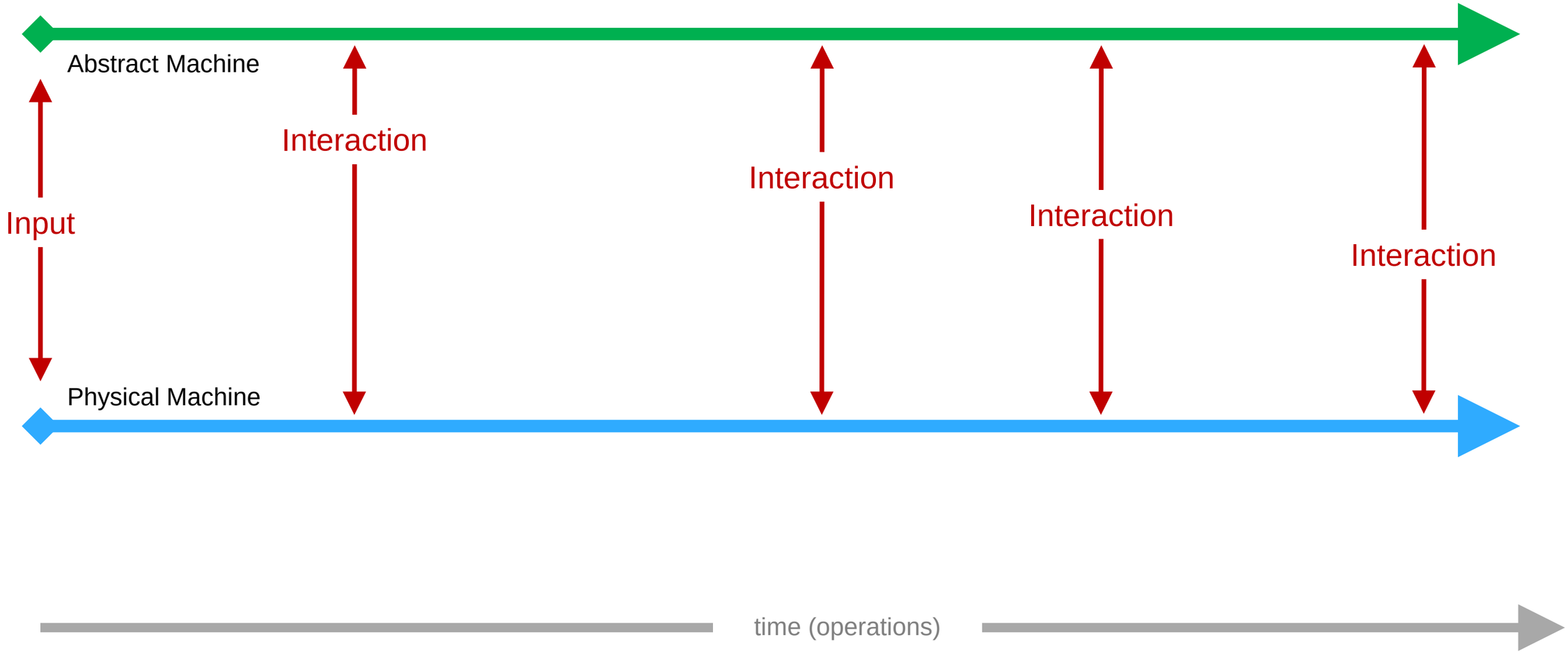
A Program in Action



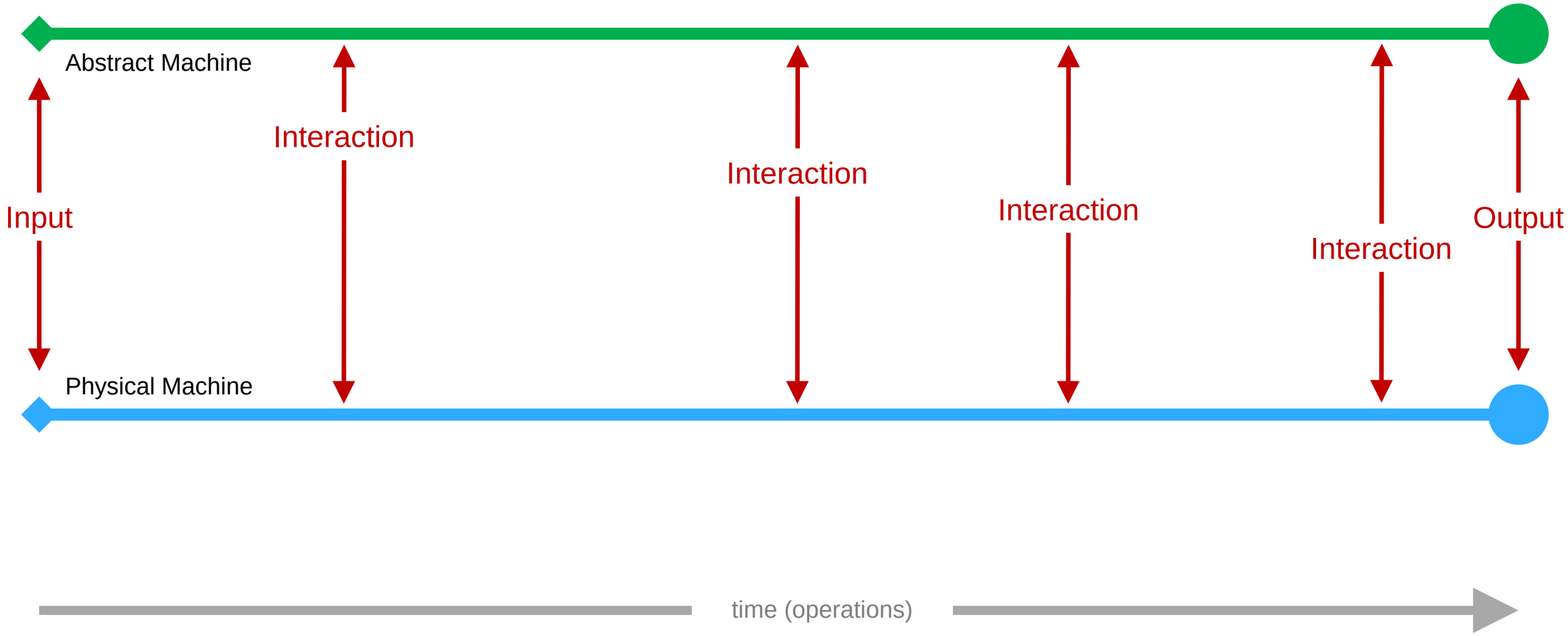
A Program in Action



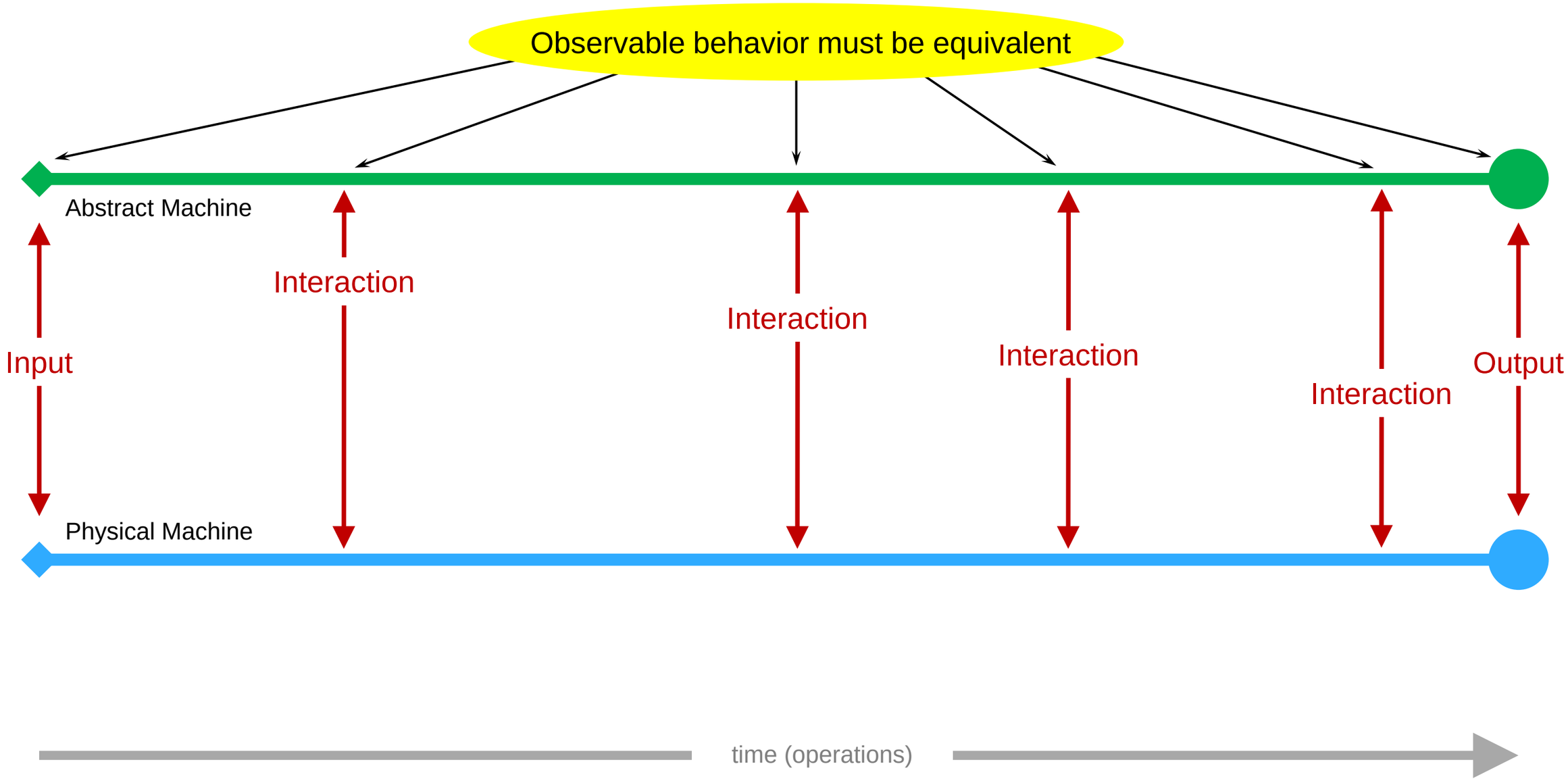
A Program in Action



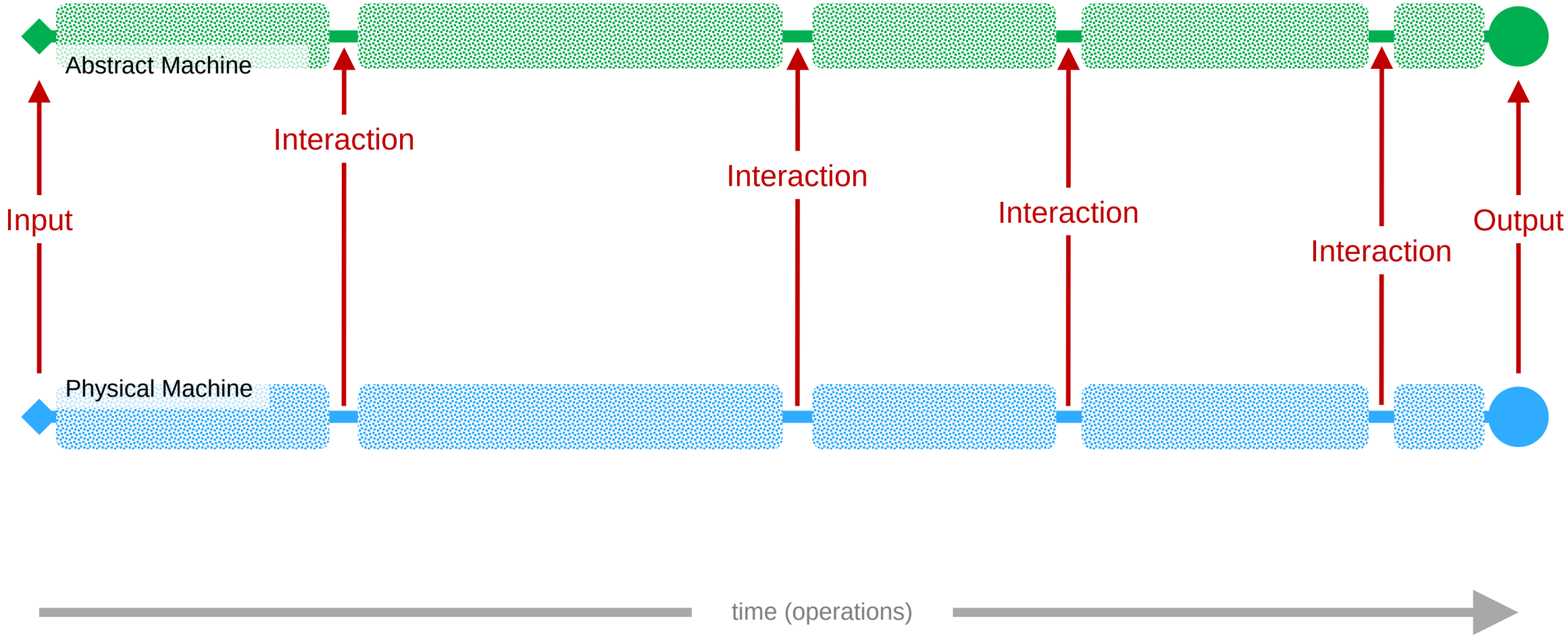
A Program in Action



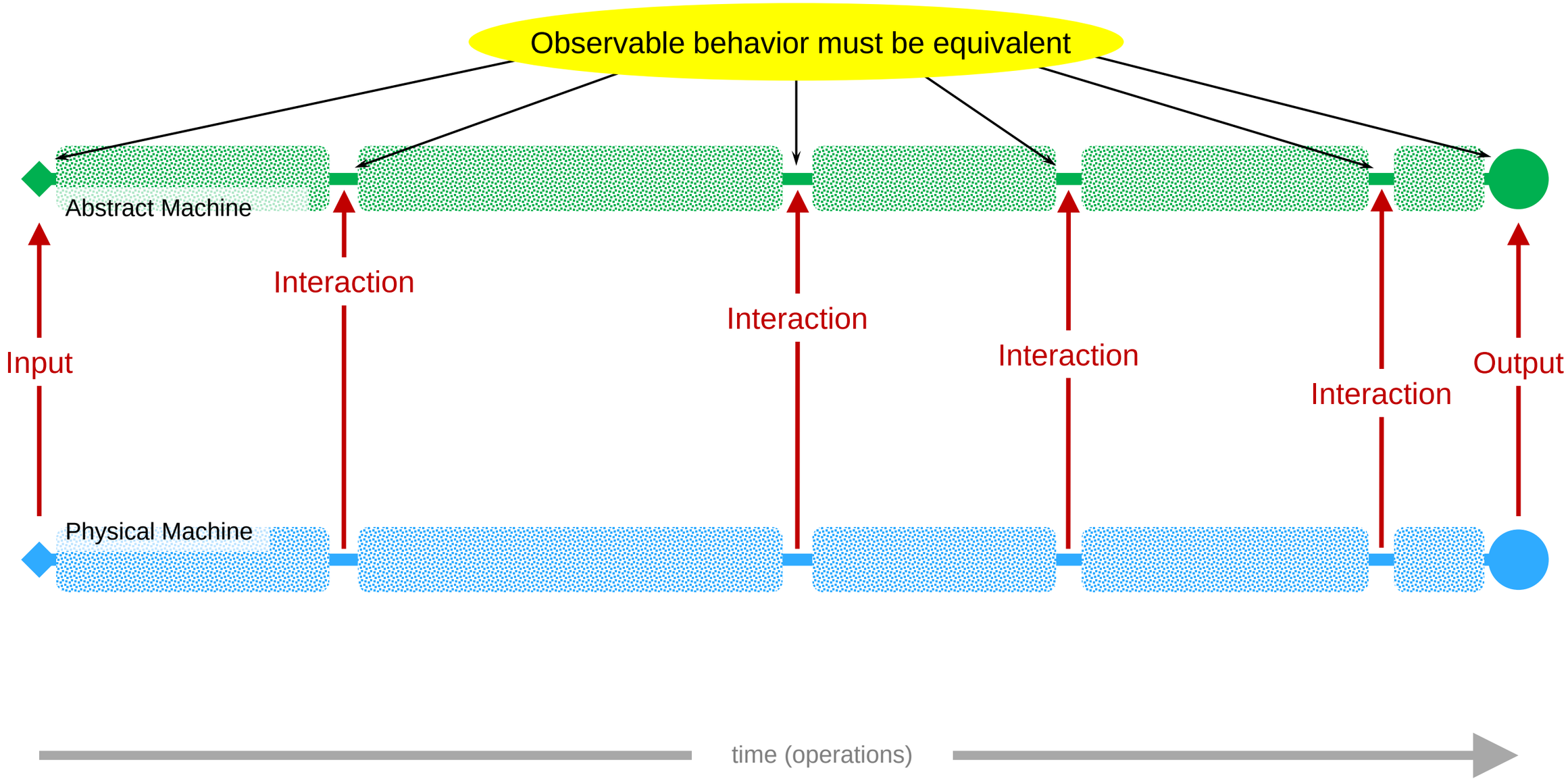
A Program in Action



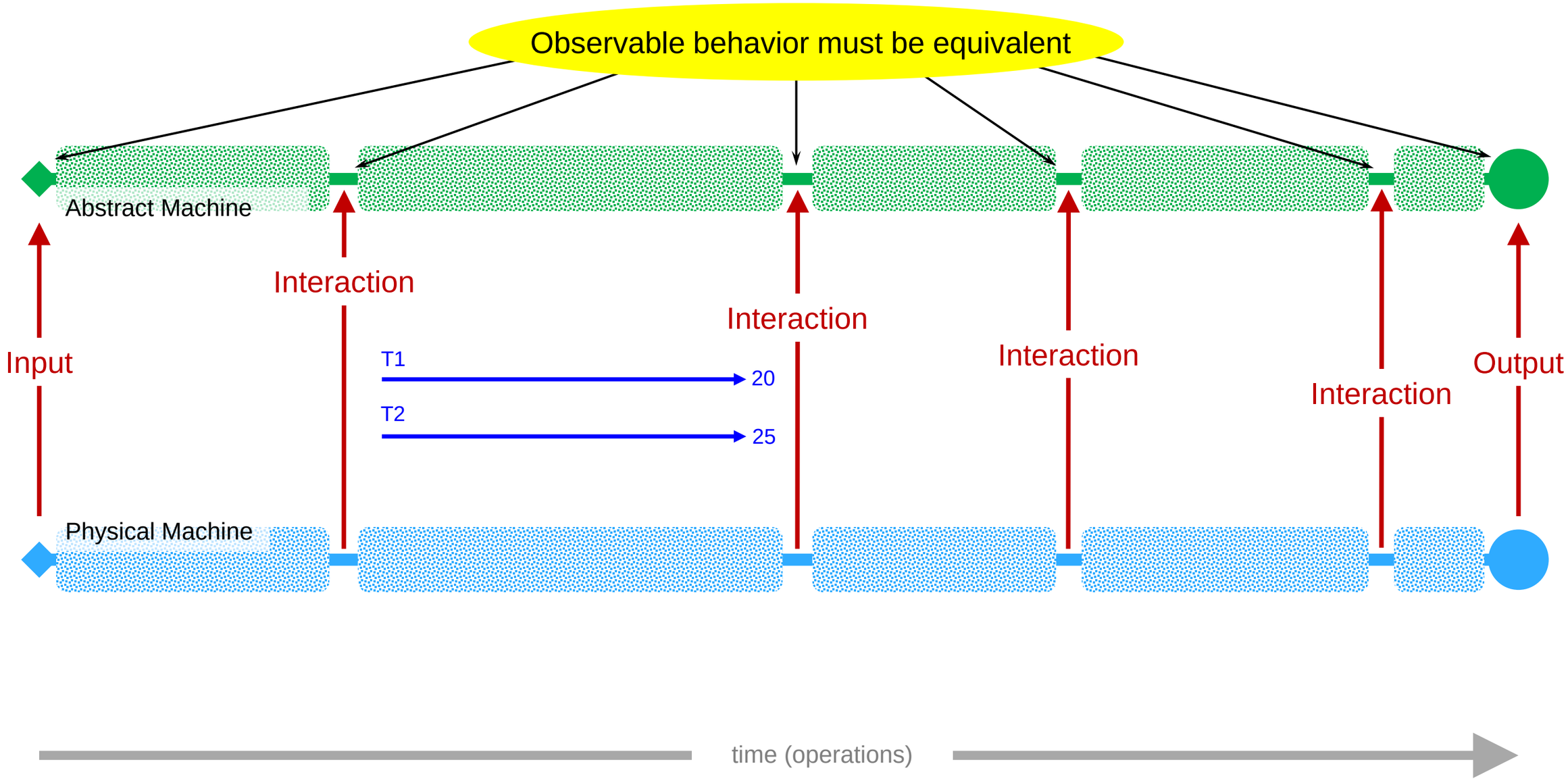
A Program in Action



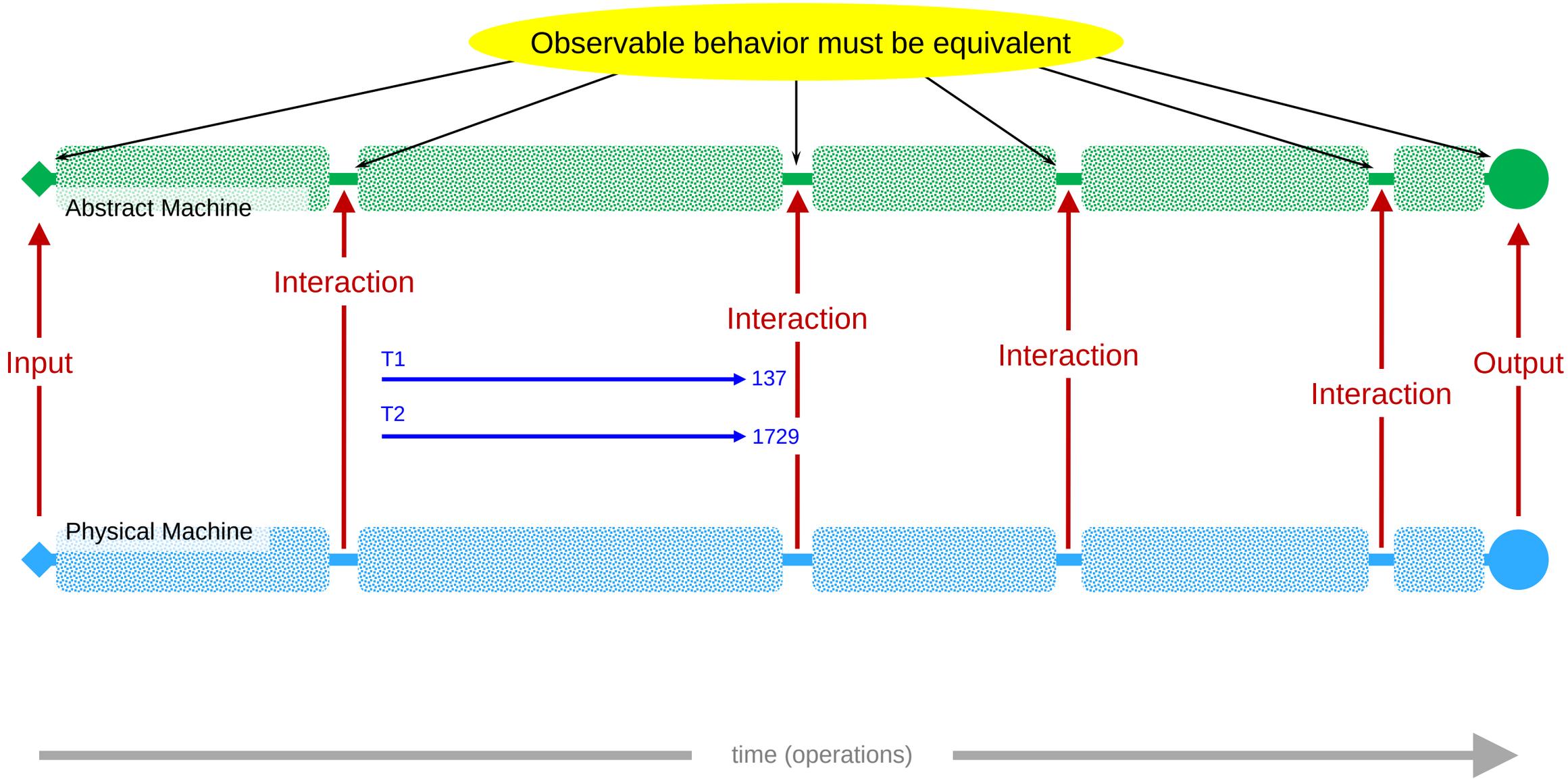
A Program in Action



A Program in Action



A Program in Action



Well-Formed Program

- [defns.well.formed](3.32) *C++ program constructed according to the syntax rules, diagnosable semantic rules, and the one-definition rule*

- This refers to a valid program (at least as far as the implementation is concerned)

Implementation-Defined Behavior

- [defs.impl.defined](3.13) *behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents*

- This is the parameterization aspect of the abstract machine

- For example:
  - `sizeof(void*)` (the size of a pointer)
  - The value of `CHAR_BIT` (indicating the number of bits in a byte)
  - The text returned by `std::bad_alloc::what()`
  - Locale-specific behavior, which depends on the implementation-supplied locale.

Unspecified Behavior

- [defns.unspecified](3.31) *behavior, for a well-formed program construct and correct data, that depends on the implementation*

- Each unspecified behavior yields one result from the set of all possible valid results

- This is the nondeterminism aspect of the abstract machine

- For example:
  - The order of evaluation of arguments in a function call
  - Whether identical string literals are stored distinctly by a program
  - The order, contiguity, and initial value of storage returned by successive allocation requests

Undefined Behavior (UB)

- [defns.undefined](3.30) *behavior for which this document imposes no requirements*

- In this case there are no restrictions on the behavior of the program
  - Compilers are not required to diagnose undefined behavior
  - A program with undefined behavior is not required to do anything meaningful

- For example:
  - Dereferencing a null pointer
  - Memory accesses outside of array bounds
  - Signed integer overflow
  - Accessing an object through a pointer of different type

Ill-Formed

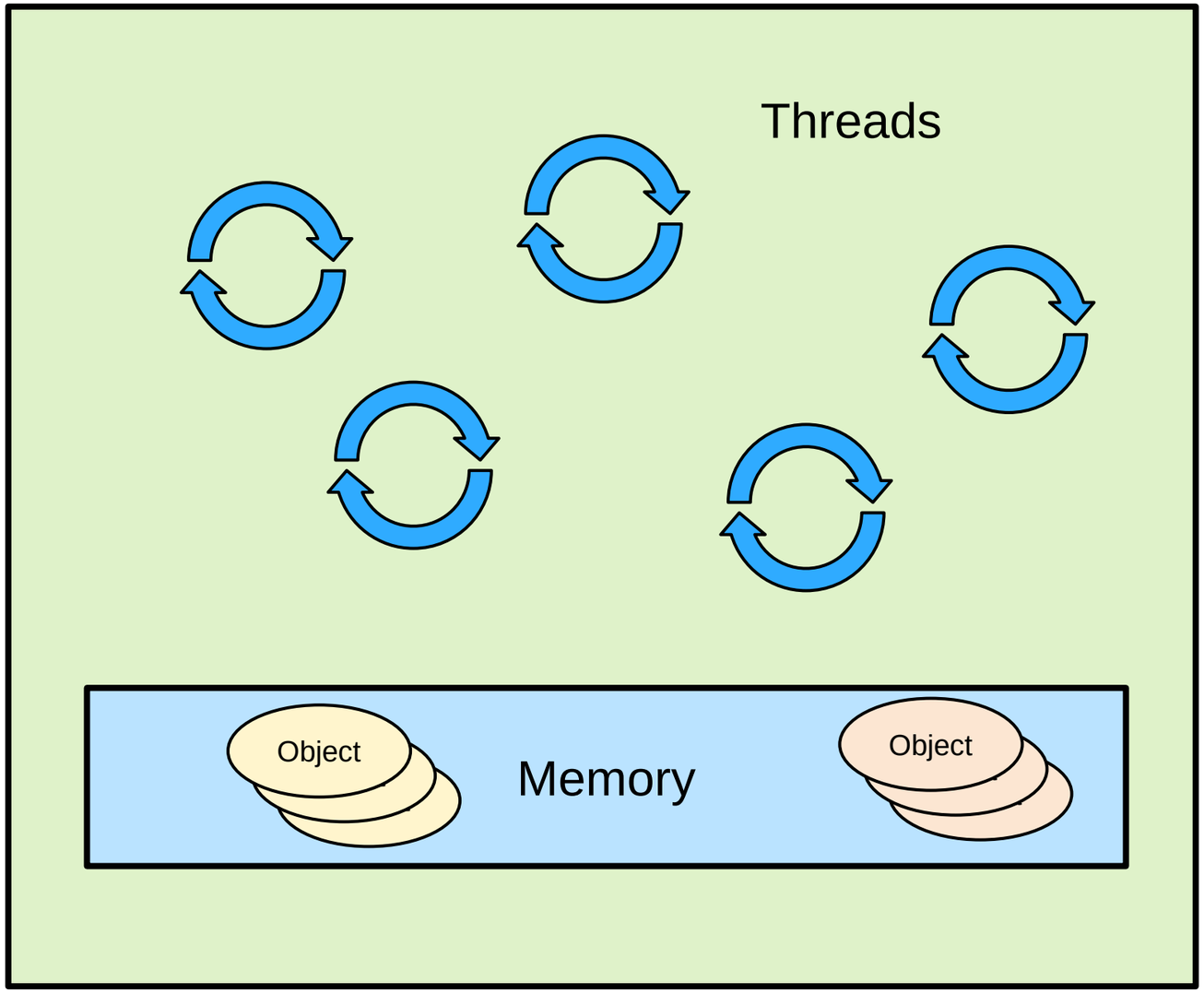
- [defns.ill.formed](3.12) *a program that is not well-formed*
- The program has syntax errors or diagnosable semantic errors
- An implementation must issue a diagnostic, even if that implementation defines a language extension that assigns meaning to such code
- The standard uses the words *shall*, *shall not*, and *ill-formed* to indicate requirements that must be met for a program to not be ill-formed

Ill-Formed, No Diagnostic Required (IFNDR)

- In this case, the program has semantic errors which may not be diagnosable at compile time
- The behavior is undefined if such program is executed
- Examples:
  - Violations of the *one definition rule*
  - A constructor delegates to itself, directly or indirectly
  - Declaring a function `noreturn` in one translation unit and declaring without `noreturn` in another translation unit

The Abstract Machine – Structure

- Memory
- Objects
- Threads



Structure of the Abstract Machine – Memory

- Memory is a single flat space
- All parts of memory are equally reachable by the abstract machine
- There is no memory hierarchy
  - No concept of stack, registers, or cache
    - (Although *stack unwinding* is mentioned several times)
  - No concept of heterogeneous memory (e.g., GPUs, co-processors)
- Access to memory has uniform latency

Structure of the Abstract Machine – Memory

- Memory is composed of **bytes**
  - The standard specifies the minimum requirements for what a byte is in terms of what it must be able to represent
- The memory available to a program consists of one or more sequences of contiguous bytes
  - Any operation in a program can potentially access any memory location in those sequences of bytes
- Every byte has a unique location in memory – its **address**
  - Addresses are represented in our program by **pointers**

Structure of the Abstract Machine – Objects

- Operations in a program create, destroy, refer to, access, and manipulate **objects**, which have
  - size
  - alignment
  - storage duration
  - lifetime
  - type
  - value
  - name (optional)

- An object may have at most one **memory location**
  - An implementation is allowed to store two objects at the same address, or not store an object at all, if the program cannot observe the difference (the *as-if* rule at work!)

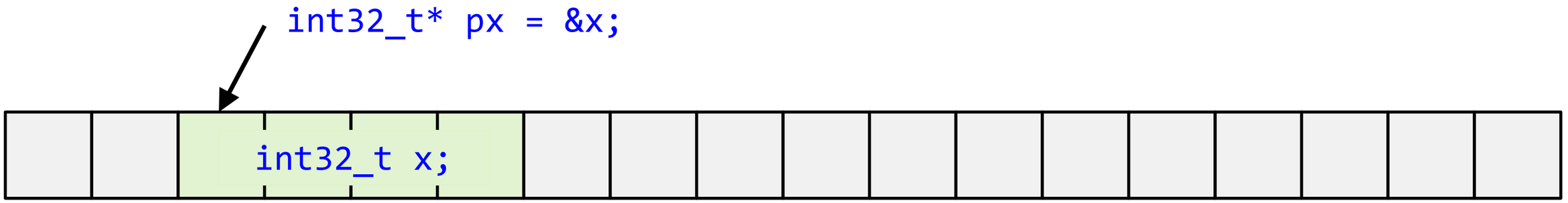
Structure of the Abstract Machine – Objects

- An object having a memory location is stored in a contiguous sequence of one or more bytes

Structure of the Abstract Machine – Objects

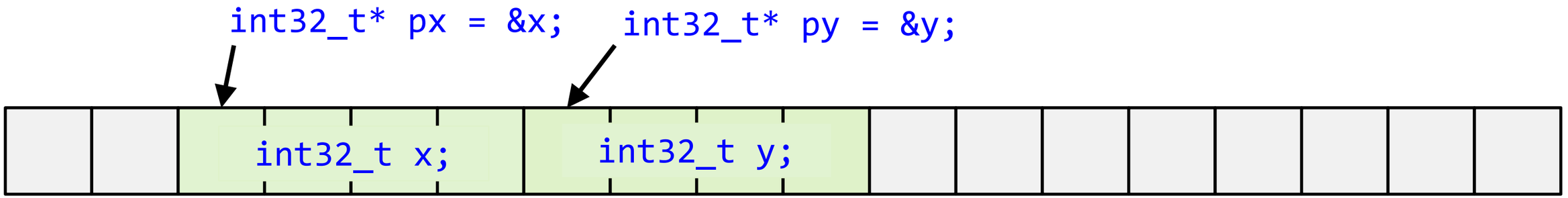
- An object having a memory location is stored in a contiguous sequence of one or more bytes

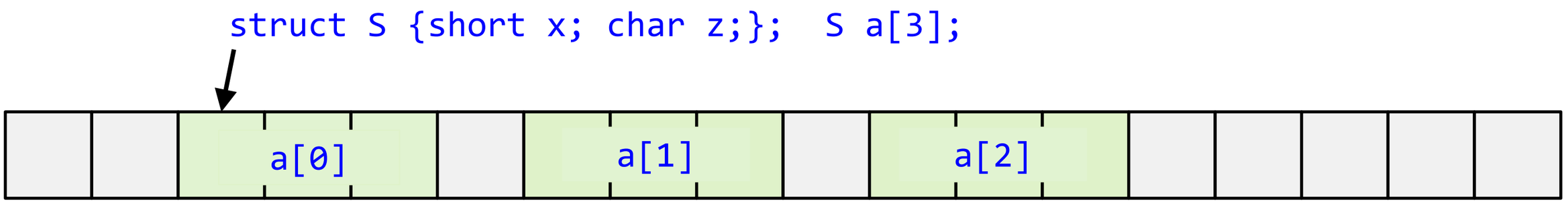
- The address of an object stored in memory is the address of its first byte
  - The address is represented by a **pointer**, which is itself a kind of object



Structure of the Abstract Machine – Objects

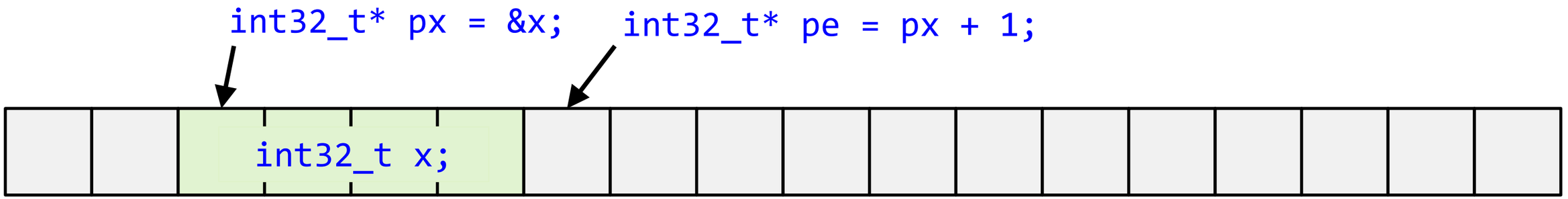
- Arrays of objects, and data members of classes may or may not occupy contiguous sequences of bytes
  - Structure padding is permitted
  - However, arrays are contiguously indexable from 0 to N-1

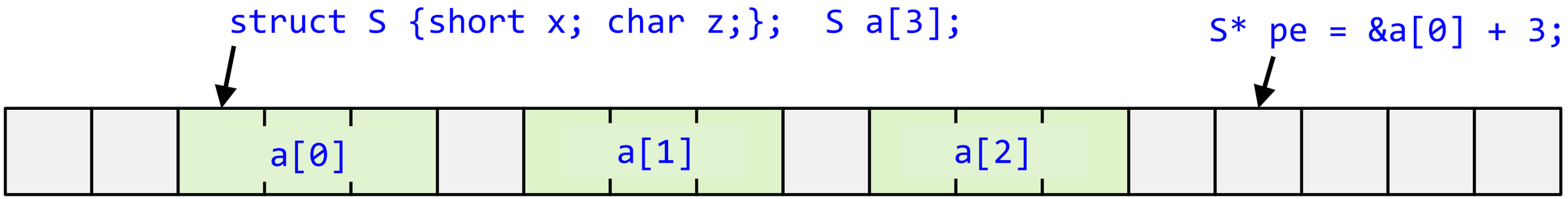




Structure of the Abstract Machine – Objects

- Pointers may point to the one-past-end element of an array, one-past-end of an object

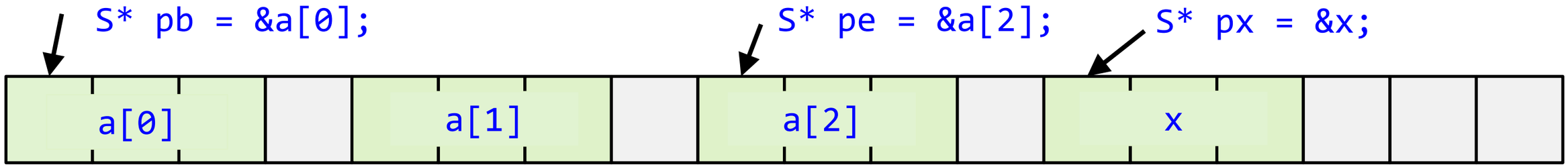




Structure of the Abstract Machine – Objects

- Pointers to objects may be compared in certain circumstances
  - Equality and inequality when pointing to the same type
  - For ordering when pointing to elements of same array
  - For ordering when pointing data members of the same class

```
struct S {short x; char z;};  S a[3], x;
```




```
pb == pe;  // OK
pb == px;  // OK
pb != px;  // OK
pb <  pe;  // OK
pb <  px;  // Invalid - UB
```

Structure of the Abstract Machine – Objects

- An object has a **storage duration**

- **automatic** storage duration
  - Object storage is allocated at the beginning of the enclosing block and deallocated at the end of the block
  - Applies to all local objects except those declared `thread_local`, `static`, or `extern`

- **dynamic** storage duration
  - Object storage is allocated and deallocated by the program using functions that perform dynamic memory allocation
  - Objects with this duration can be created using *new-expressions* and destroyed using *delete-expressions*

Structure of the Abstract Machine – Objects

- **static** storage duration
  - Object storage is allocated at the beginning of the program and deallocated at the end of the program
  - Applies to all objects declared at namespace scope, including the global namespace
  - Applies to all objects declared `static` or `extern`
  - There is only one instance of an object with static duration in the program

- **thread** storage duration
  - Object storage is allocated when the thread creating the object begins and deallocated when that thread ends
  - Applies only to objects declared `thread_local`
  - Each thread has its own instance of an object with thread duration

Structure of the Abstract Machine – Objects

- An object has a **lifetime**
- The lifetime of an object of type `T` begins when
  - Storage with the proper size and alignment is obtained, and
  - Its initialization is complete
- The lifetime of an object `obj` of type `T` ends when
  - The object is destroyed, if `T` is a non-class type; or,
  - The destructor call starts, if `T` is a class type; or,
  - The storage which the object occupies is released, or is reused by an object that is not nested within `obj`

Structure of the Abstract Machine - Threads

- A thread of execution (also known as a thread) is a single flow of control within a program
  - This includes the initial invocation of a specific top-level function, and
  - Recursively includes every function invocation subsequently executed by the thread
  - A C++ program can have more than one thread running concurrently
  - Every thread in a program can potentially access every object and function in that program
- When one thread creates another, the initial call to the top-level function of the new thread is executed by the new thread, not by the creating thread

Structure of the Abstract Machine - Threads

KEWB COMPUTING

- `main()` is a special top-level function
  - It must have C++ language linkage
  - It must have a return type of int and implementations must provide at least these forms:
    - `int main()`
    - `int main(int, char**)`
  - It cannot be overloaded, and cannot be a coroutine
  - It cannot be called (invoked) by the program
  - It cannot declared as `delete`'d, `static`, `inline`, or `constexpr`
  - It cannot be declared with a linkage specification
  - A program cannot declare a variable named `main` at global scope
  - A program cannot declare the name main with C language linkage

Structure of the Abstract Machine - Threads

KEWB COMPUTING

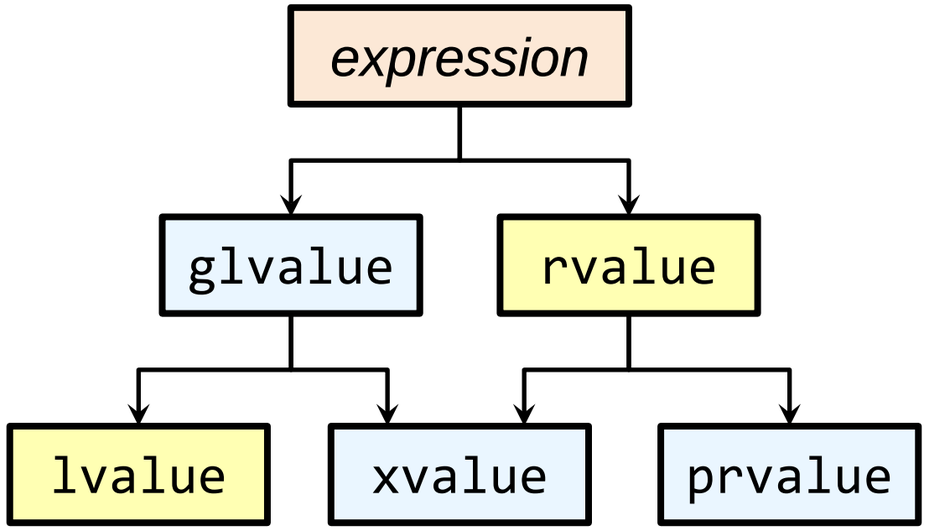
- `main()` is the entry point for the program
- Executing a program starts the main thread of execution which calls `main()`
  - Objects of static storage duration are initialized before `main()` is called and destroyed after it exits
  - Every program has at least one thread (the main thread)
  - The top level function for every program is `main()`
- What does it mean to invoke and execute `main()`?

Structure of the Abstract Machine - Threads

- What does it mean to invoke and execute a function?
  - Functions consist of statements
  - Statements consist of expressions
- An expression is a sequence of operators and operands that specifies a computation
  - To perform that computation, the expression must be **evaluated**
  - Evaluating an expression can result in a value and can cause side effects
- Evaluating an expressions that causes side effects results in changes to the program's execution state, and possibly to observable behavior

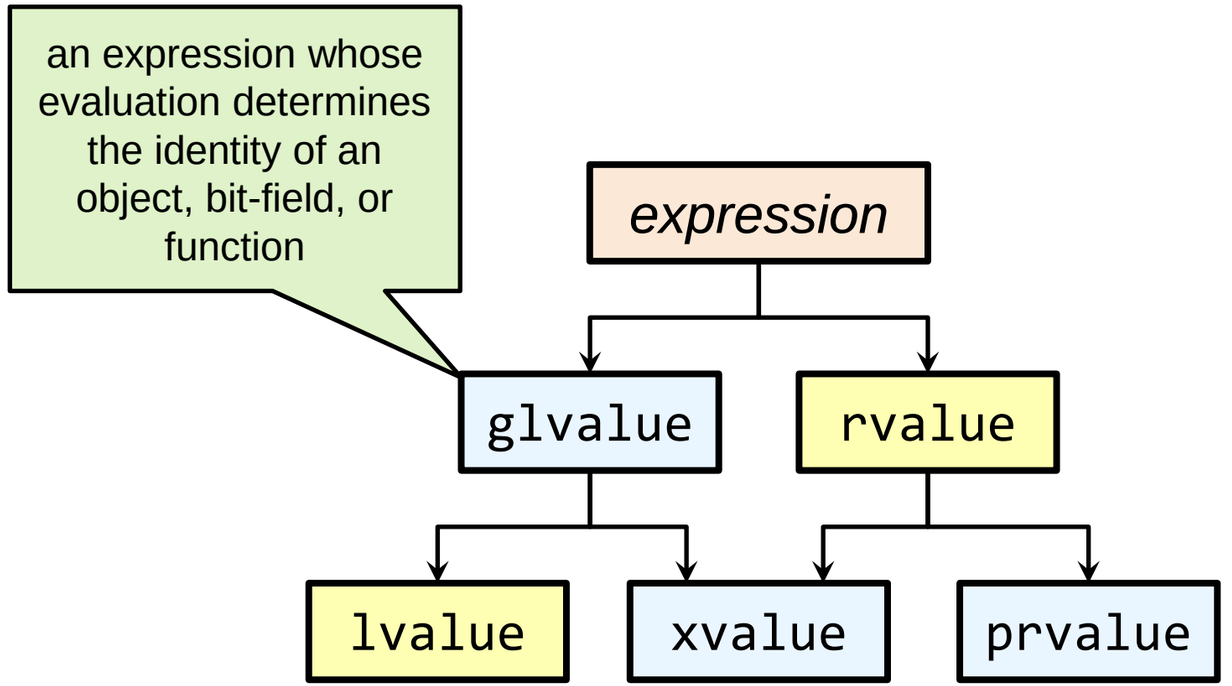
Structure of the Abstract Machine – Threads and Expressions

- Every C++ expression has an associated **type**
- Every C++ expression is a member of a **value category**
- **rvalues** indicate objects that are
  - Nameless
  - Temporary objects, literals, etc.
  - Cannot have their address taken
- **lvalues** indicate objects that
  - Have names, directly or indirectly
  - Are non-temporary
  - May have their address taken



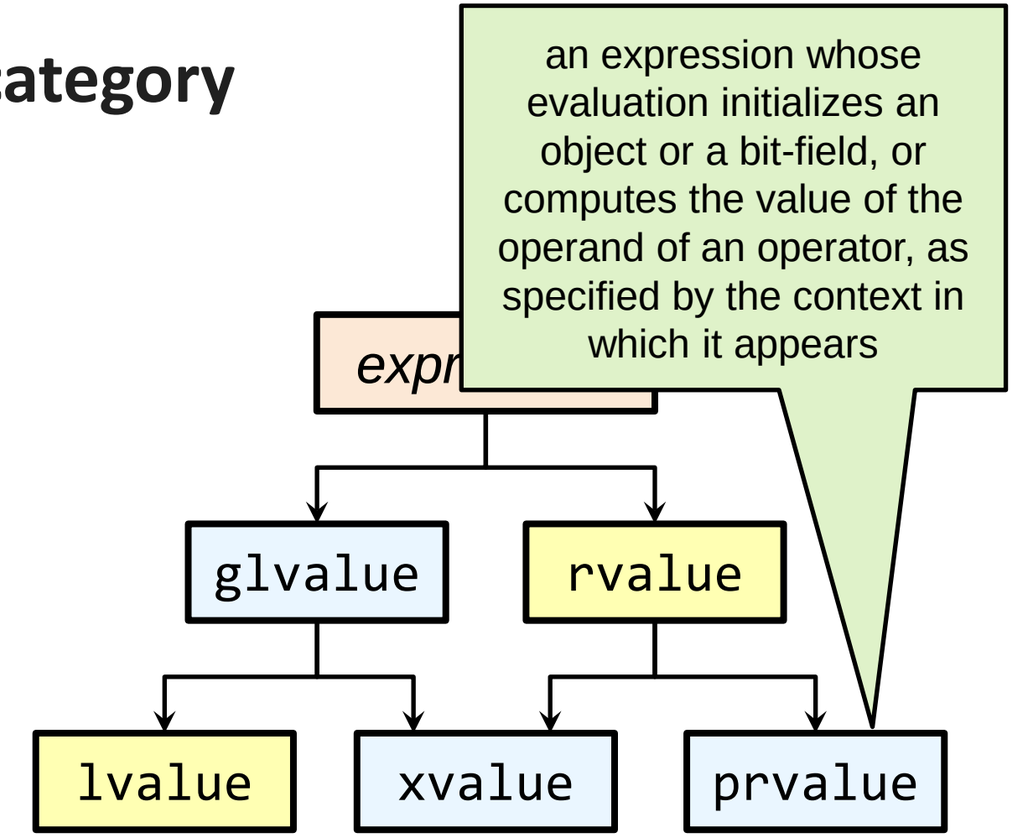
Structure of the Abstract Machine – Threads and Expressions

- Every C++ expression has an associated **type**
- Every C++ expression is a member of a **value category**
- **rvalues** indicate objects that are
  - Nameless
  - Temporary objects, literals, etc.
  - Cannot have their address taken
- **lvalues** indicate objects that
  - Have names, directly or indirectly
  - Are non-temporary
  - May have their address taken



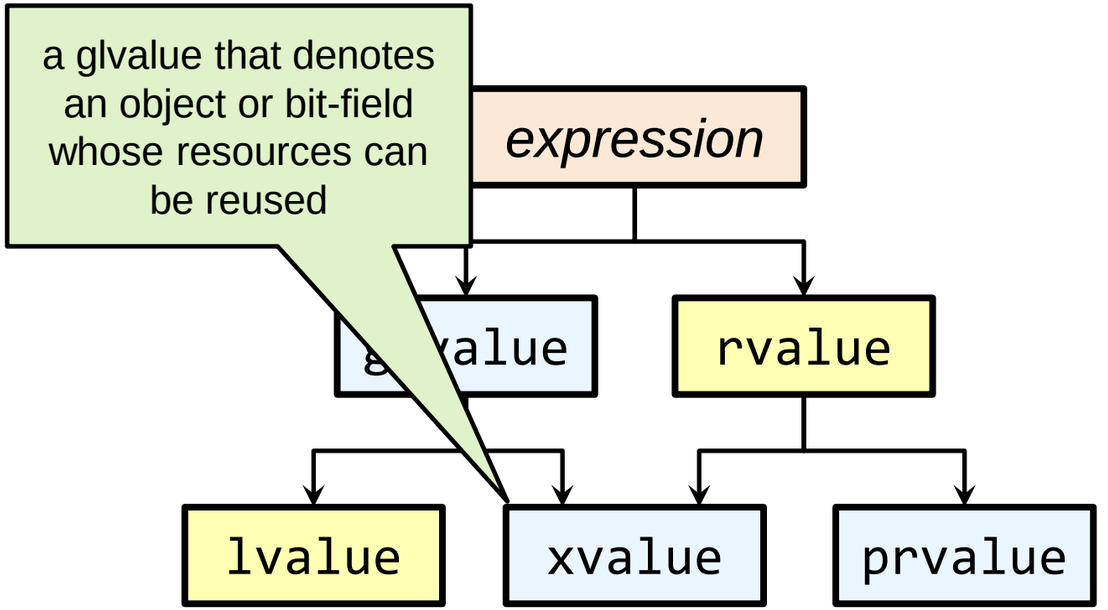
Structure of the Abstract Machine – Threads and Expressions

- Every C++ expression has an associated **type**
- Every C++ expression is a member of a **value category**
- **rvalues** indicate objects that are
  - Nameless
  - Temporary objects, literals, etc.
  - Cannot have their address taken
- **lvalues** indicate objects that
  - Have names, directly or indirectly
  - Are non-temporary
  - May have their address taken



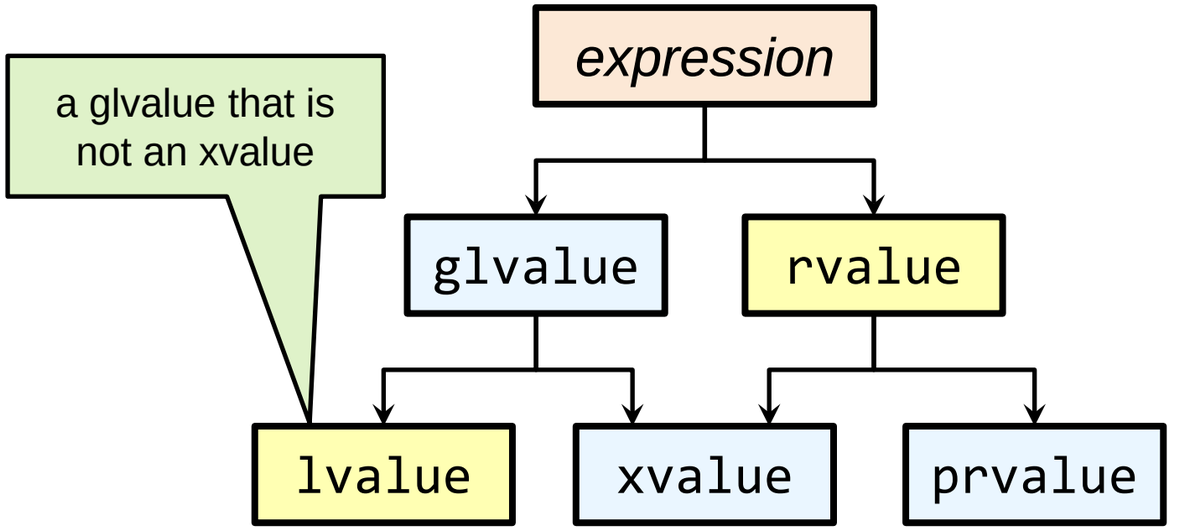
Structure of the Abstract Machine – Threads and Expressions

- Every C++ expression has an associated **type**
- Every C++ expression is a member of a **value category**
- **rvalues** indicate objects that are
  - Nameless
  - Temporary objects, literals, etc.
  - Cannot have their address taken
- **lvalues** indicate objects that
  - Have names, directly or indirectly
  - Are non-temporary
  - May have their address taken



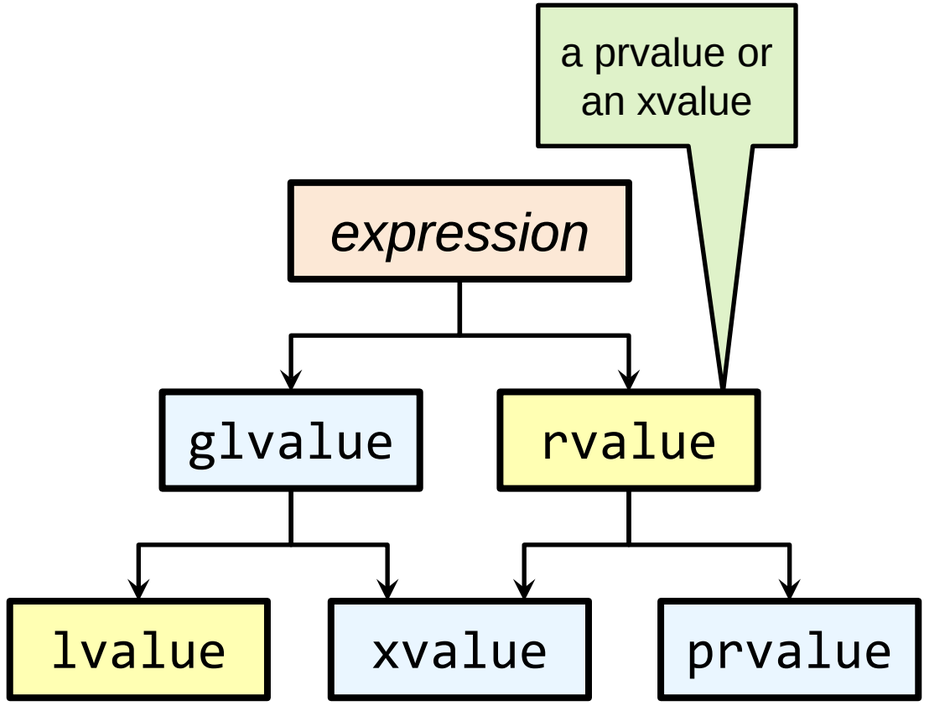
Structure of the Abstract Machine – Threads and Expressions

- Every C++ expression has an associated **type**
- Every C++ expression is a member of a **value category**
- **rvalues** indicate objects that are
  - Nameless
  - Temporary objects, literals, etc.
  - Cannot have their address taken
- **lvalues** indicate objects that
  - Have names, directly or indirectly
  - Are non-temporary
  - May have their address taken



Structure of the Abstract Machine – Threads and Expressions

- Every C++ expression has an associated **type**
- Every C++ expression is a member of a **value category**
- **rvalues** indicate objects that are
  - Nameless
  - Temporary objects, literals, etc.
  - Cannot have their address taken
- **lvalues** indicate objects that
  - Have names, directly or indirectly
  - Are non-temporary
  - May have their address taken



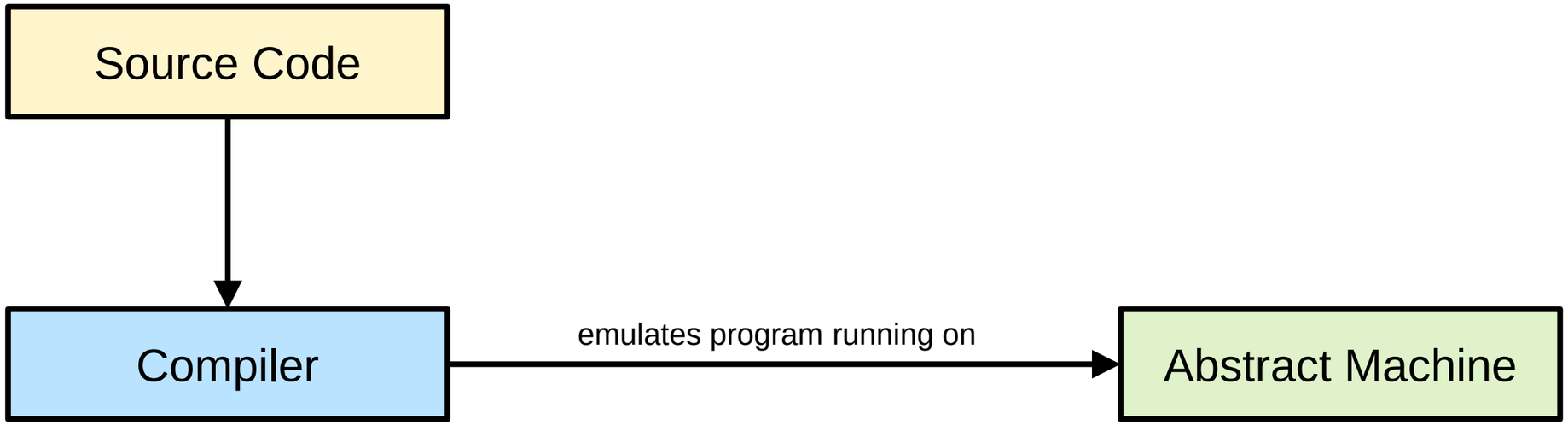
Structure of the Abstract Machine – Threads and Expressions

- Functions consist of statements, statements consist of expressions
- An expression is a sequence of operators and operands that specifies a computation
- Evaluating an expression that causes side effects results in changes to the program's execution state, and possibly to observable behavior
- The rules governing the evaluation of expression are formulated in terms of an expression's type and value category

Summary

- Our programs describe operations performed on the abstract machine

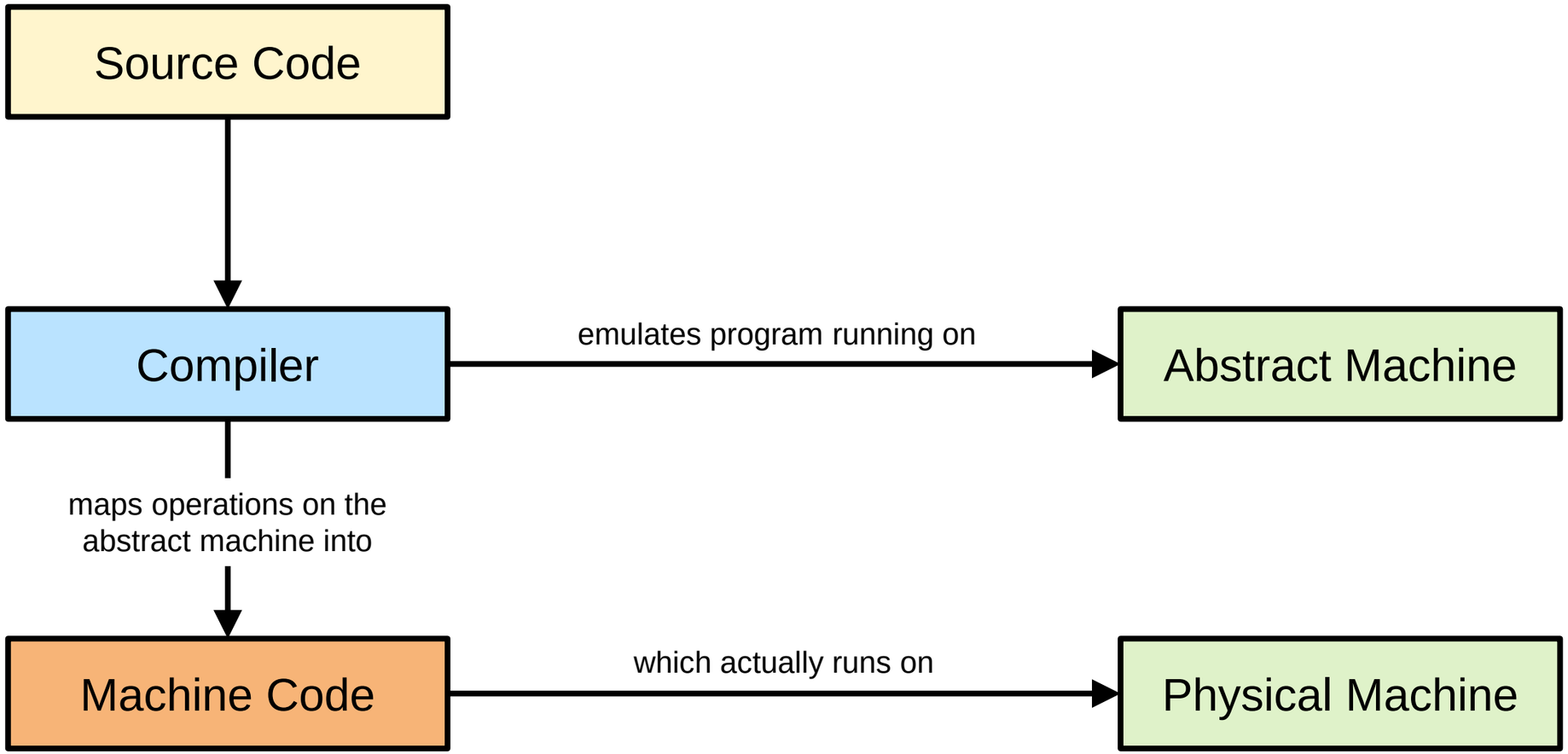
The Abstract Machine



Summary

- Our programs describe operations performed on the abstract machine
- Implementations translate abstract machine operations into physical machine operations

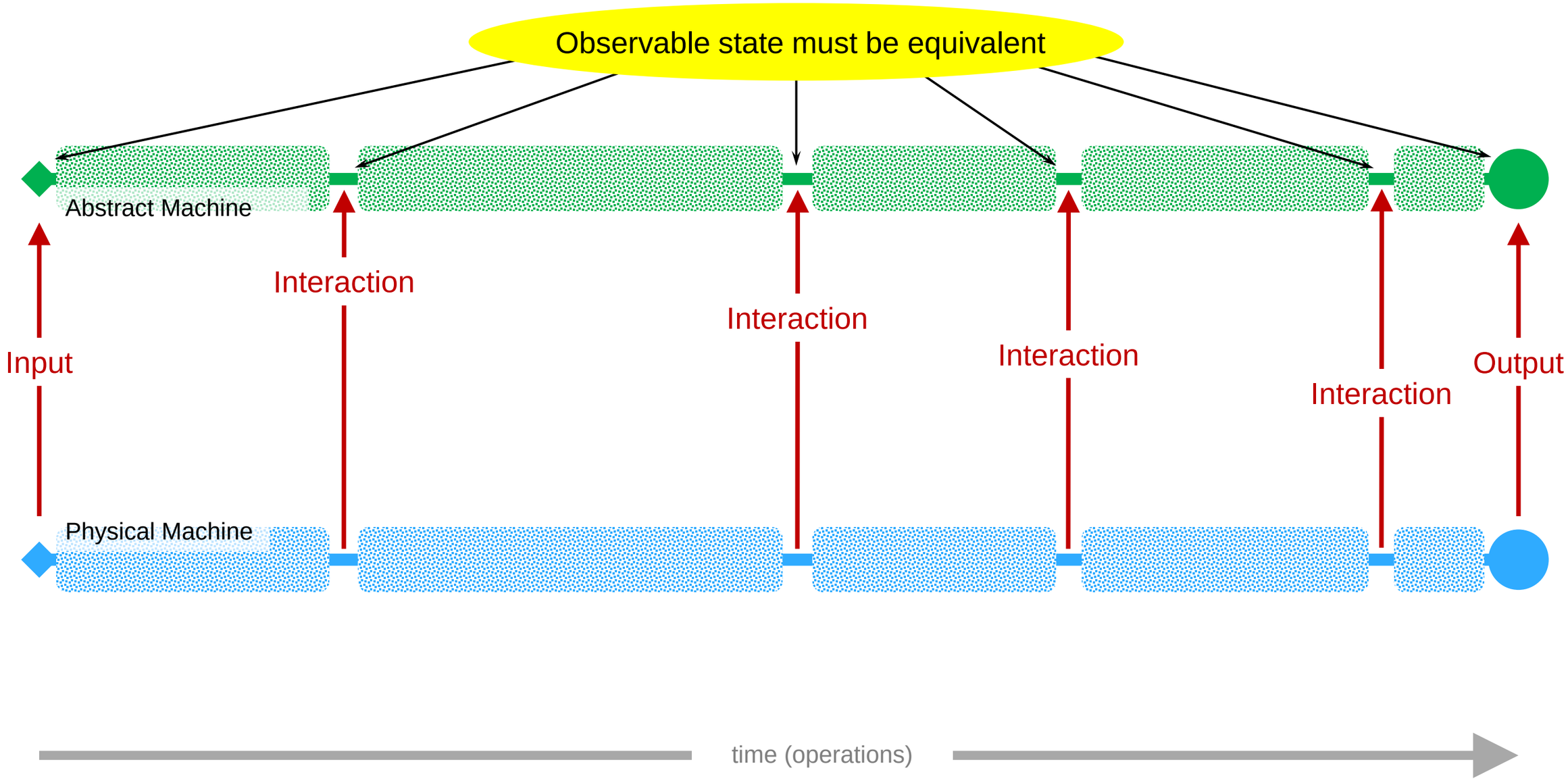
The Abstract Machine



Summary

- Our programs describe operations performed on the abstract machine
- Implementations translate abstract machine operations into physical machine operations
- Implementations must emulate the observable behavior of the abstract machine

The Abstract Machine

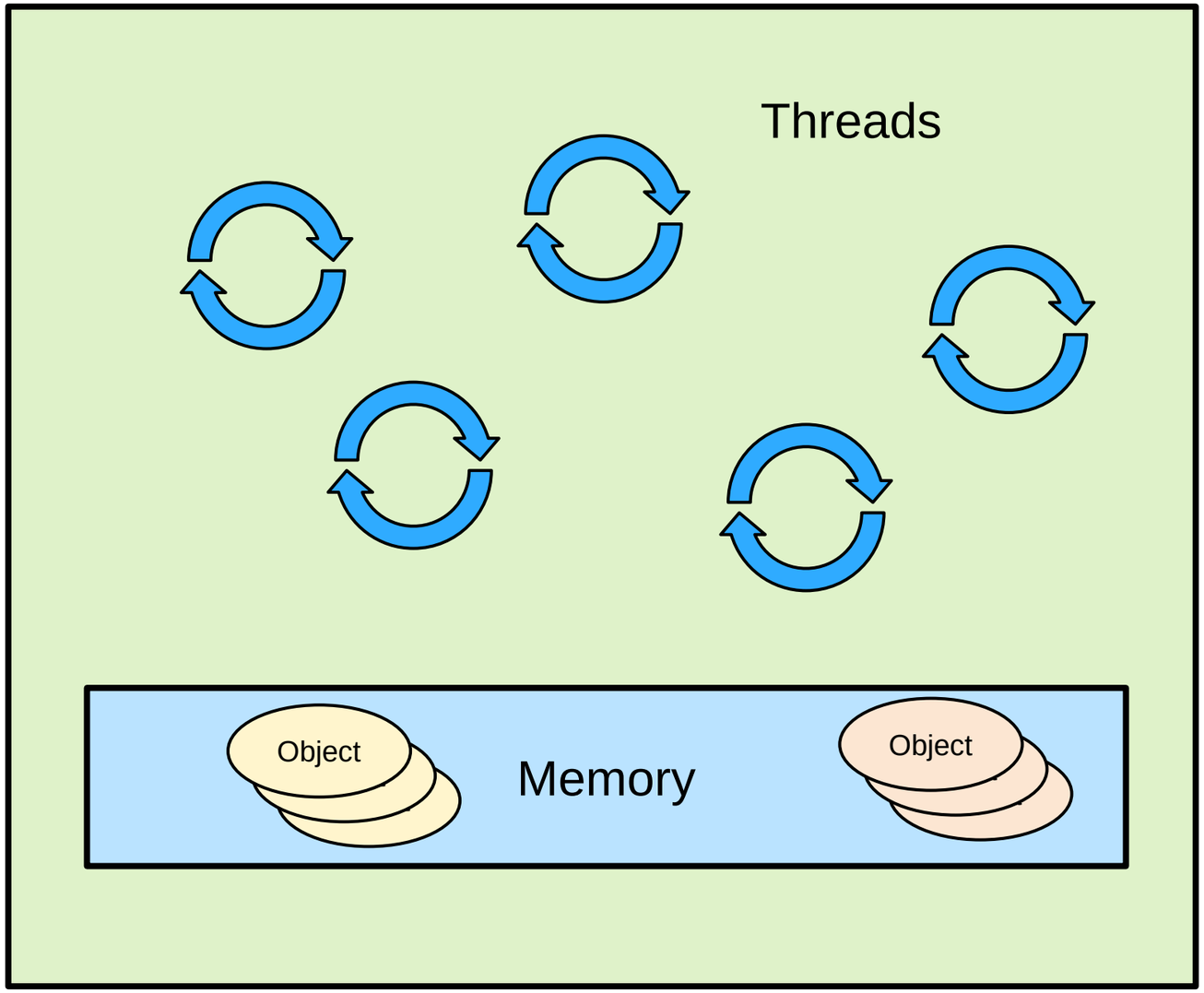


Summary

- Our programs describe operations performed on the abstract machine
- Implementations translate abstract machine operations into physical machine operations
- Implementations must emulate the observable behavior of the abstract machine
- The abstract machine has
  - Memory, which provides for storage
  - Objects, which for the most part, reside in the storage provided by memory
  - Threads, flows of control that carry out the operations specified by our program

The Abstract Machine

- Memory
- Objects
- Threads



Summary

When we write C++ code, we are writing to the C++ abstract machine

Thank You for Attending!

Talk: https://github.com/BobSteagall/CppCon2020

Blog: https://bobsteagall.com