Representing Data Collections for Analysis and Transformation

Tommy McMichen

March 8, 2024

Northwestern University
ARCANA Rethinks Compiler Abstractions for New Applications
memoir

A Brief History of Compilation.

A Brief History of Compilation.

Motivation.
The problem with compilers.

A Brief History of Compilation.

Motivation.
The problem with compilers.

How do we solve it?

A Brief History of Compilation.

Motivation.
The problem with compilers.

How do we solve it?

Feel free to ask questions at any time!

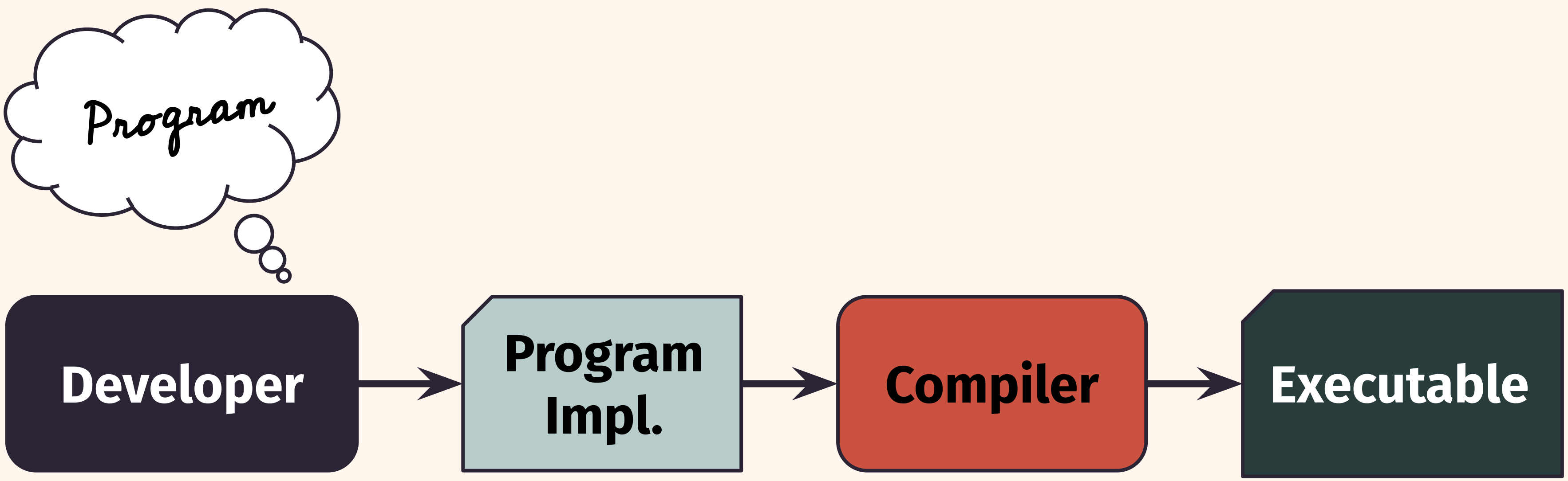
Compilation.

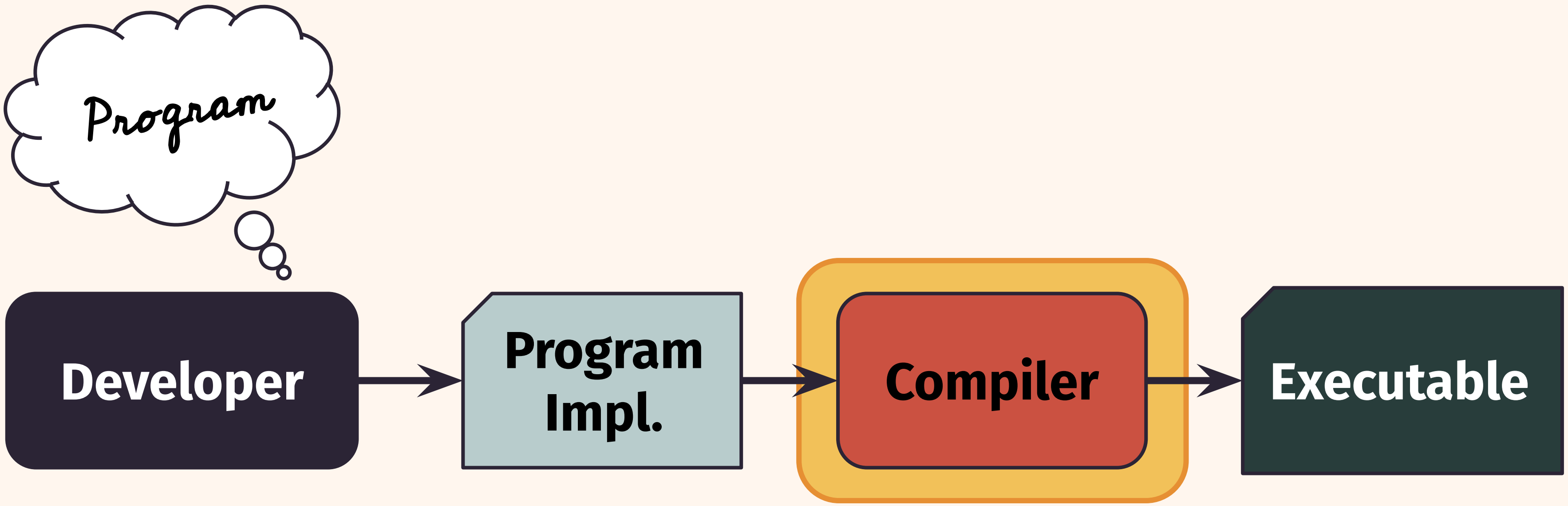
What is it?

Introduction

What is Compilation?

What is a *Pragmatic Definition* of Compilation?



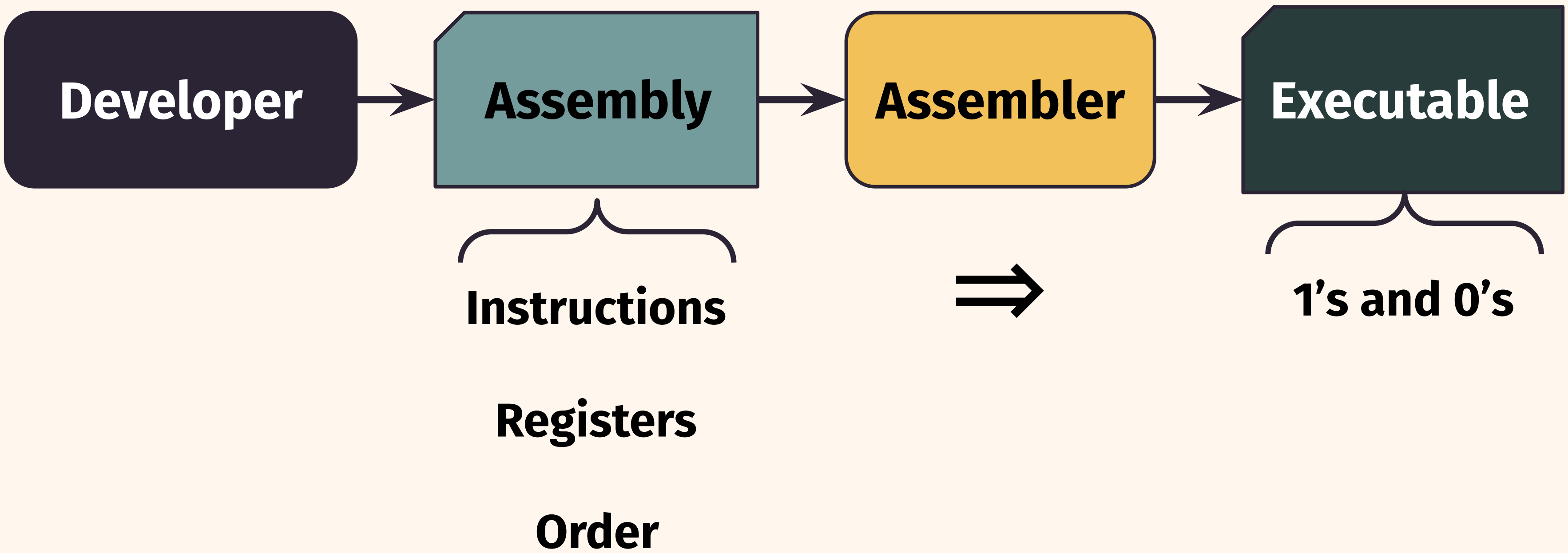


Compilers.

How do they work?

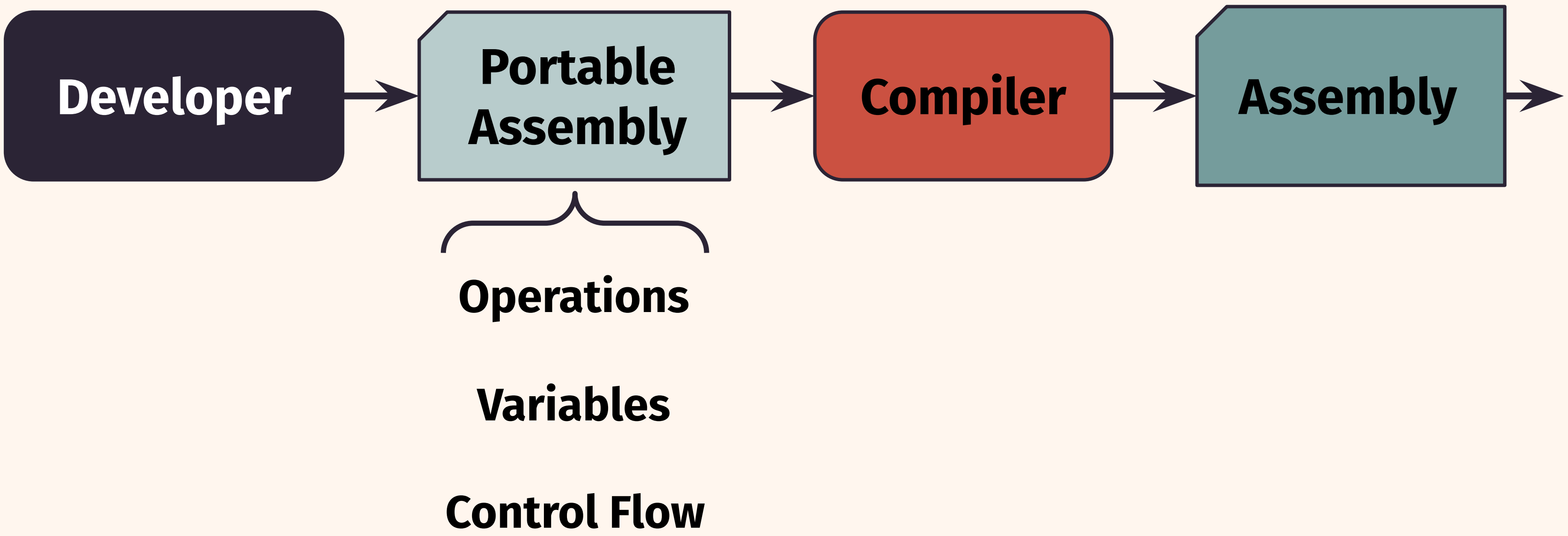
History

Manual Compilation



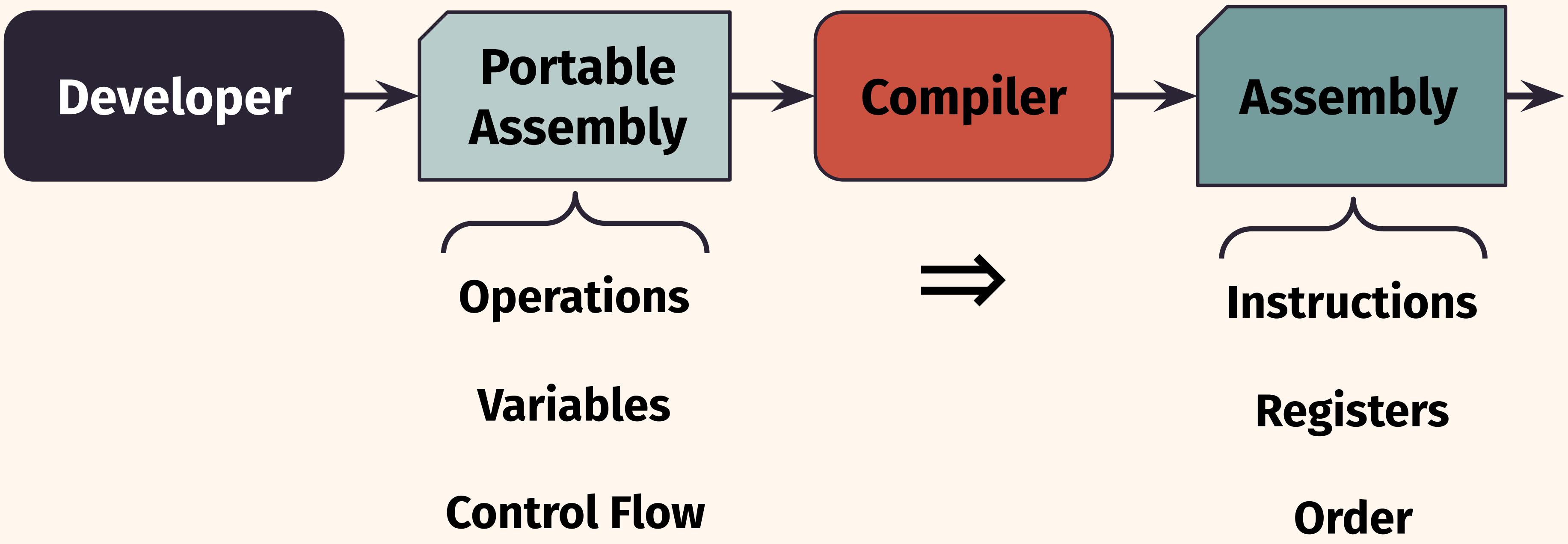
History

The Compiler as a *Translator*



History

The Compiler as a *Translator*



History

The Compiler as a *Translator*

Operations ⇒ Machine Instructions

The Compiler as a *Translator*

Operations ⇒ Machine Instructions

Instruction Selection

```
i = i + 1      ⟶      r1 ← add r1, #1
```

History

The Compiler as a *Translator*

Unlimited # of Variables ⇒ Limited Set of Registers

The Compiler as a *Translator*

Unlimited # of Variables ⇒ Limited Set of Registers

Register Allocation

```
a = 1              r1 ← #1
b = 2      ⟶       r2 ← #2
c = a + b          r3 ← add r1, r2
```

History

The Compiler as a *Translator*

Control Flow ⇒ Program Order

History

The Compiler as a *Translator*

Control Flow ⇒ Program Order

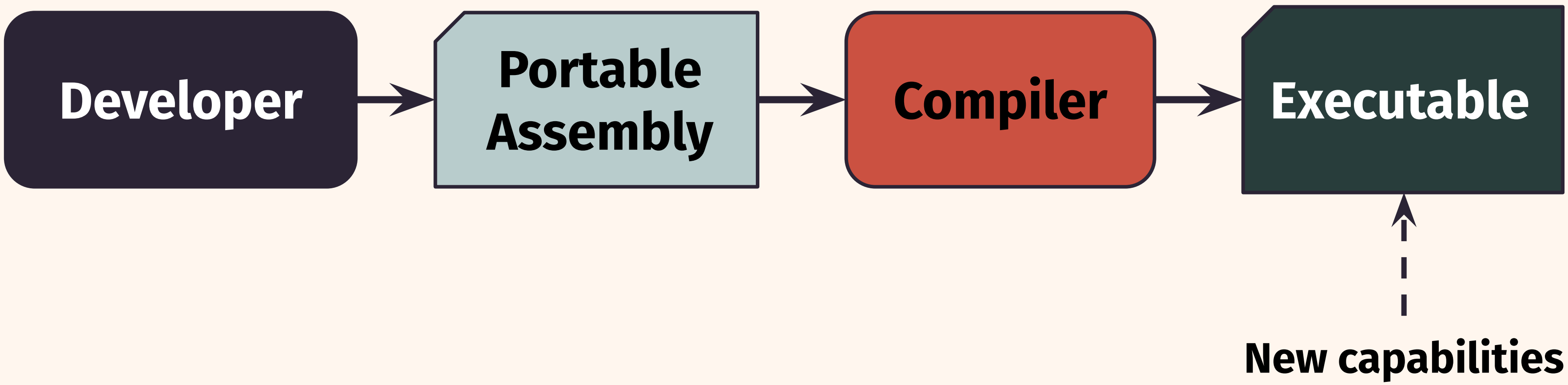
Linearization

```
if ? {   }
else {   }
return
```

```
      if ? goto true

      goto continue
true:
done: return
```

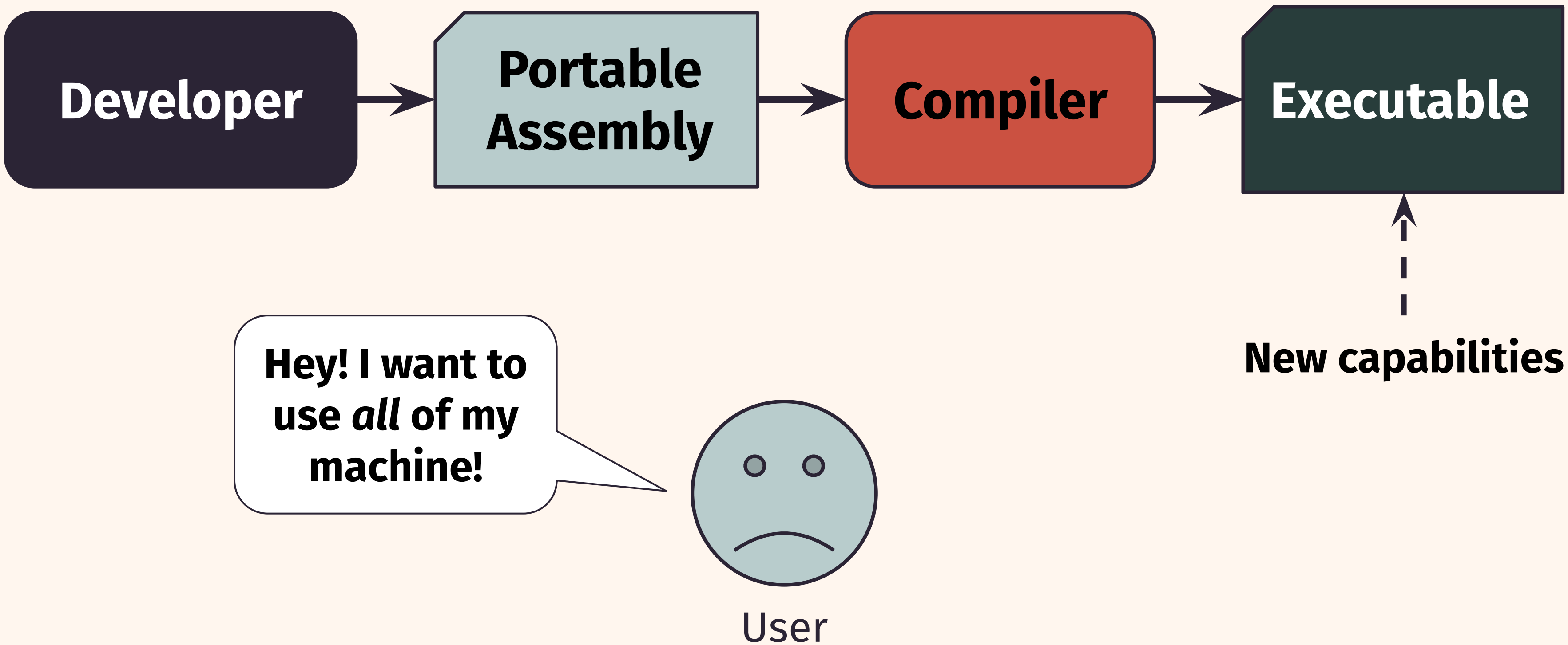
History

Machine *Capabilities* Improve



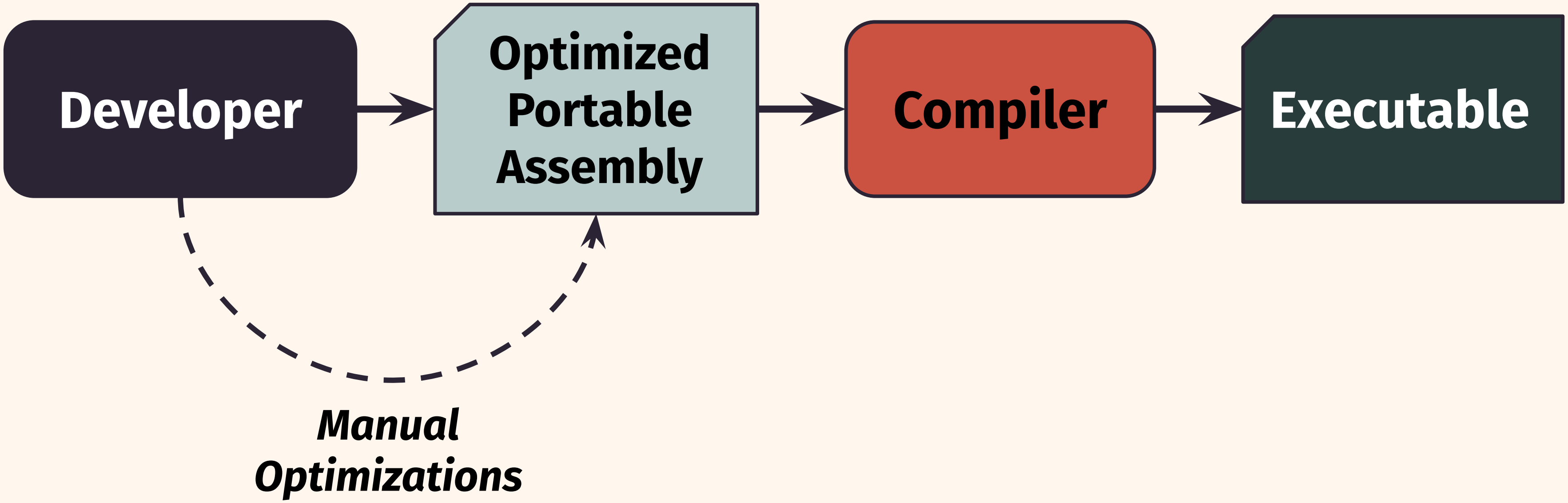
History

...and Users want to *Fully Utilize Machines*

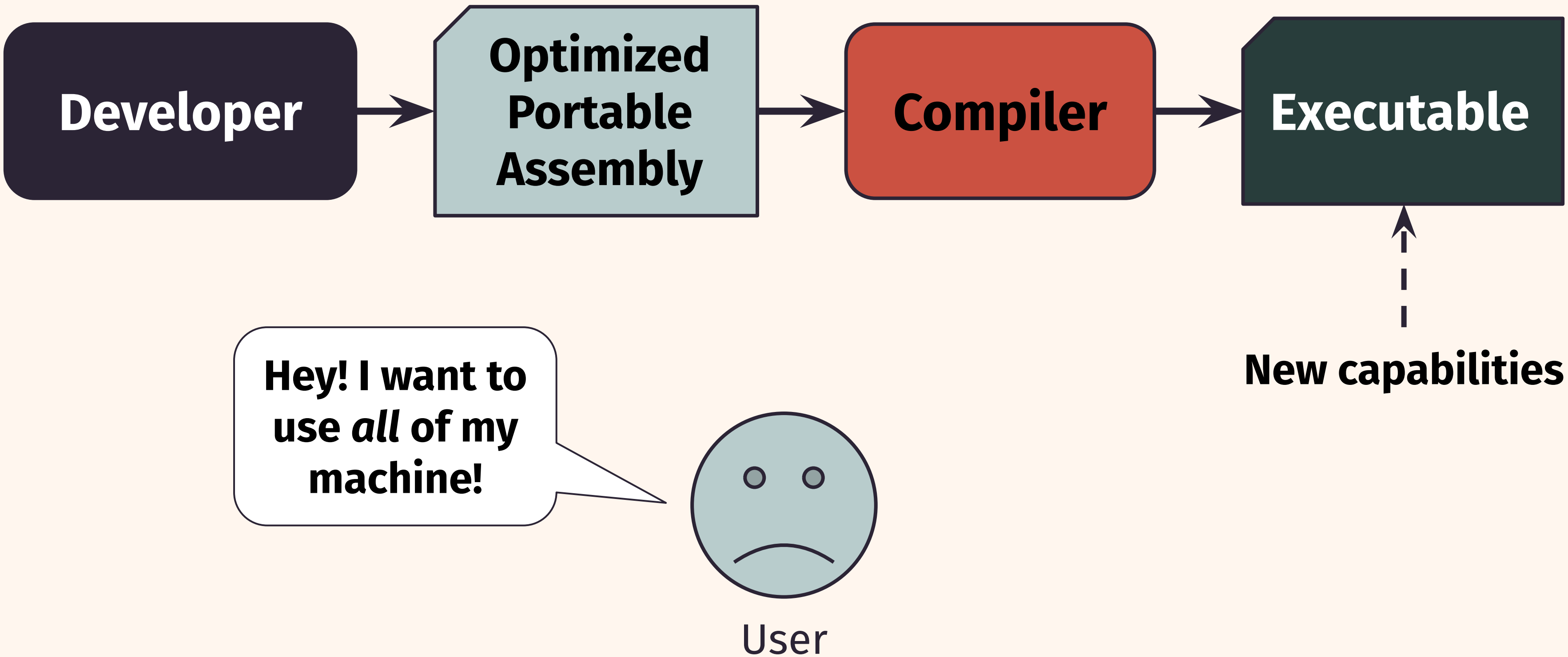


History

So Developers Oblige.

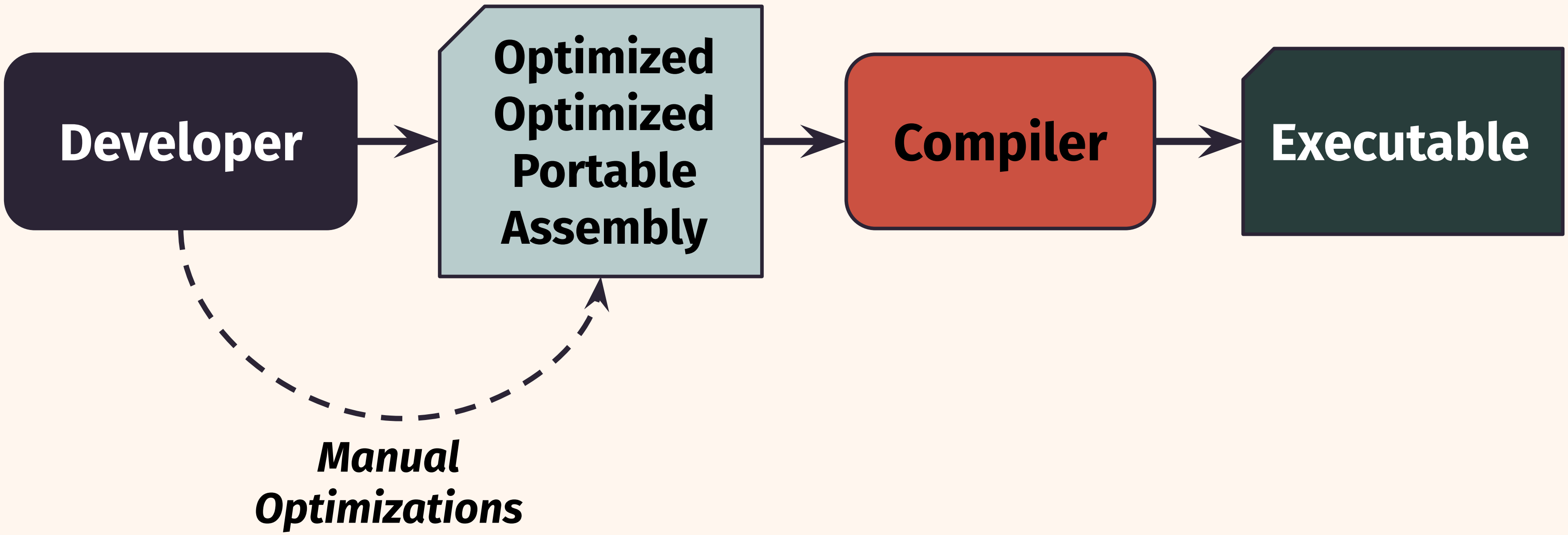


Machine Capabilities are a *Moving Target*



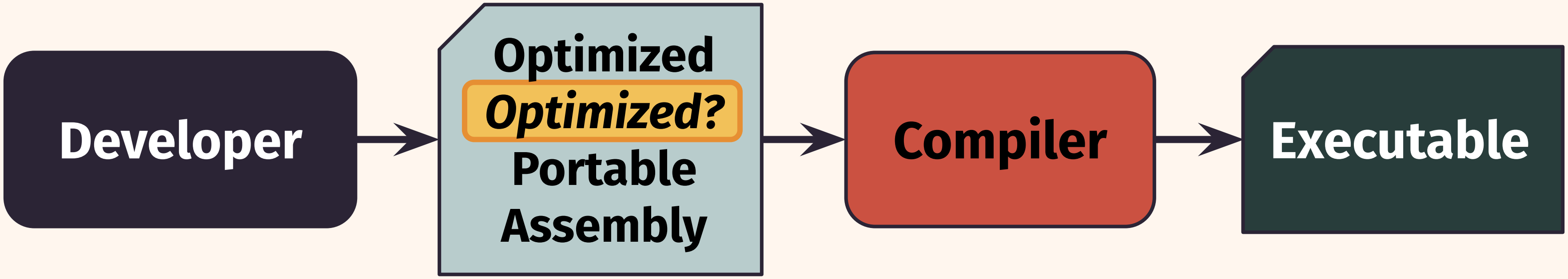
History

…so Developers oblige, again…



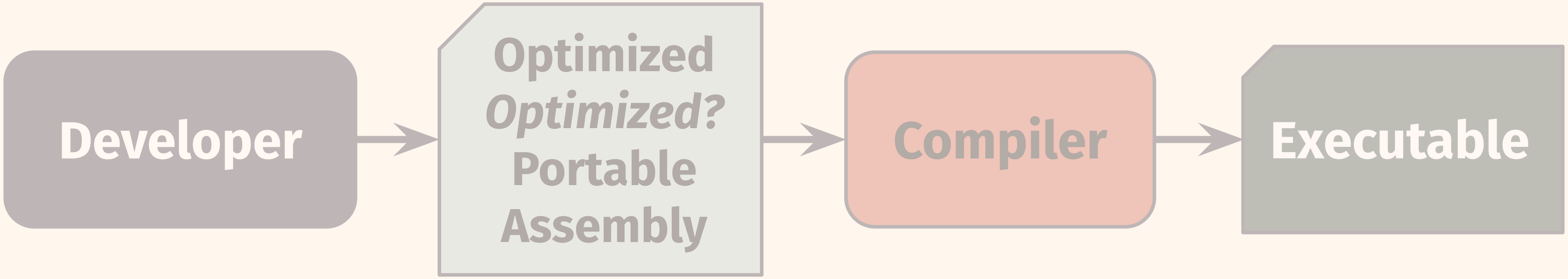
History

...but optimizations are not necessarily composable.



History

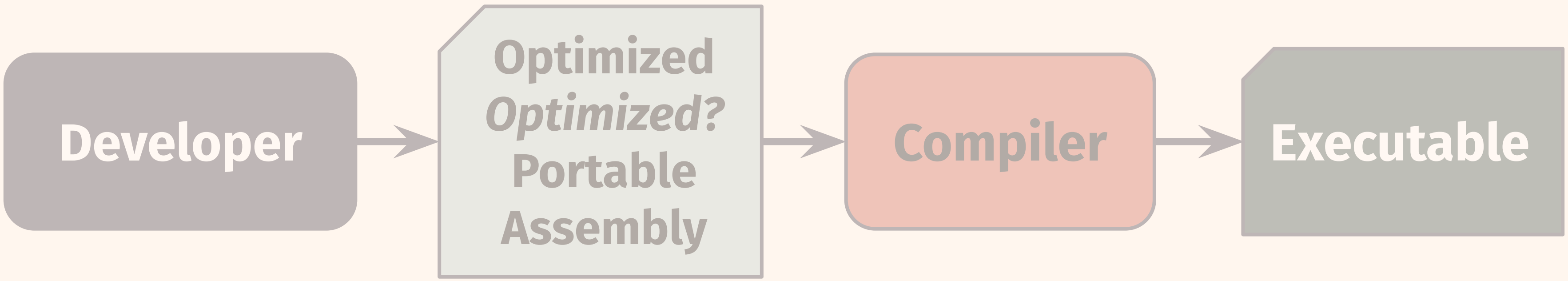
...and Manual Optimizations make Programs Unmaintainable!



Today's *optimized program* becomes tomorrow's *unmaintainable program*

History

...and Manual Optimizations increase Engineering Costs!

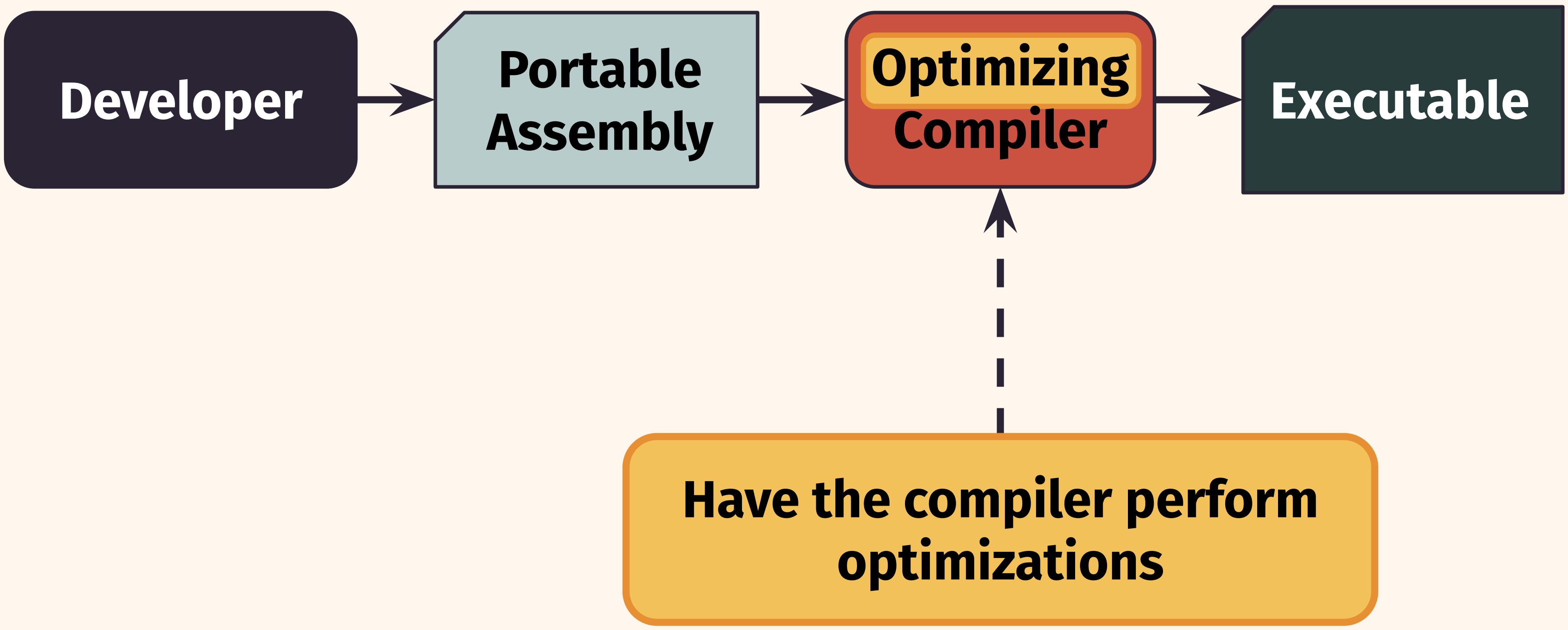


Today's *optimized program* becomes tomorrow's *unmaintainable program*

Maintenance is the *true cost* of software.
Bug fixes, security patches, platform updates, etc.

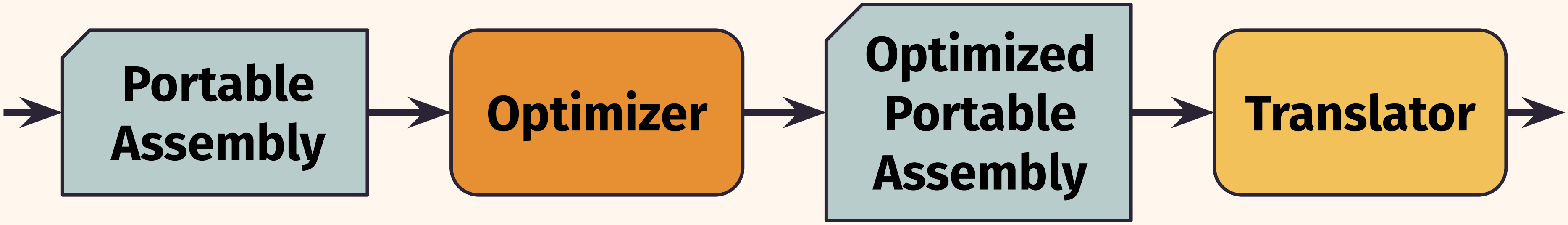
History

How do we fix this?



History

The Compiler as an *Optimizer*



The Compiler as an *Optimizer*

Constant Folding

```
a = 1
b = 2
c = a + b
```

$\longrightarrow$

```
c = 3
```

The Compiler as an *Optimizer*

Constant Folding

```
a = 1
b = 2              →  c = 3
c = a + b
```

Instruction Combining

```
i = i + 1          →  i = i + 2
i = i + 1
```

History

The Compiler as an *Optimizer*

Constant Folding

```
a = 1
b = 2          ──────►  c = 3
c = a + b
```

Instruction Combining

```
i = i + 1      ──────►  i = i + 2
i = i + 1
```

Dead Code Elimination

```
x = ...                 x = ...
y = x + 2      ──────►  return x
return x
```

History

The Compiler as an *Optimizer*

Constant Folding

```
a = 1
b = 2
c = a + b
```

→

```
c = 3
```

Instruction Combining

```
i = i + 1
i = i + 1
```

→

```
i = i + 2
```

Dead Code Elimination

```
x = ...
y = x + 2
return x
```

→

```
x = ...
return x
```

There are 100s of optimization passes in LLVM!

Optimizations Make Software Engineering Easier!

```
Dev's Code
a = ALICE_NUM
b = BOB_NUM
c = a + b
d = c + 1
```

Single point of control

History

Optimizations Make Software Engineering Easier!

Dev's Code

```
a = ALICE_NUM
b = BOB_NUM
c = a + b
d = c + 1
```

→

```
a = 1
b = 2
c = a + b
d = c + 1
```

→

```
c = 3
d = c + 1
```

→

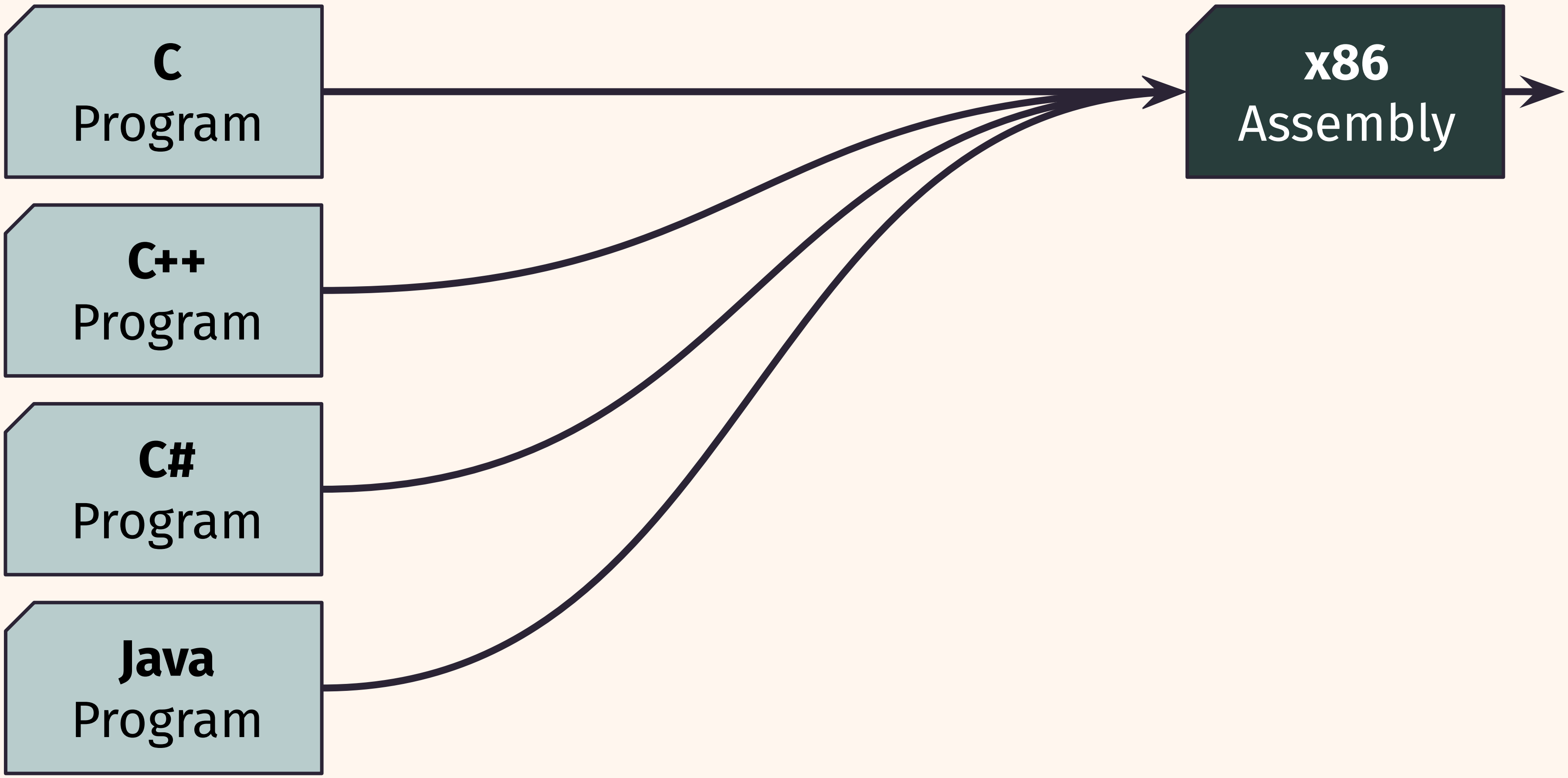
```
d = 4
```

Single point of control

History

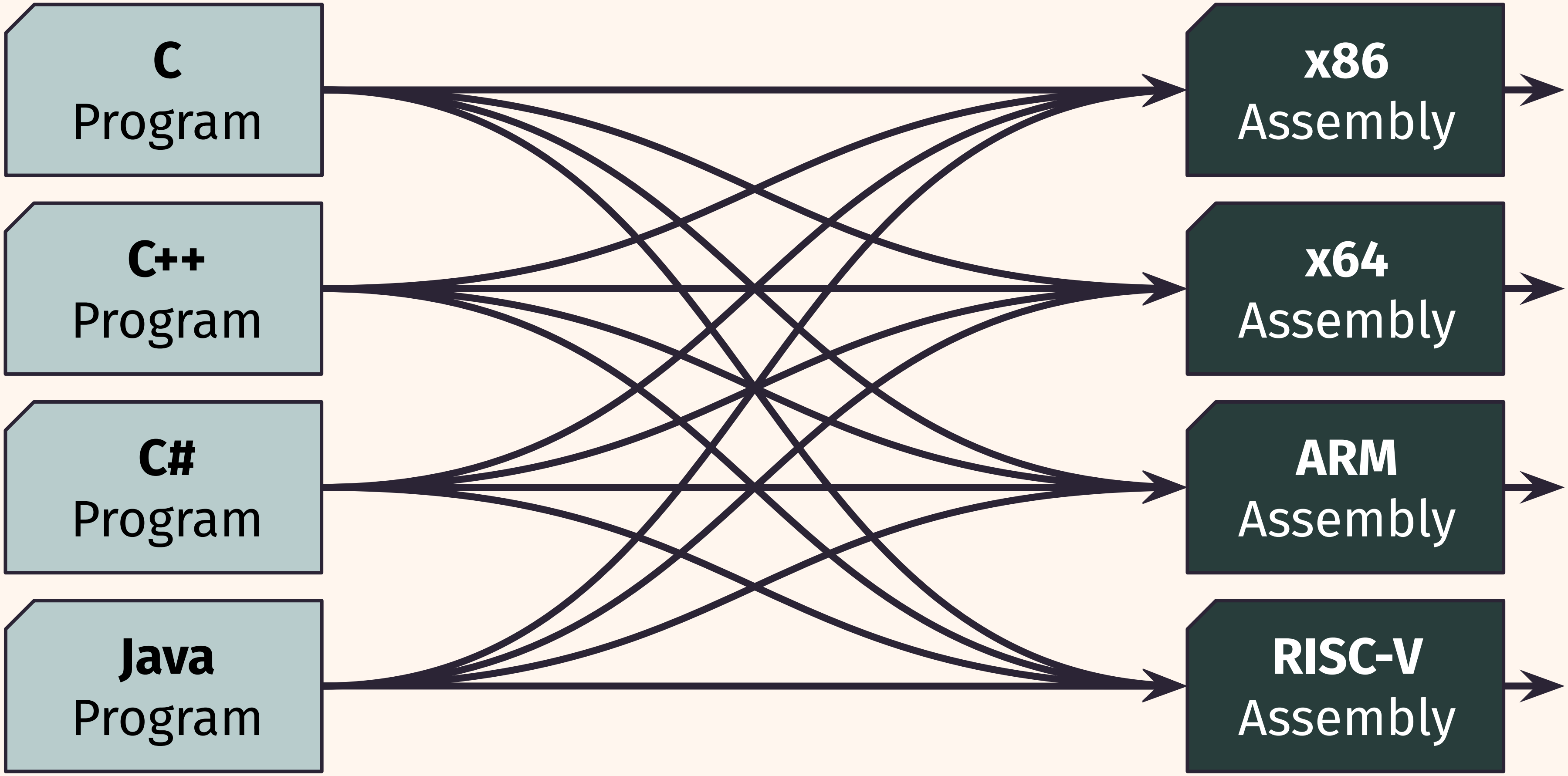
The Compiler as an *Engineering Nightmare*

The Compiler as an *Engineering Nightmare*



… P_n

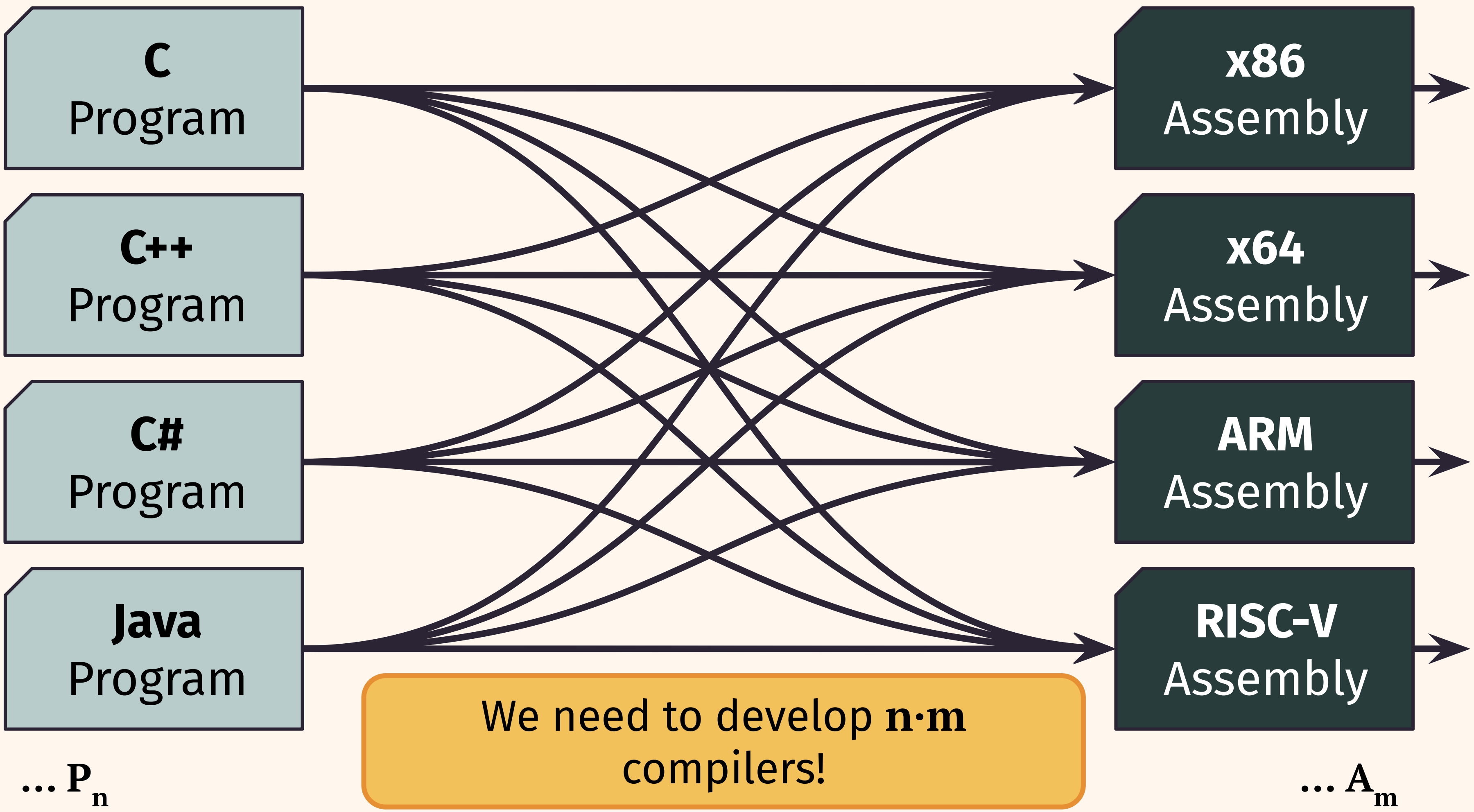
The Compiler as an *Engineering Nightmare*



… P_n … A_m

History

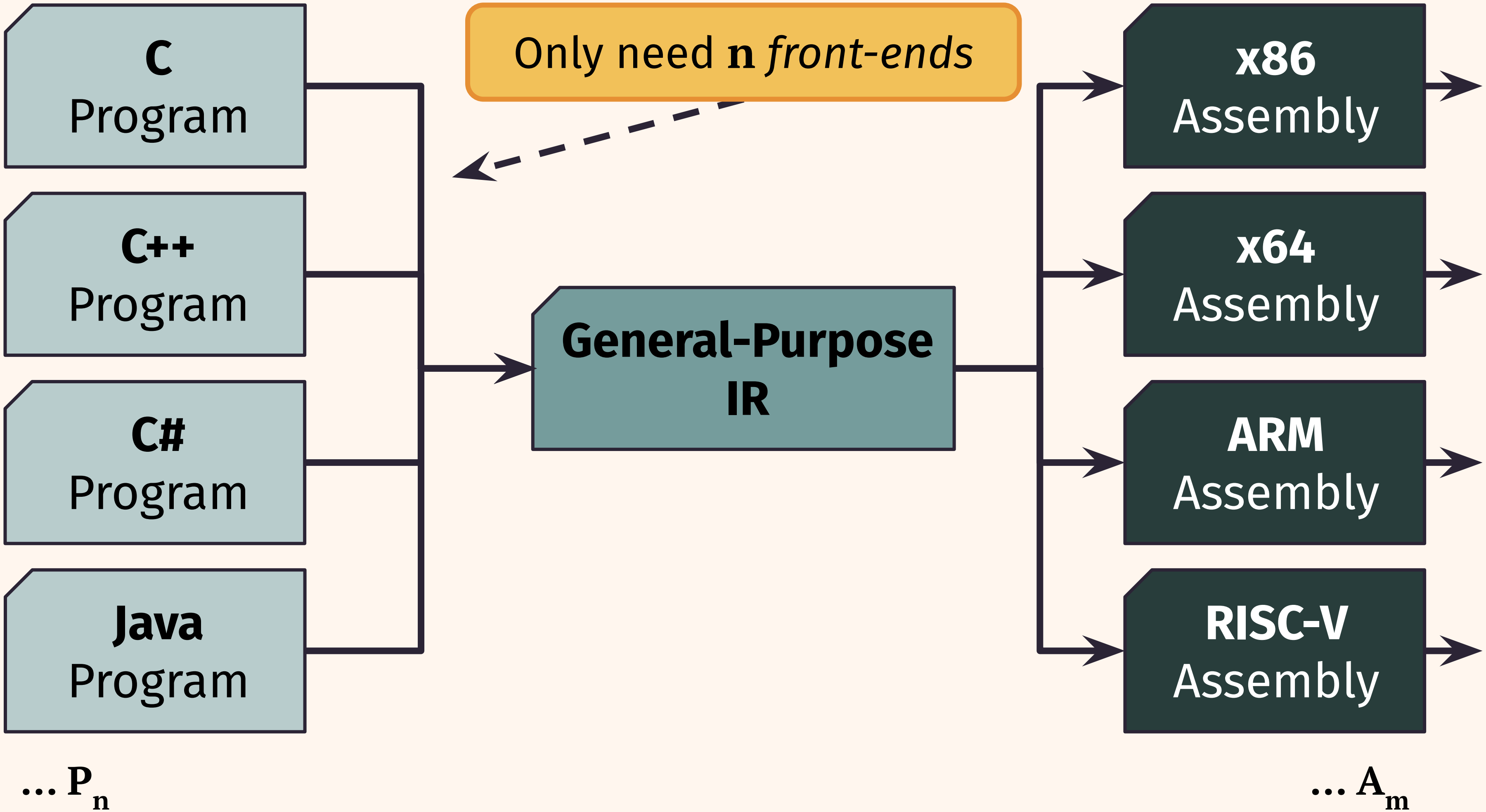
The Compiler as an *Engineering Nightmare*



We need to develop $\mathbf{n \cdot m}$ compilers!

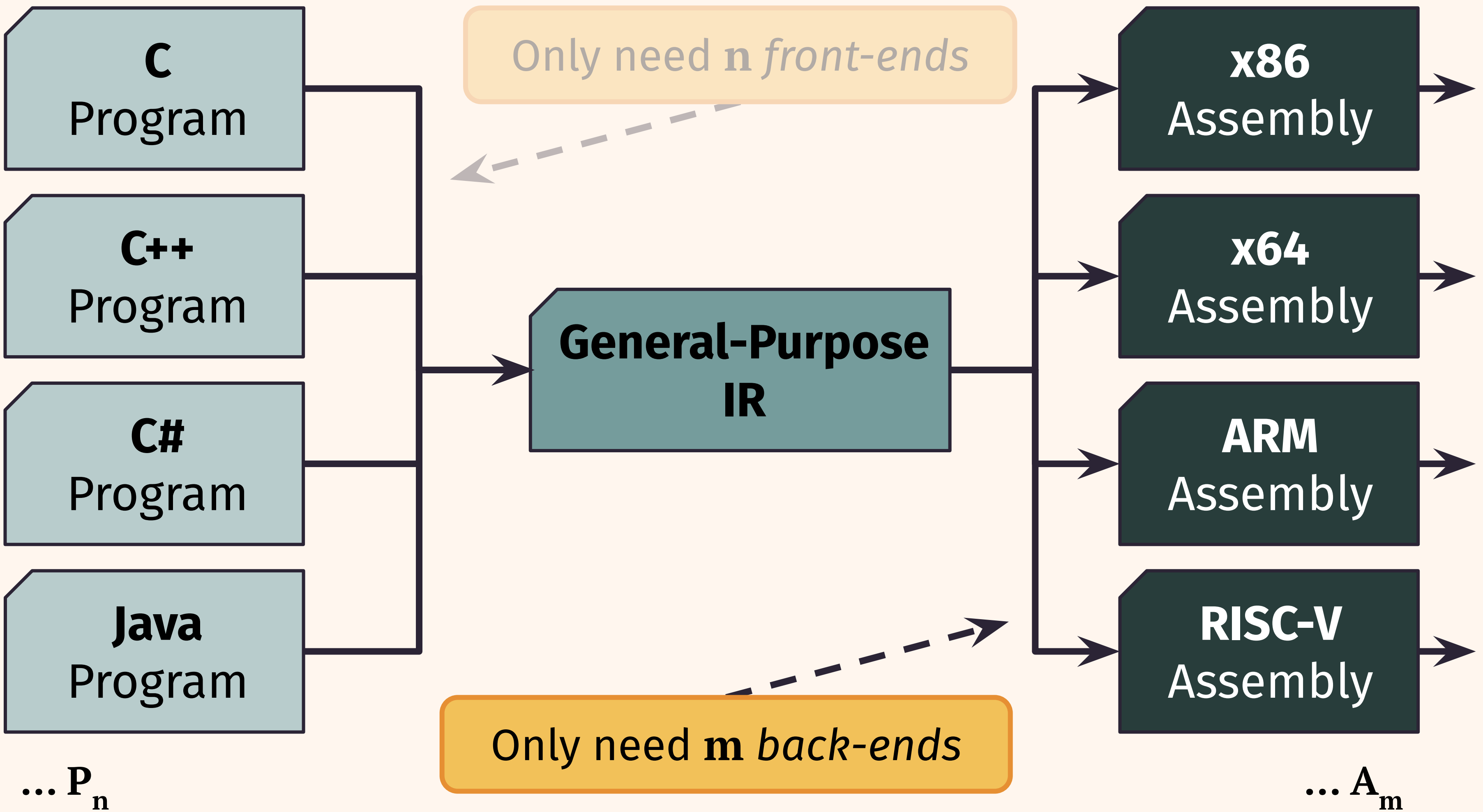
History

The Compiler as a *Bridge*



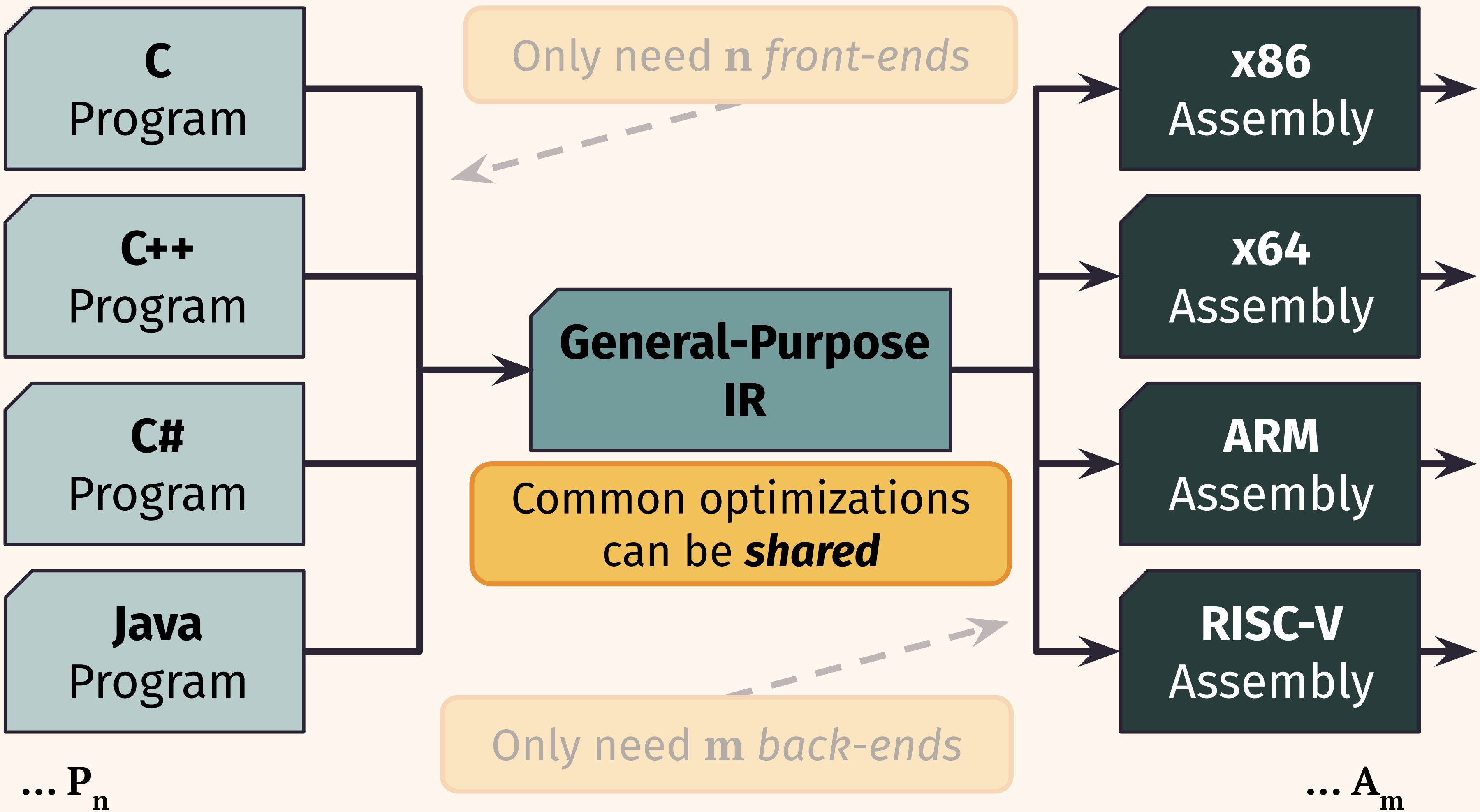
History

The Compiler as a *Bridge*



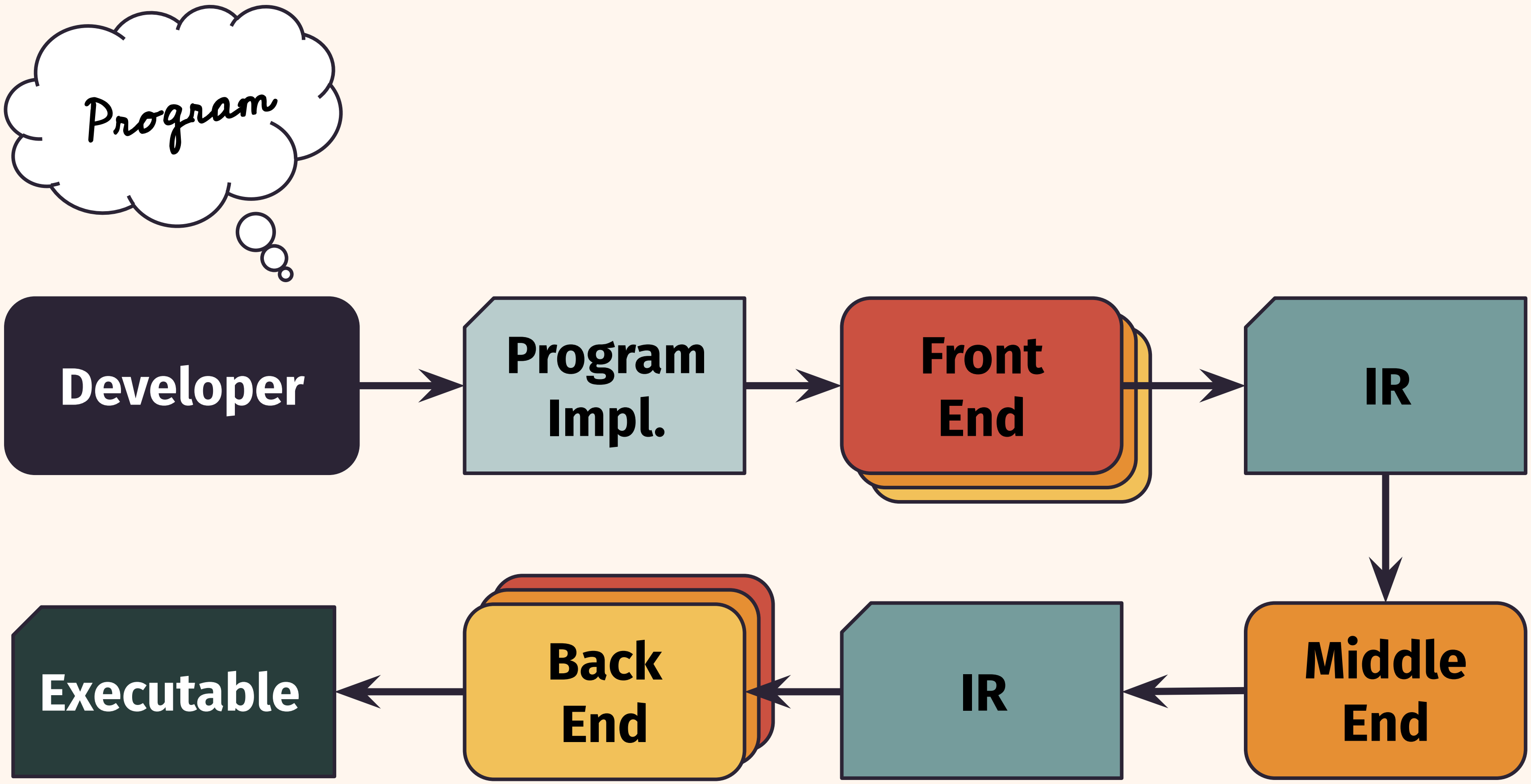
History

The Compiler as a *Bridge*



History

What is a *Modern Definition* of Compilation?



So, what's the modern problem?

Premature Lowering.

So, what's the modern problem?

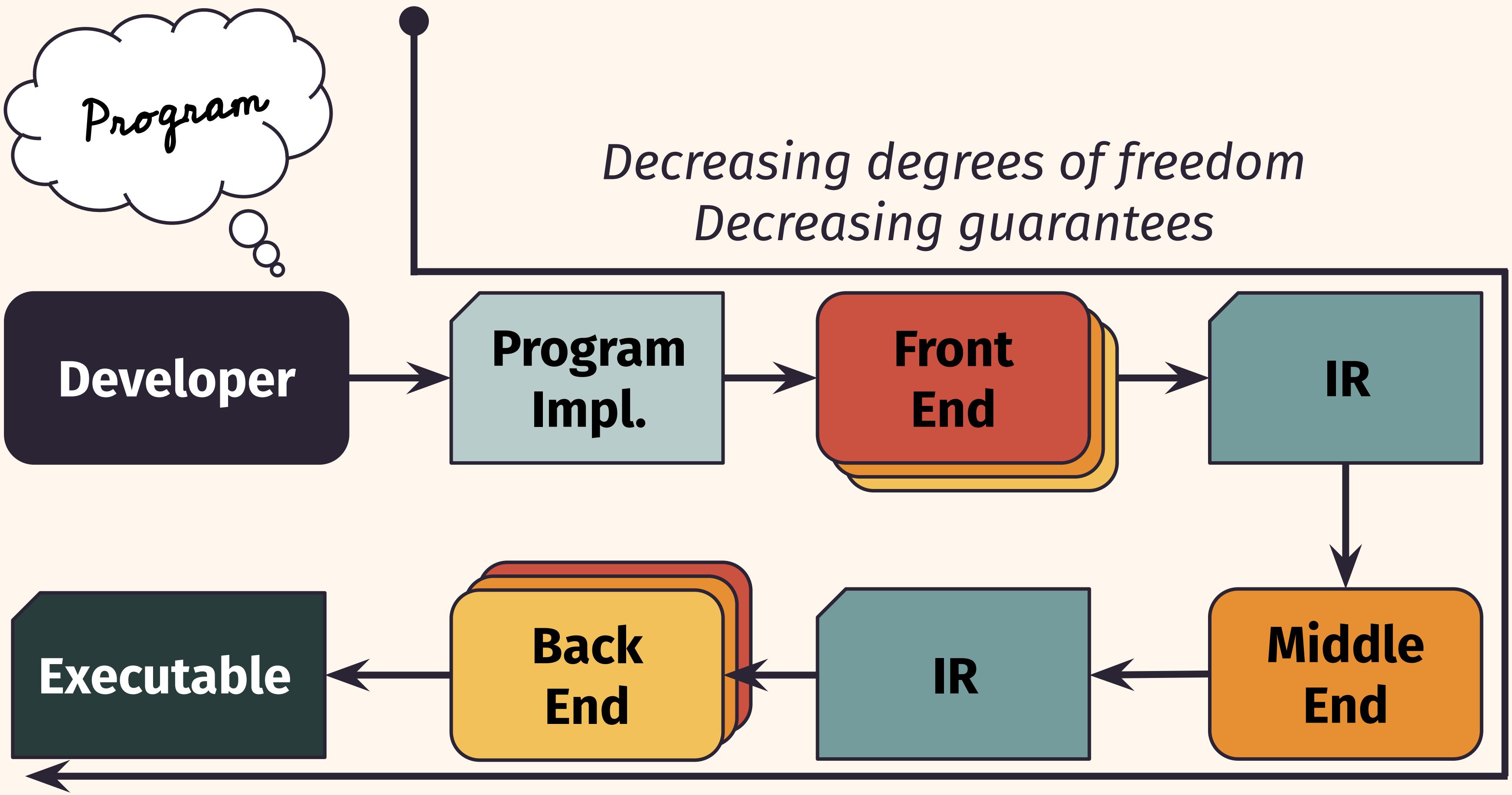
Premature Lowering.

Lowering:

Lowering is the ***destruction of high-level information*** by the ***instantiation of low-level decisions***.

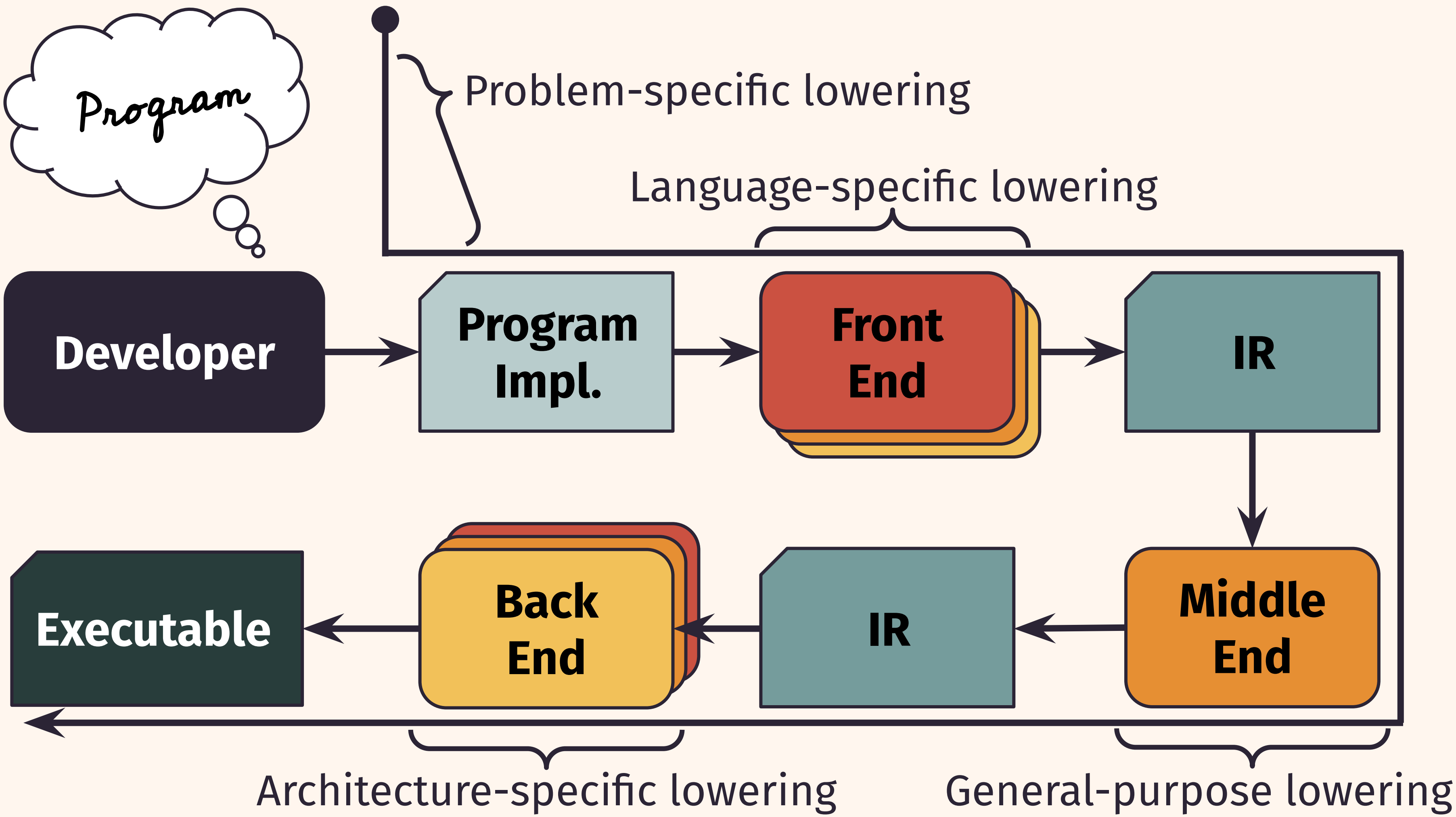
Motivation

What is *Lowering*?



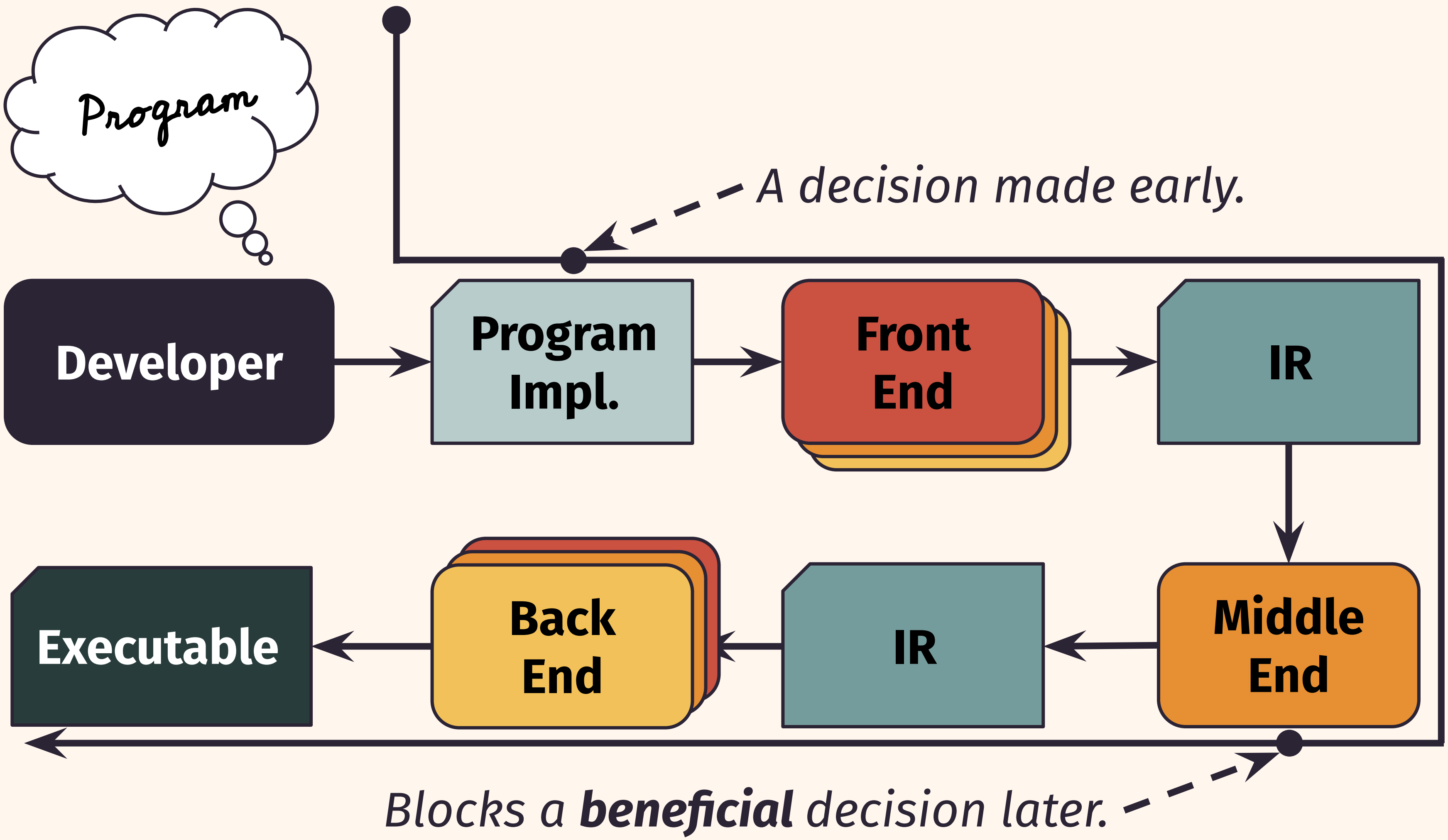
Motivation

What Kinds of Lowerings Exist?



Motivation

What is *Premature Lowering*?



An instance of premature lowering:

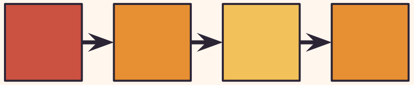
Data Collections.

Data Collection:

A Logical Organization of Data.

Motivation

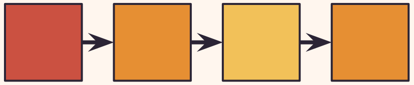
Data Collections: *Logical Organization of Data*



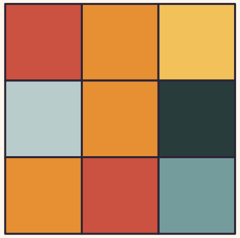
List

Motivation

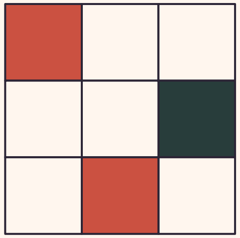
Data Collections: *Logical Organization of Data*



List



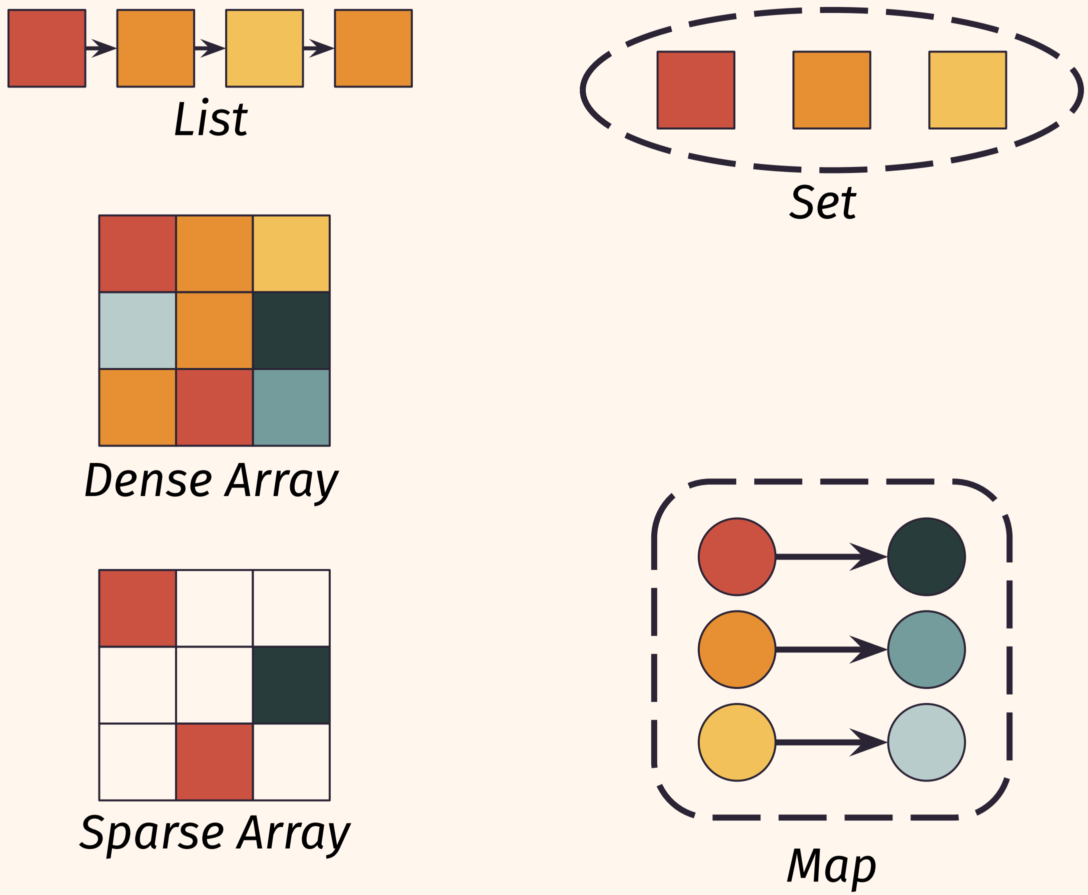
Dense Array



Sparse Array

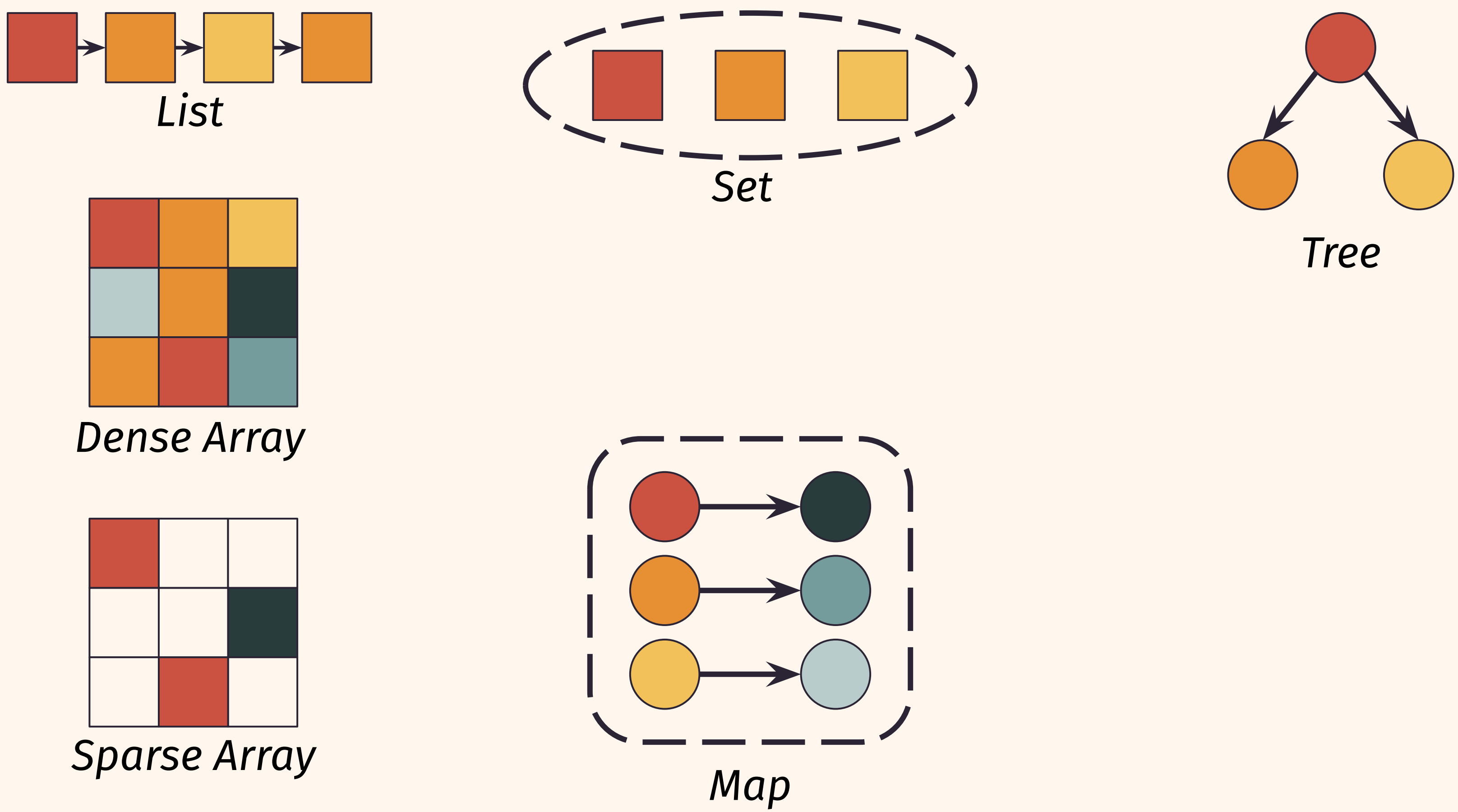
Motivation

Data Collections: *Logical Organization of Data*



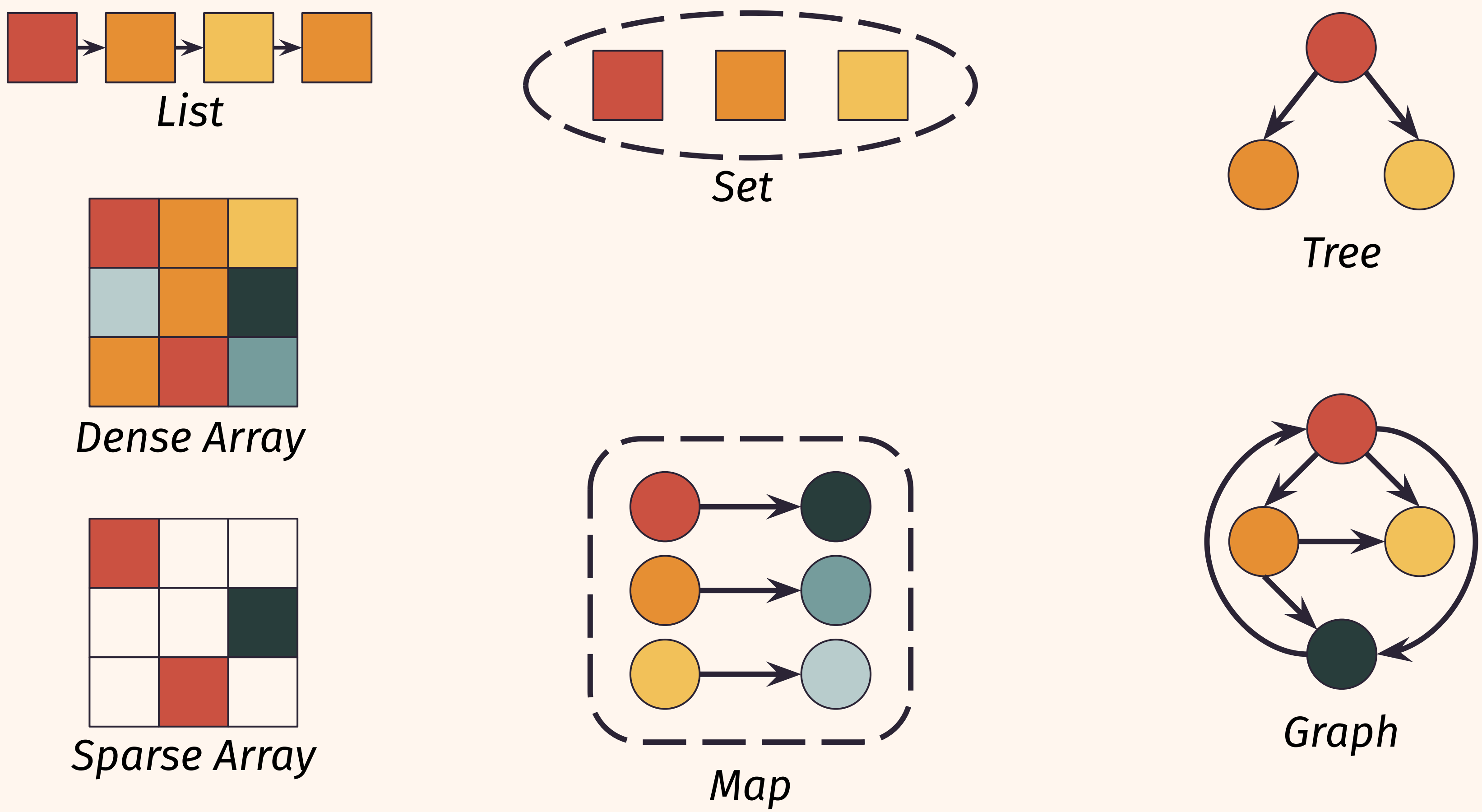
Motivation

Data Collections: *Logical Organization of Data*



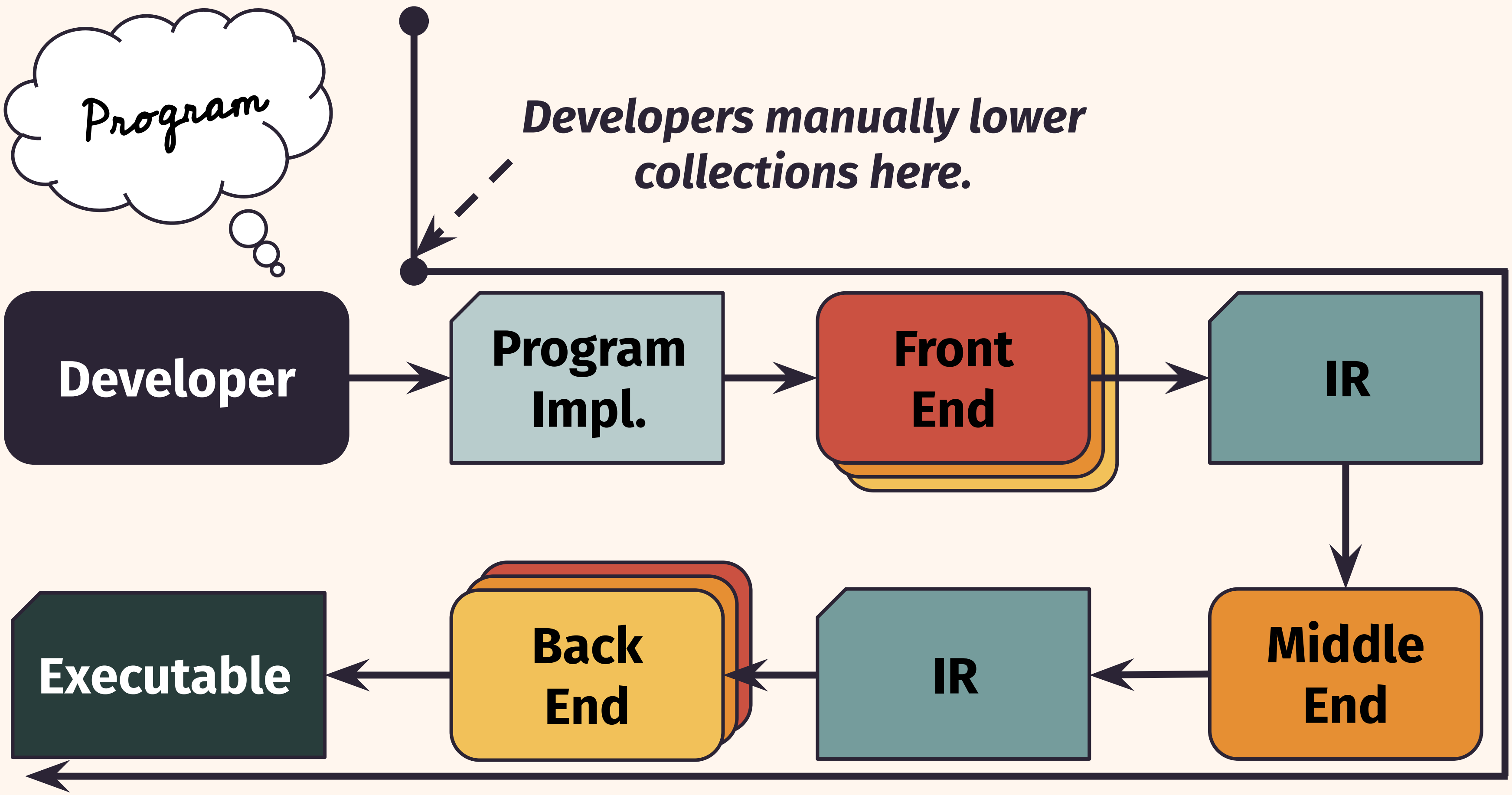
Motivation

Data Collections: *Logical Organization of Data*



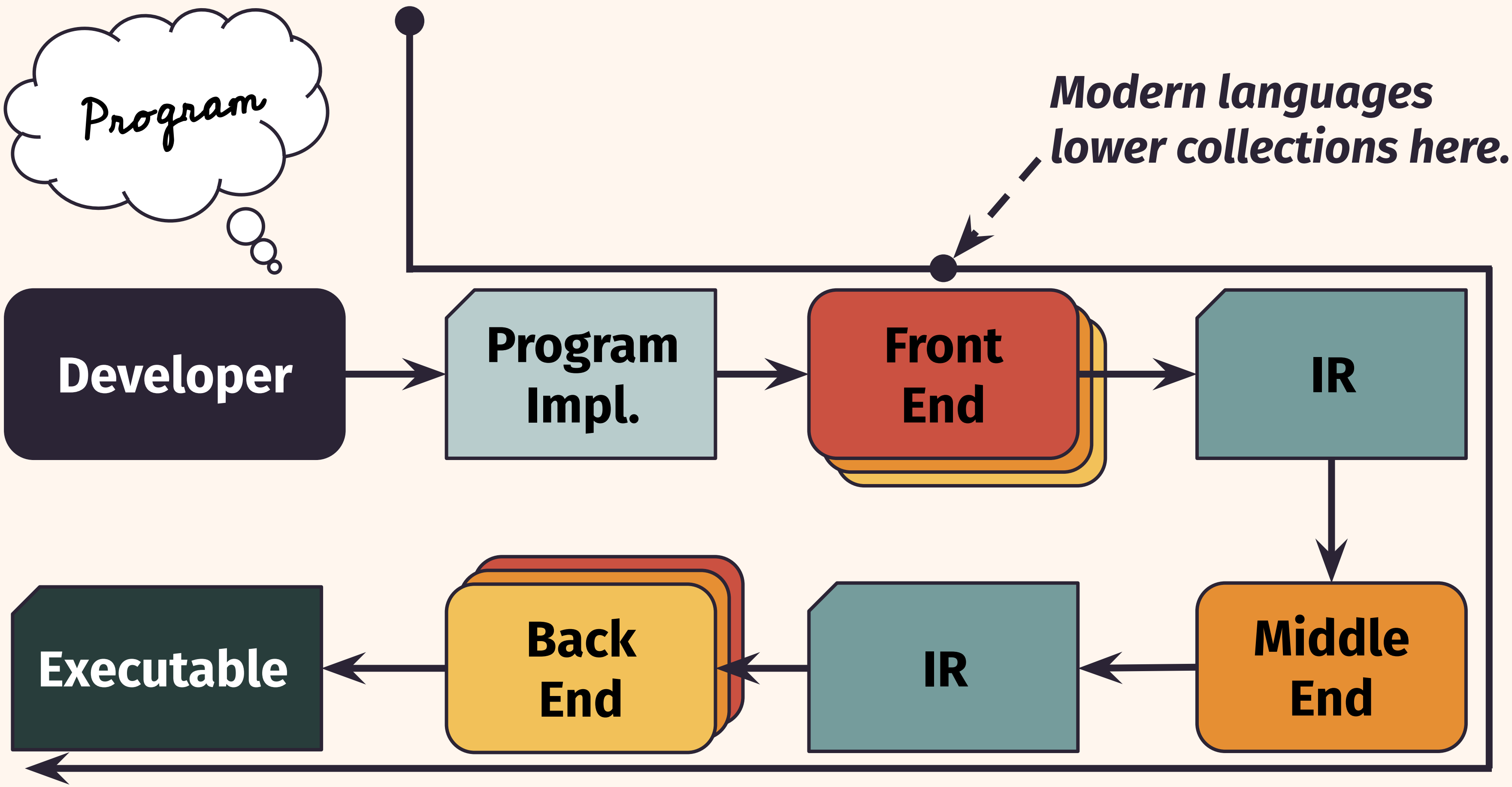
Motivation

How are Collections *Lowered?*



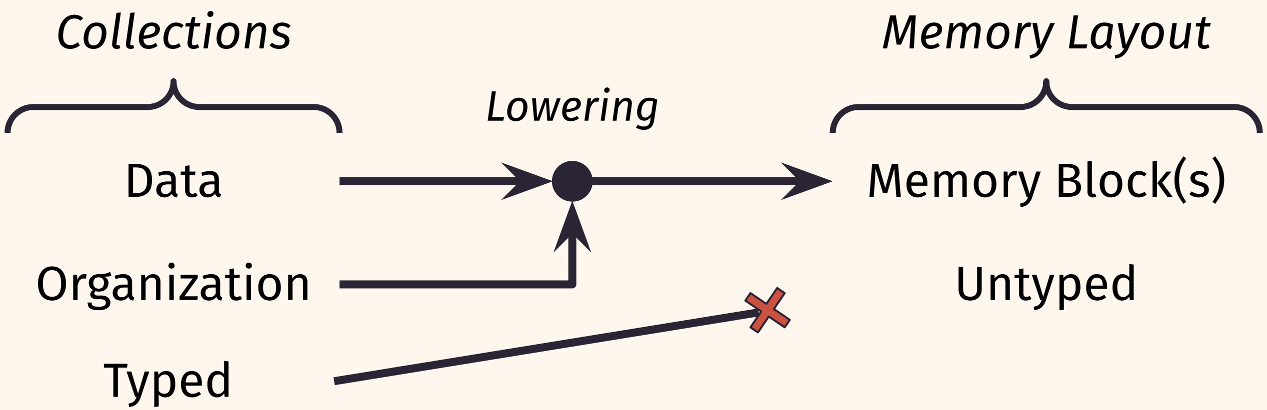
Motivation

How are Collections *Lowered?*



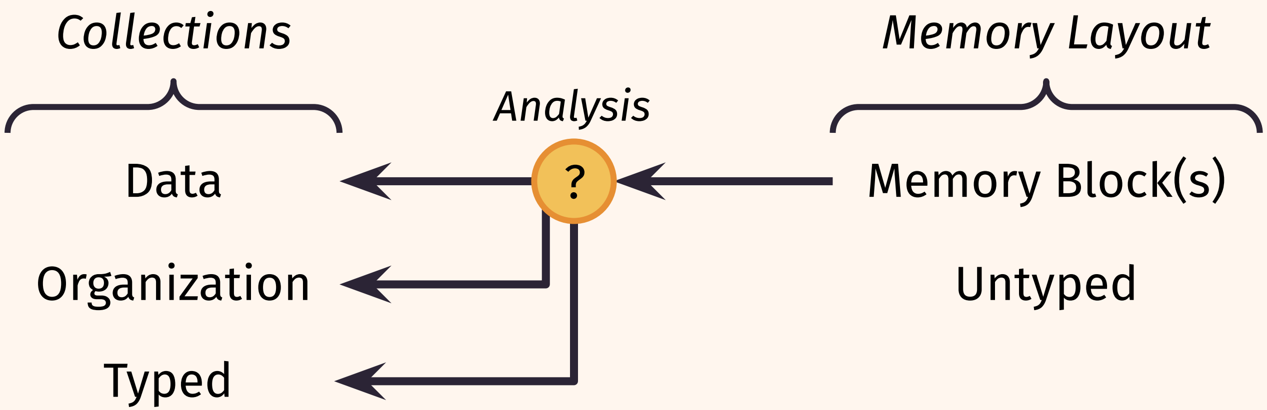
Motivation

Collections are lowered to their *in-memory layout*



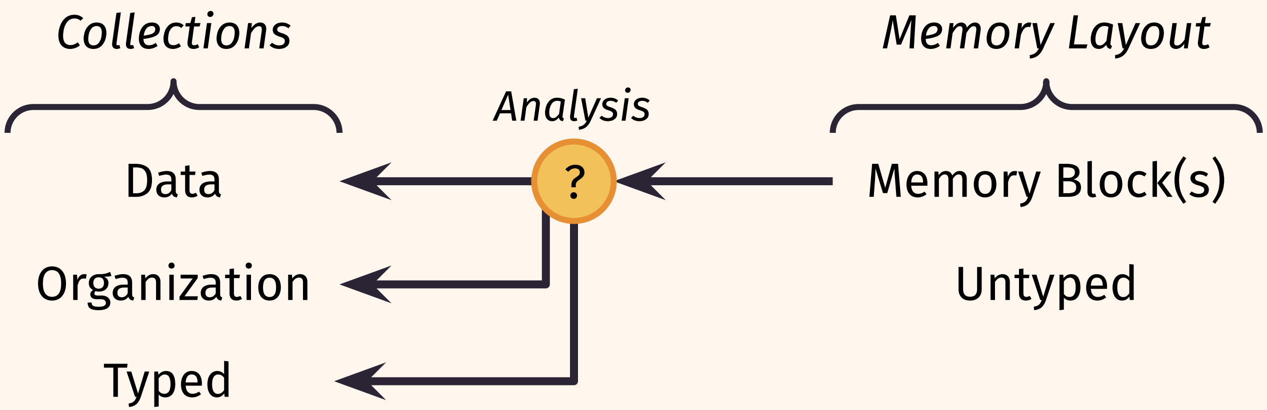
Motivation

Compilers are forced to *glean conservative information*



Motivation

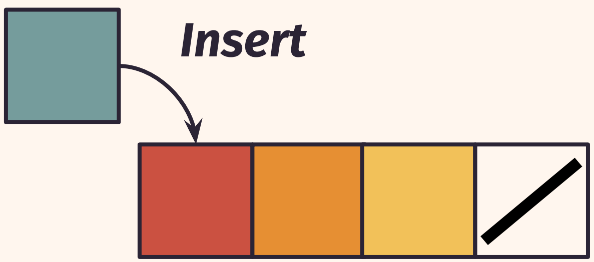
Compilers are forced to *glean conservative information*



Useful information has been destroyed!

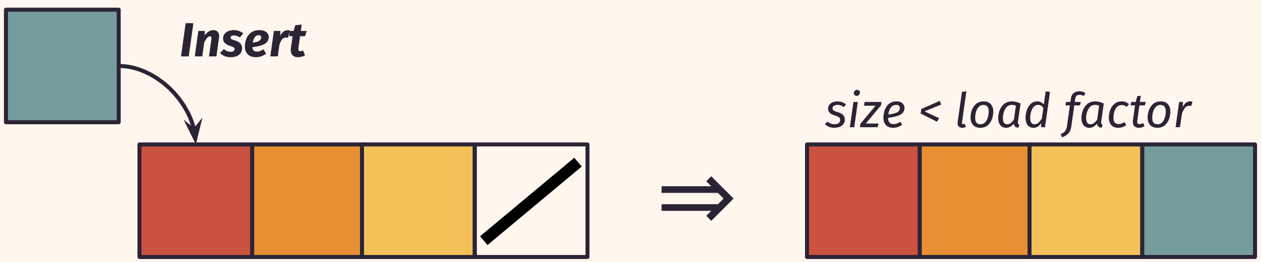
Motivation

Example: Hash table



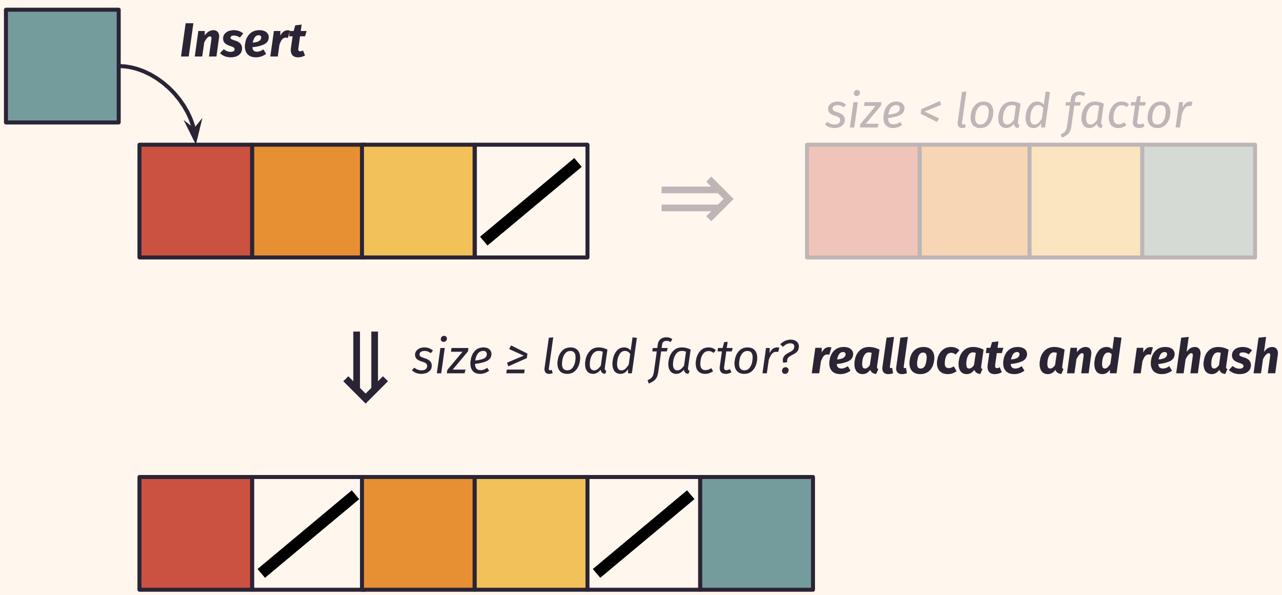
Motivation

Example: Hash table



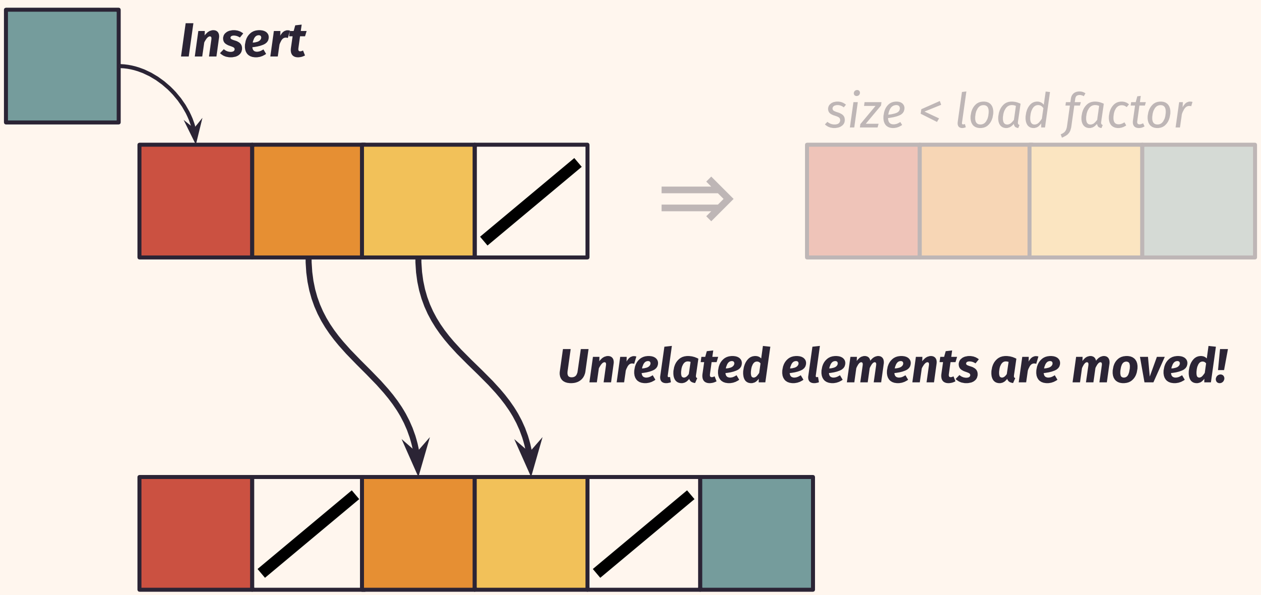
Motivation

Example: Hash table



Motivation

Example: Hash table



Motivation
What are the *consequences?*

```
std::unordered_map<int, int> table = ... ;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

Motivation

What are the *consequences?*

```
std::unordered_map<int, int> table = ... ;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

Motivation

What are the *consequences?*

```
std::unordered_map<int, int> table = ...;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

No production compiler can propagate **10** to the `print` statement

Motivation

What are the *consequences?*

```
std::unordered_map<int, int> table = ... ;

table[0] = 10;
table[1] = 20;    →  { realloc(table, ... );
print(table[0]);       rehash(table);
```

Simple operations → complex memory behavior.

Motivation

What are the *consequences?*

```
std::unordered_map<int, int> table = ... ;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

≠

```
realloc(table, ... );
rehash(table);
```

Complex memory behavior blocks optimizations!

Motivations

Constant Folding Rarely Succeeds with Memory Operations

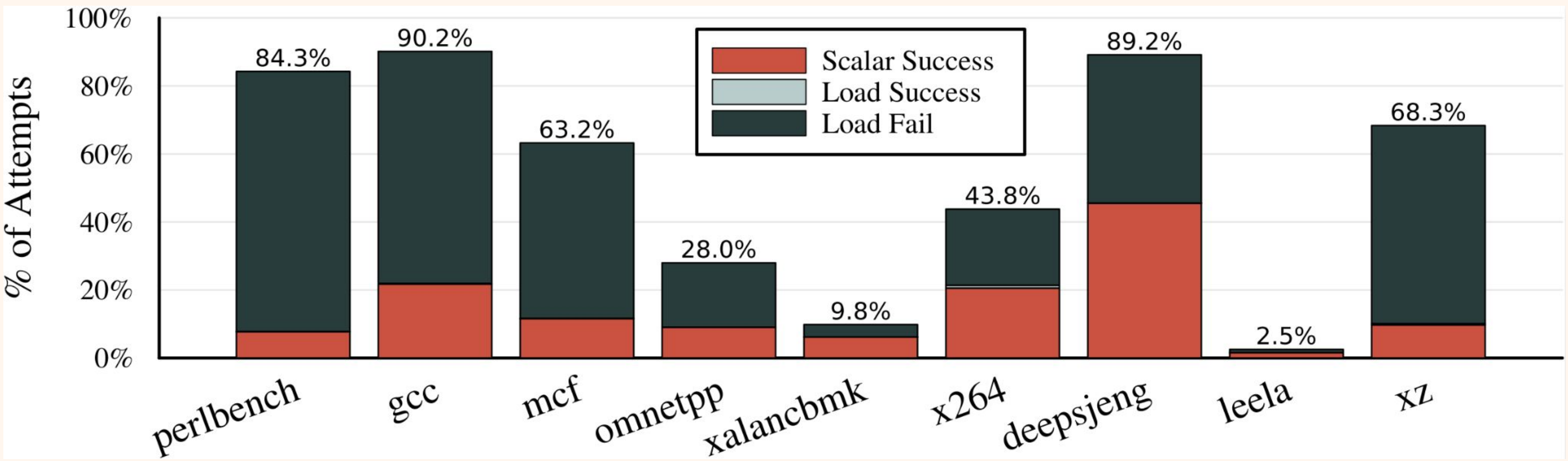
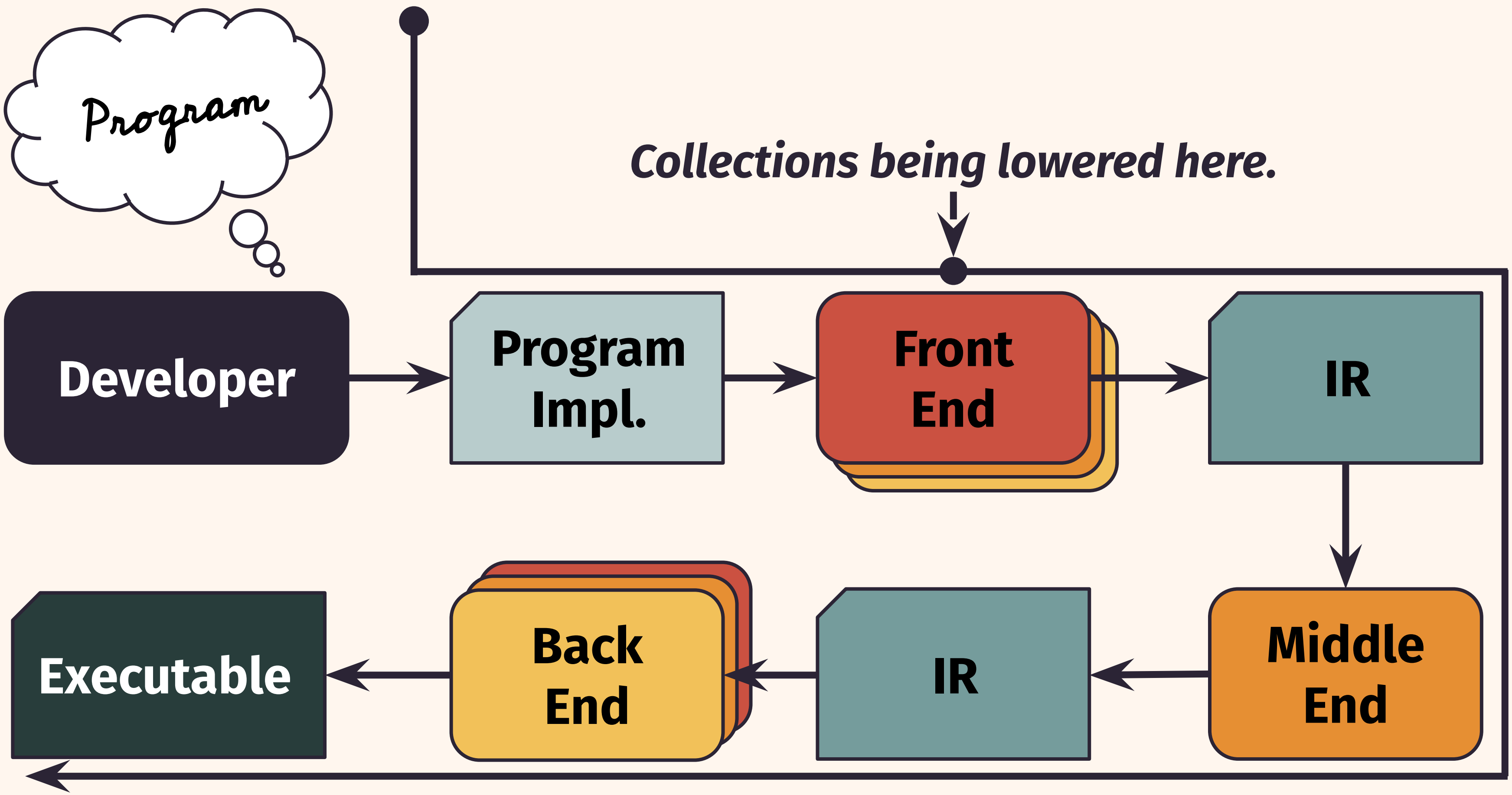


Figure: Breakdown of attempts to perform constant folding.

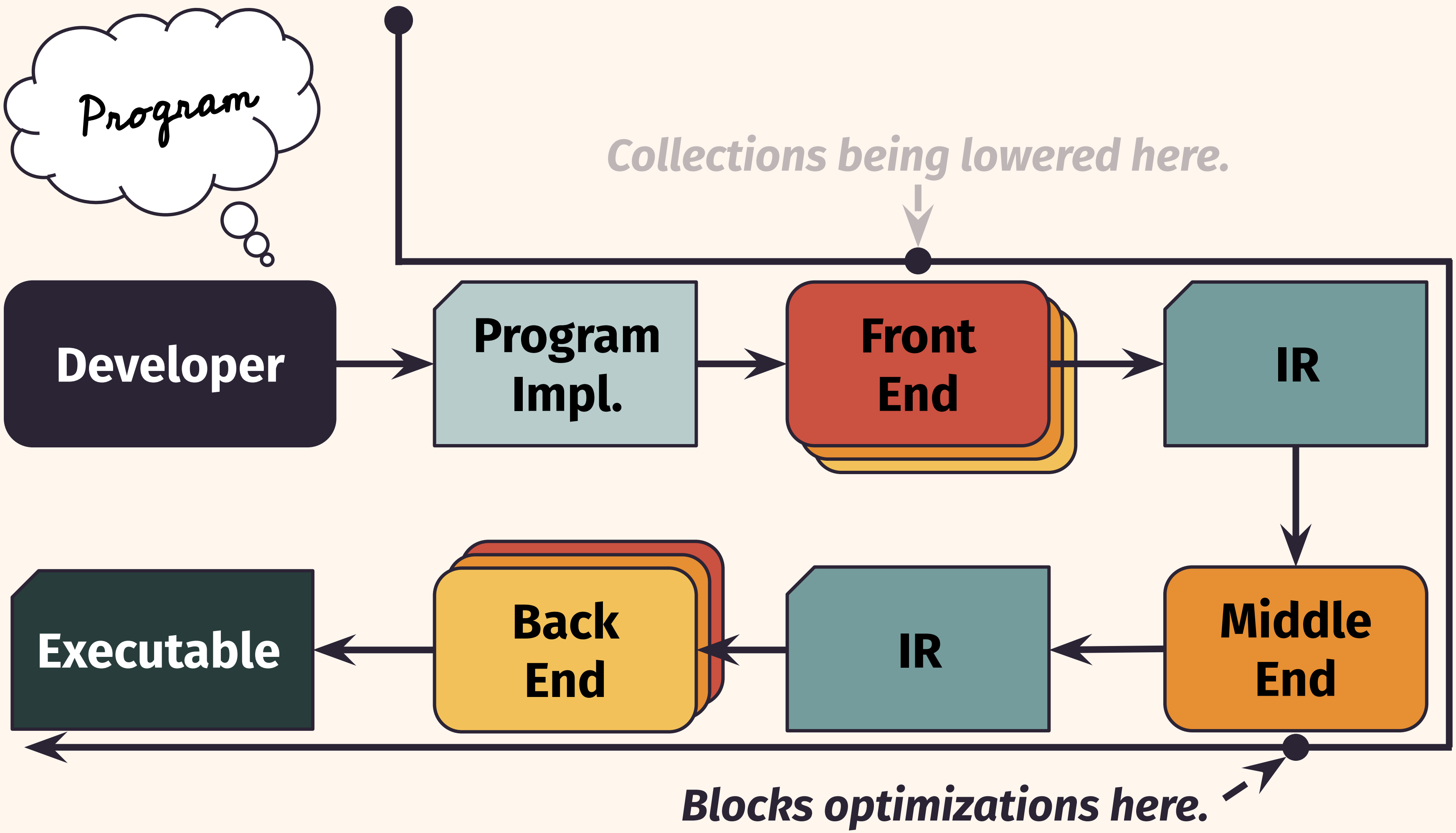
Motivation

Collections are *Prematurely Lowered.*



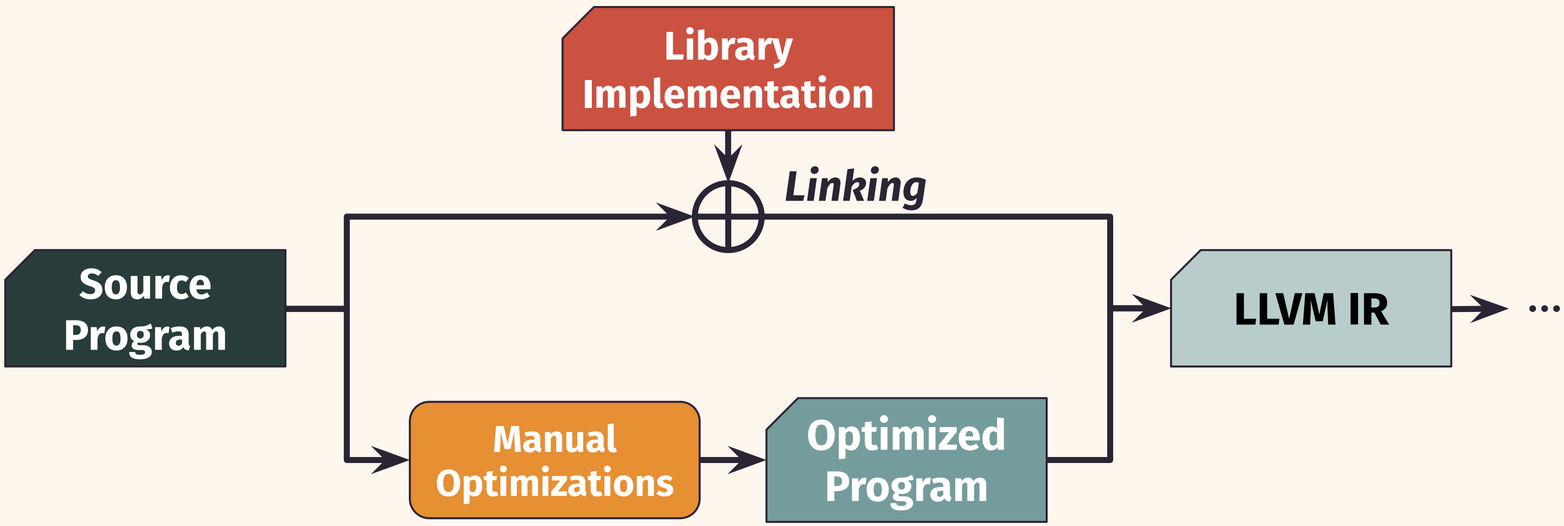
Motivation

Collections are *Prematurely Lowered.*



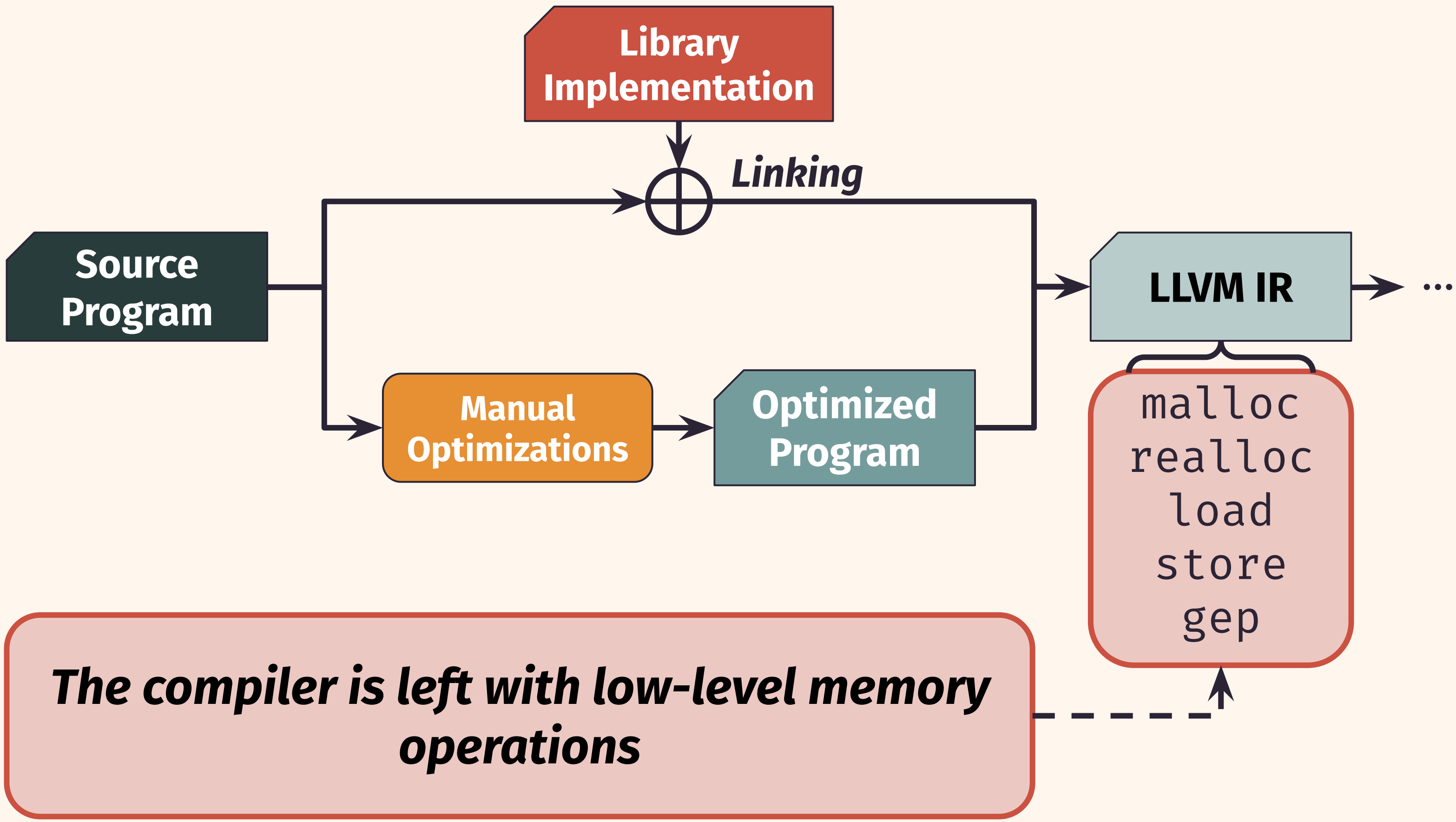
Insights

Stems from *premature lowering* to *fixed implementations* manually or via libraries



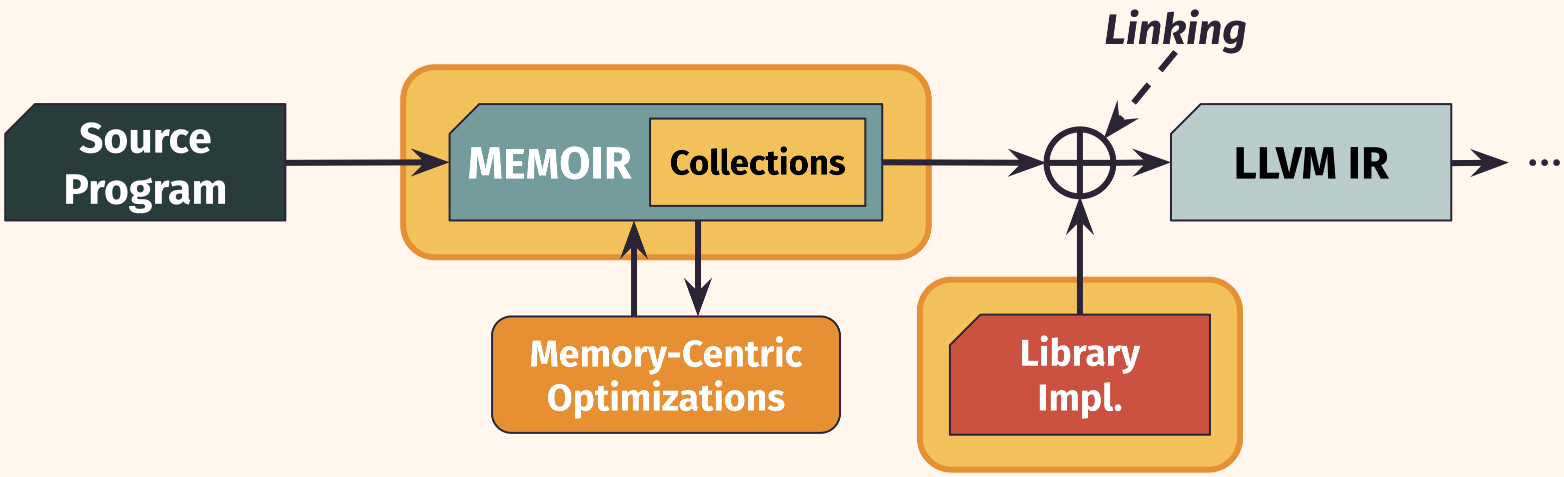
Insights

Stems from *premature lowering* to *fixed implementations* manually or via libraries



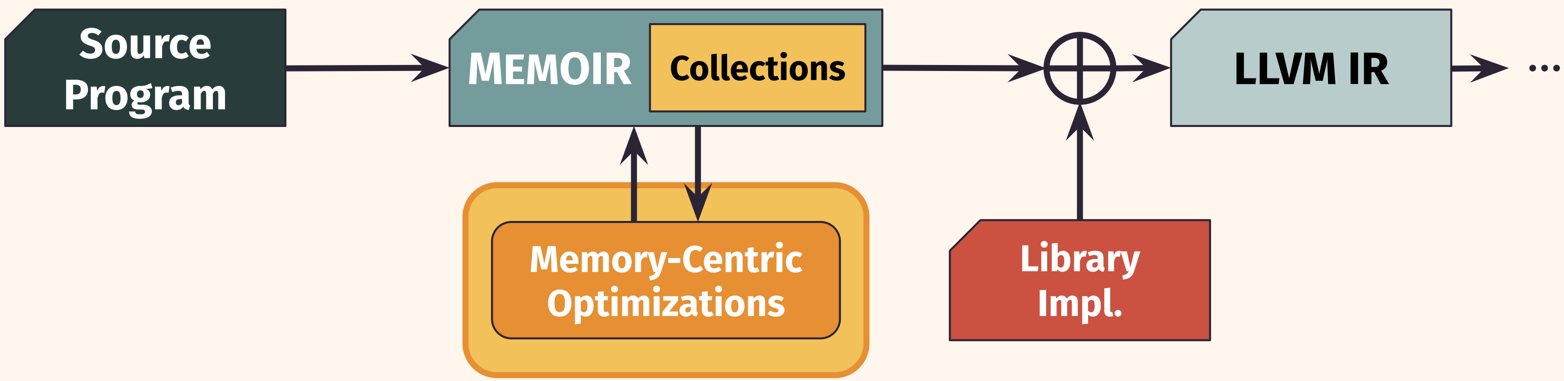
Proposal

Progressively lower **to MEMOIR** ***before*** **library implementation**



Proposal

Implement *Memory Optimizations* within the Compiler for *Easy, Automatic Reuse*



MEMOIR

The first SSA IR for data collections.

Goals

Representing Data Collections in the Compiler

General-purpose

Goals

Representing Data Collections in the Compiler

General-purpose

Amenable to Analysis

Goals

Representing Data Collections in the Compiler

General-purpose

Amenable to Analysis

Amenable to Transformation

Goals

Representing Data Collections in the Compiler

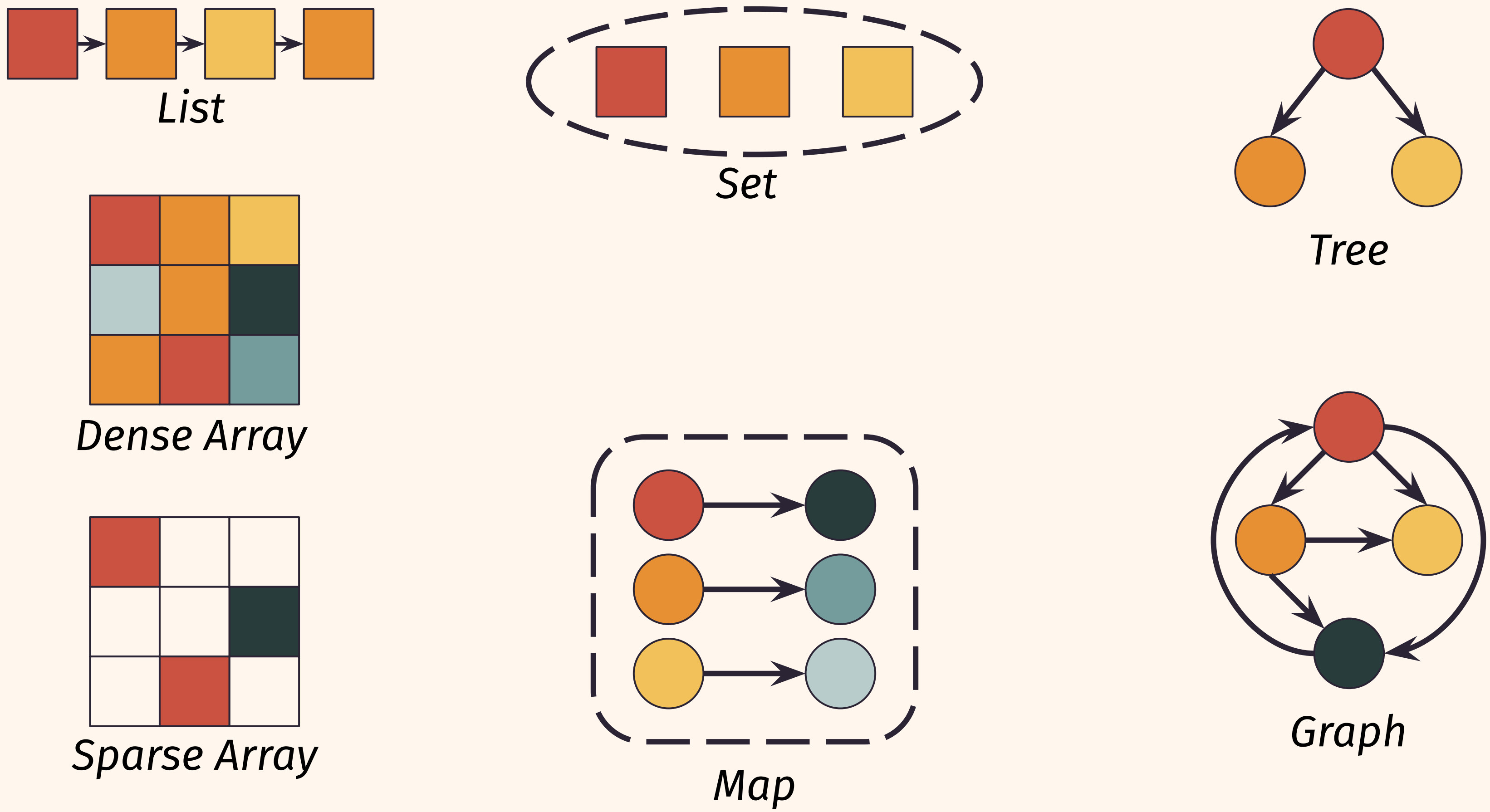
General-purpose

Amenable to Analysis

Amenable to Transformation

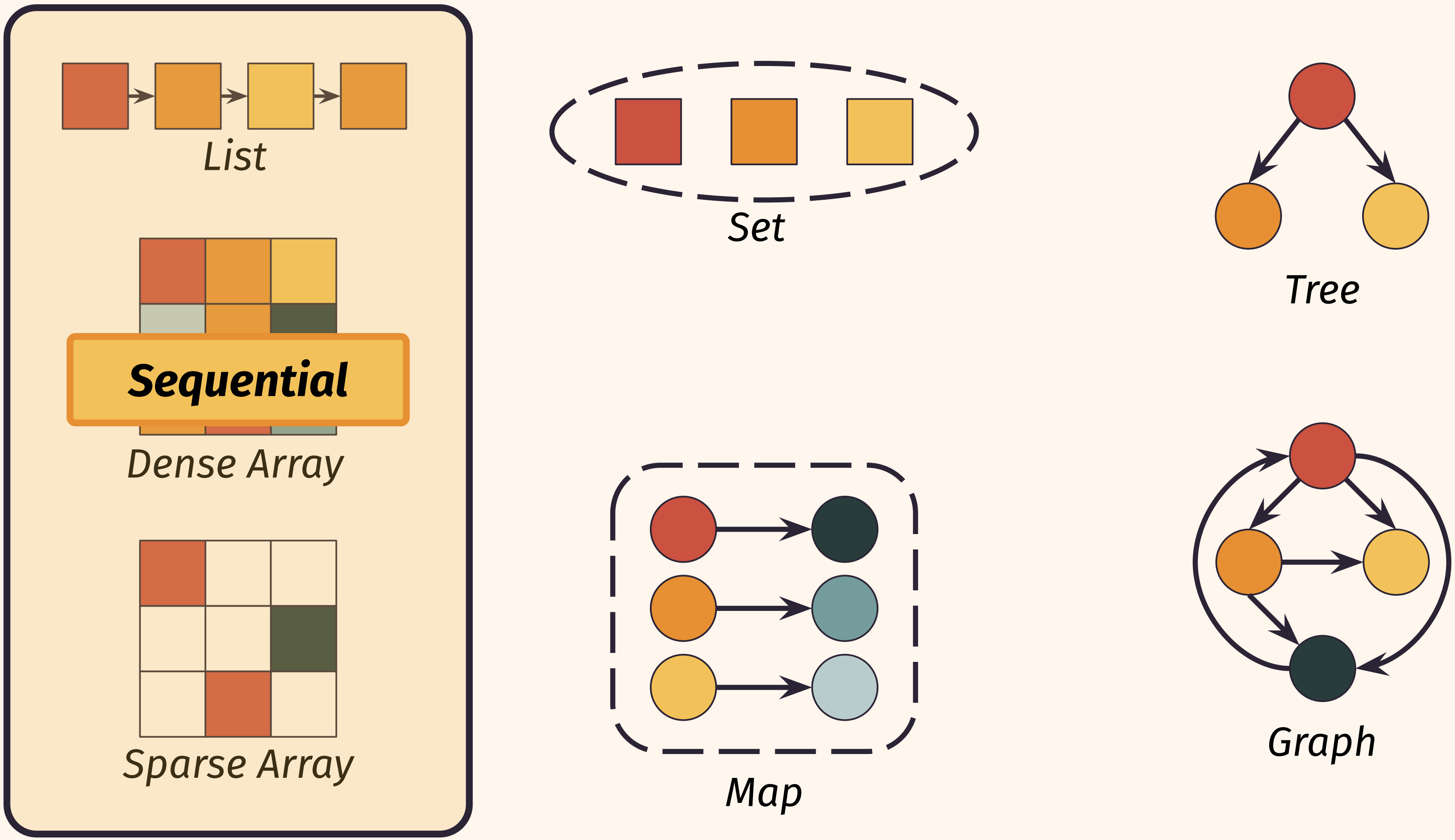
Representation

General-Purpose Data Collections



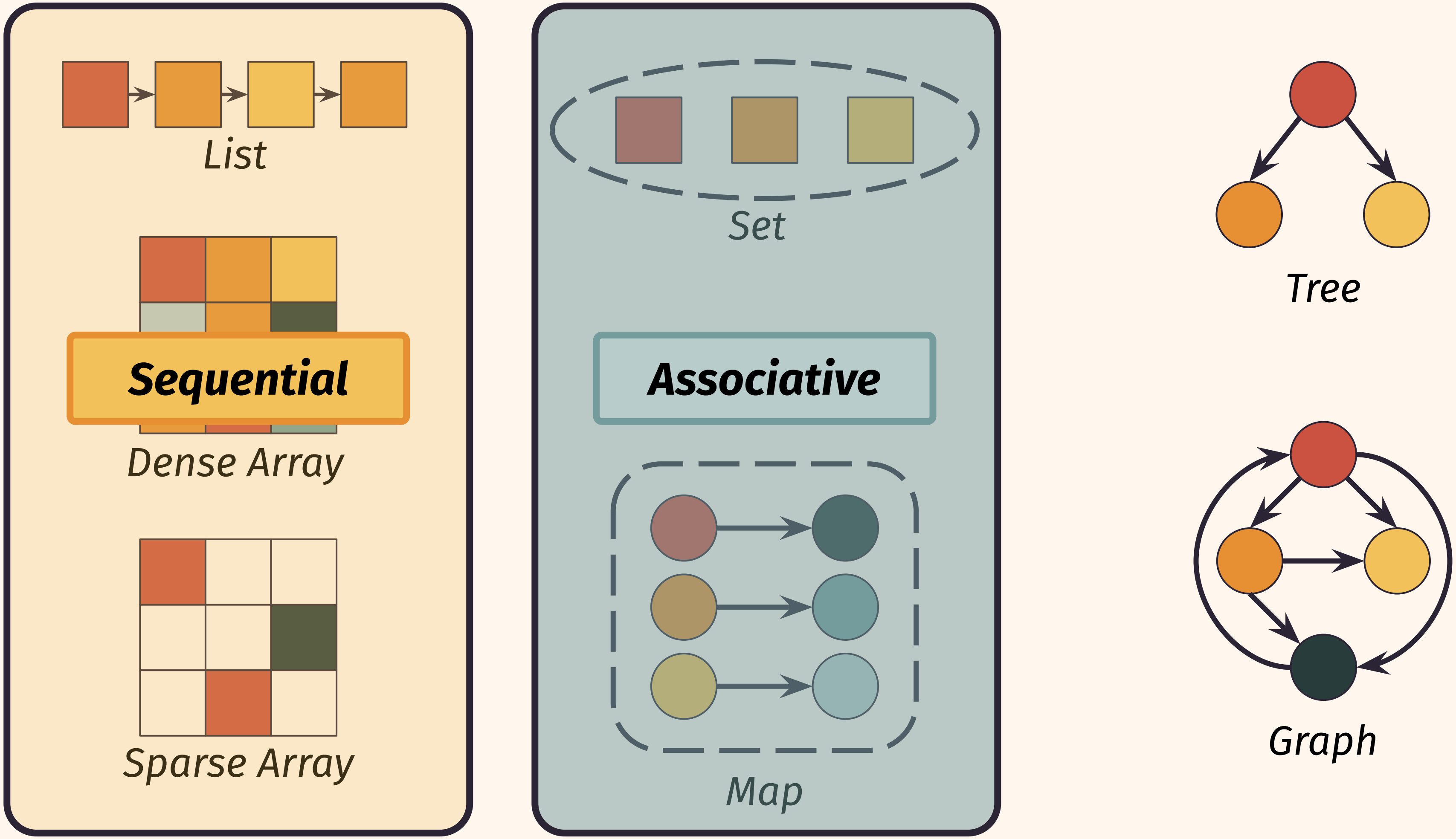
Representation

General-Purpose Data Collections



Representation

General-Purpose Data Collections



Representation

Most accesses to heap memory is for structured data

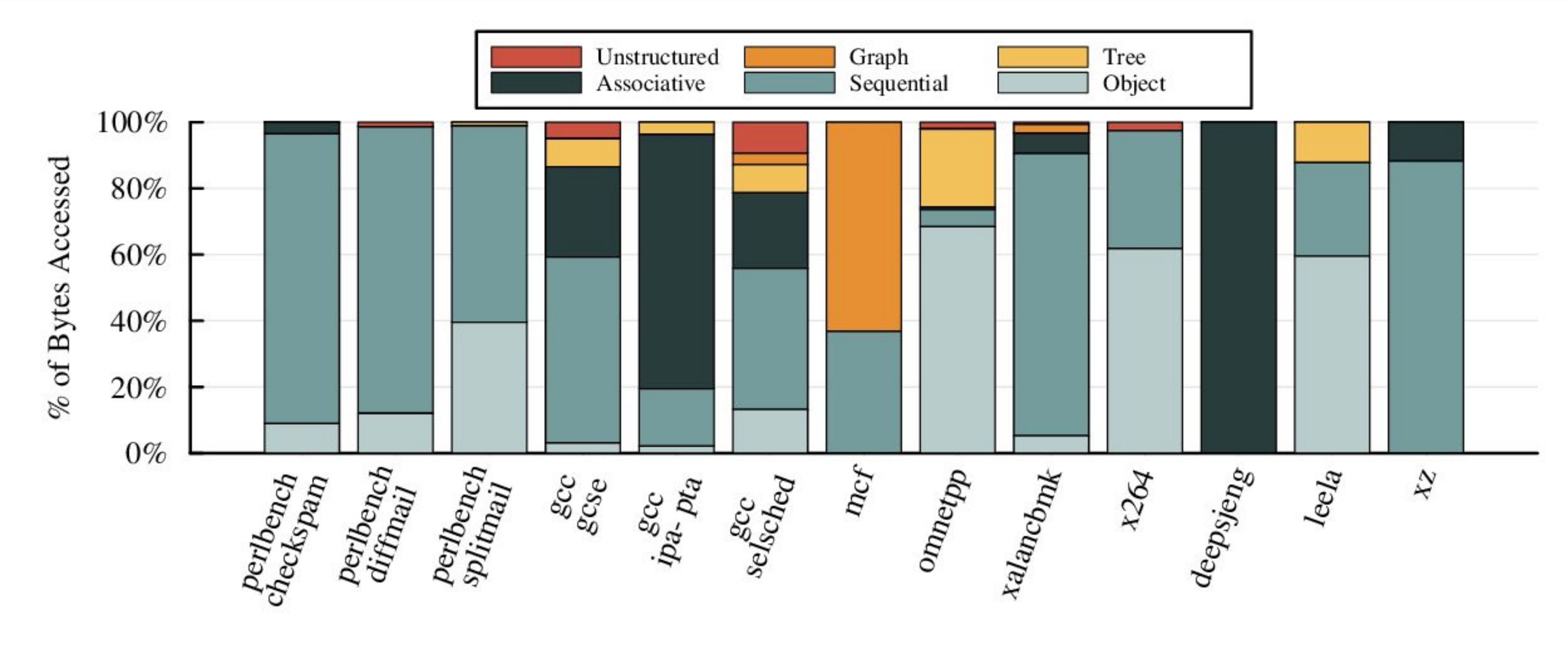


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

Representation

Most accesses to heap memory is for structured data

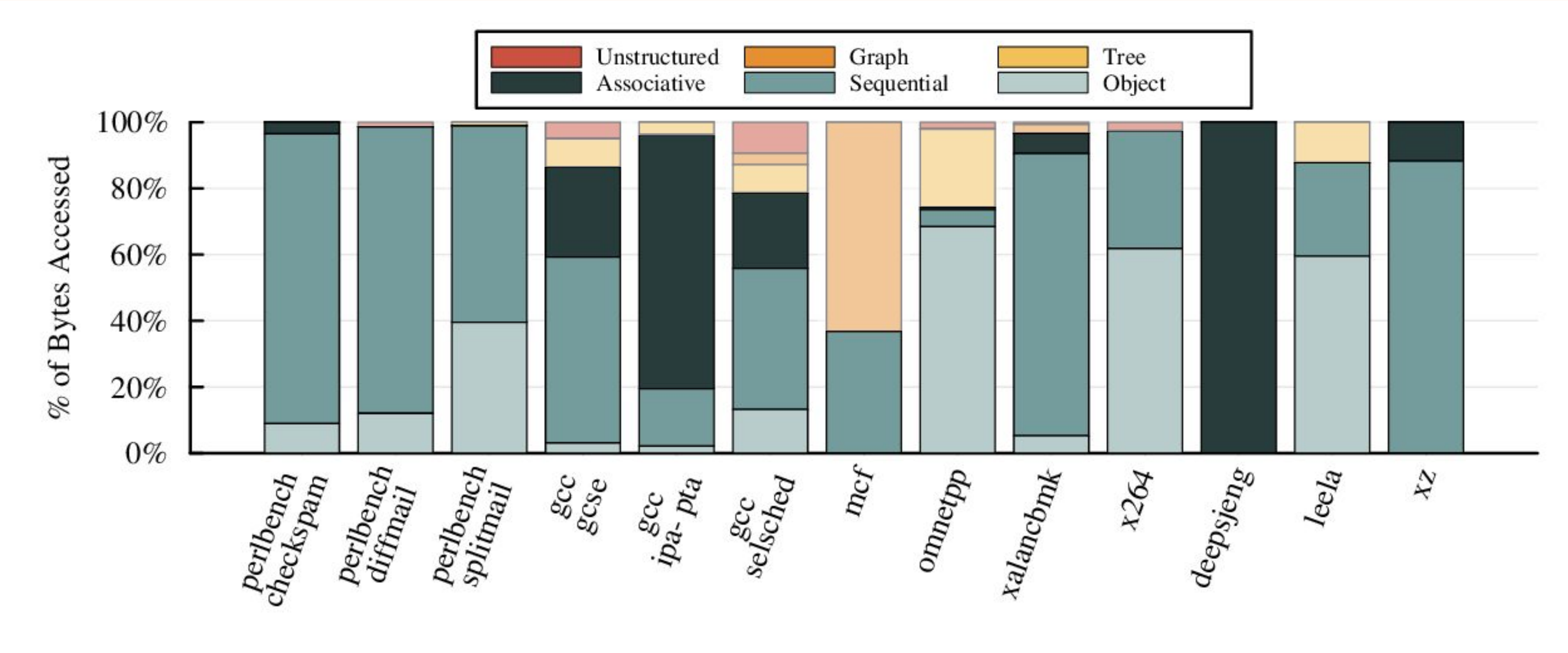
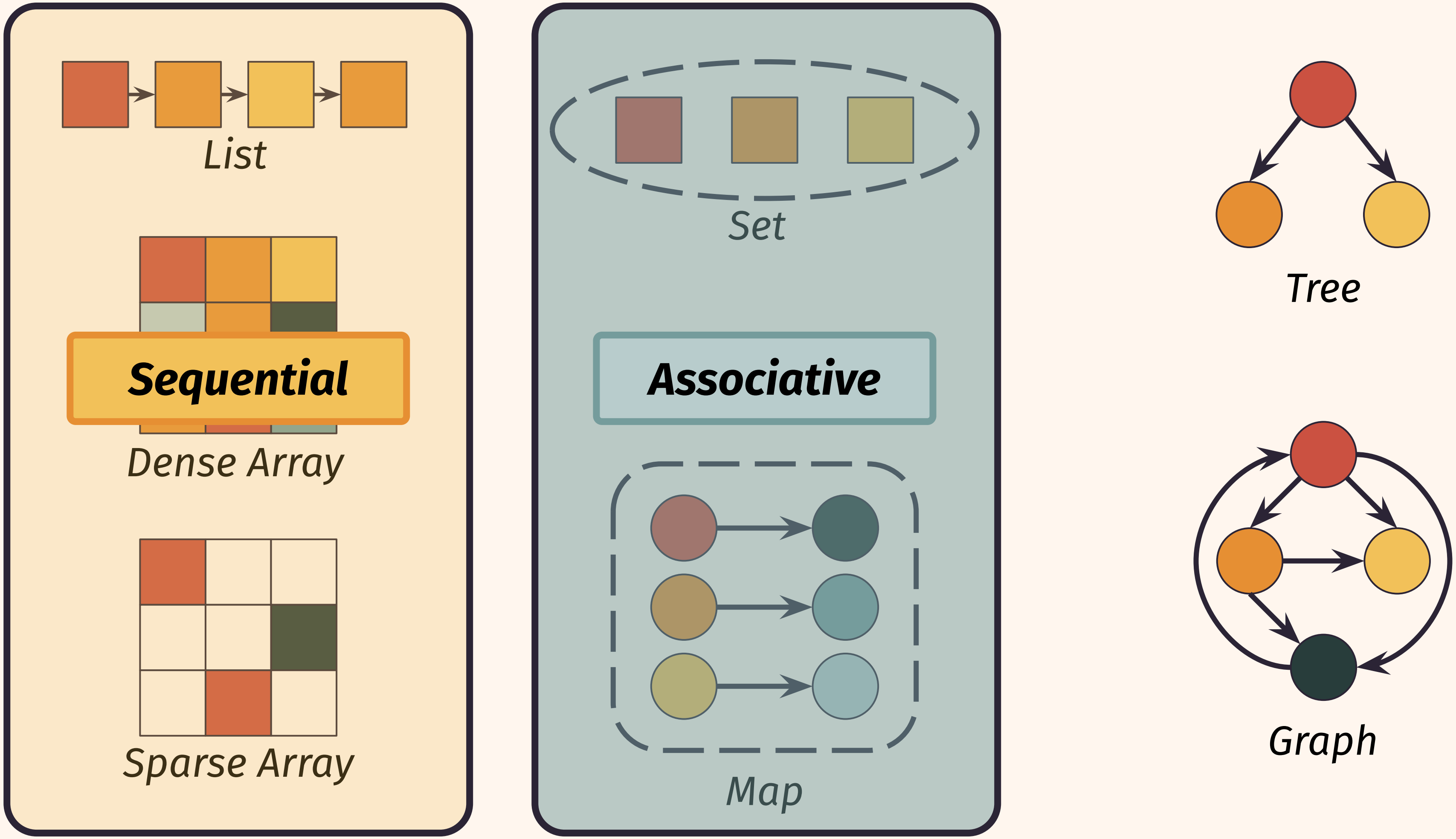


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

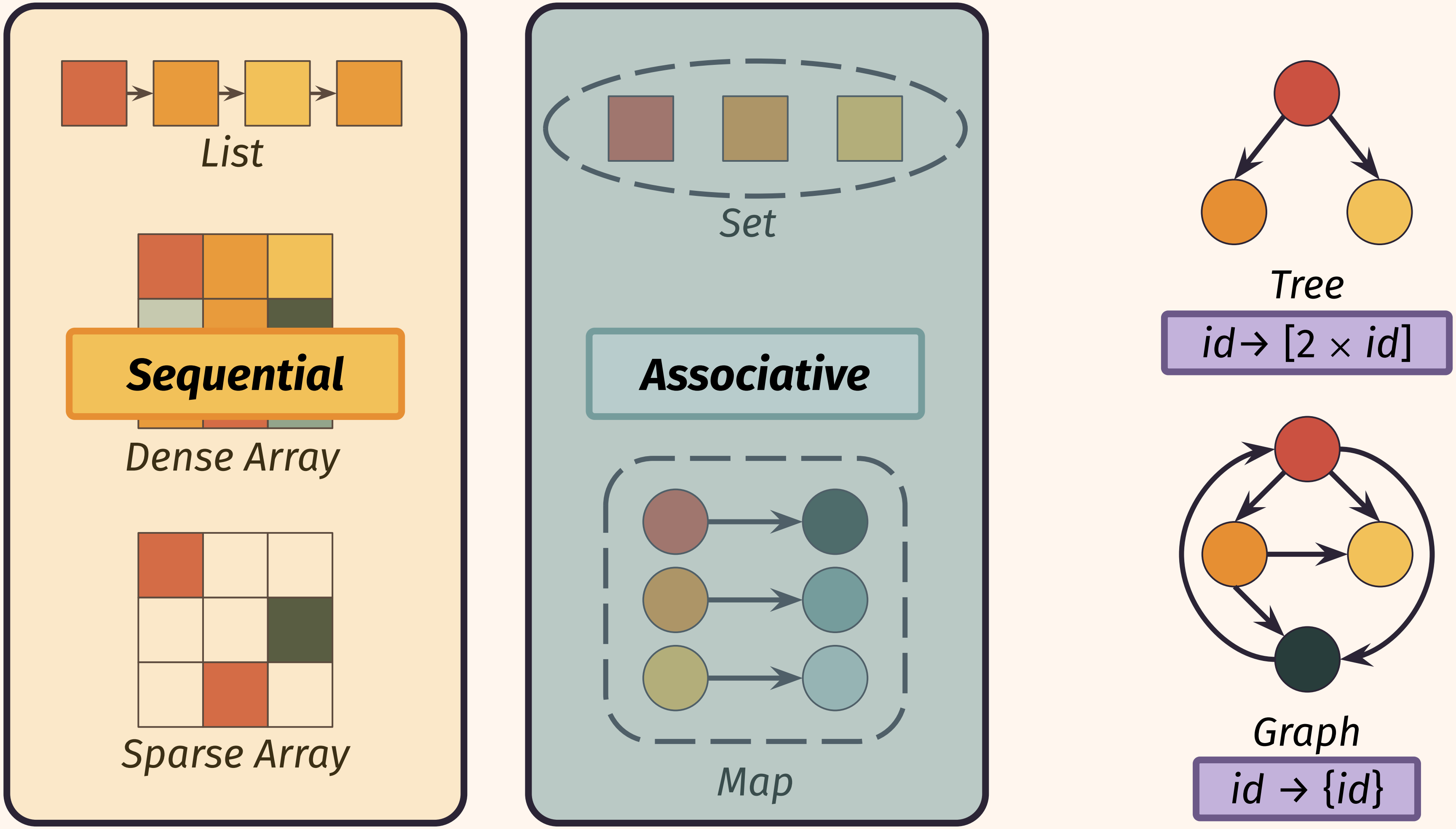
Representation

General-Purpose Data Collections



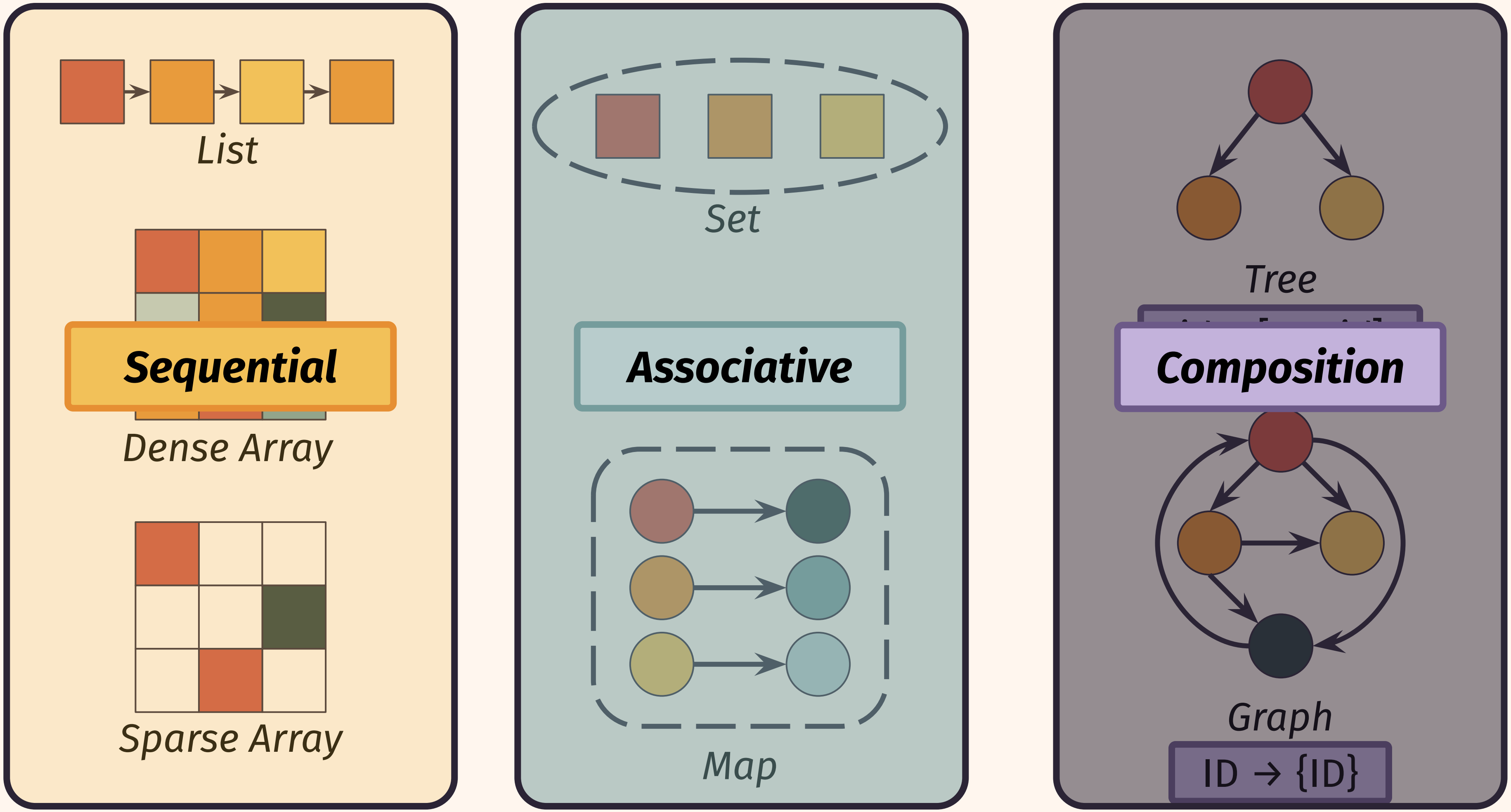
Representation

General-Purpose Data Collections



Representation

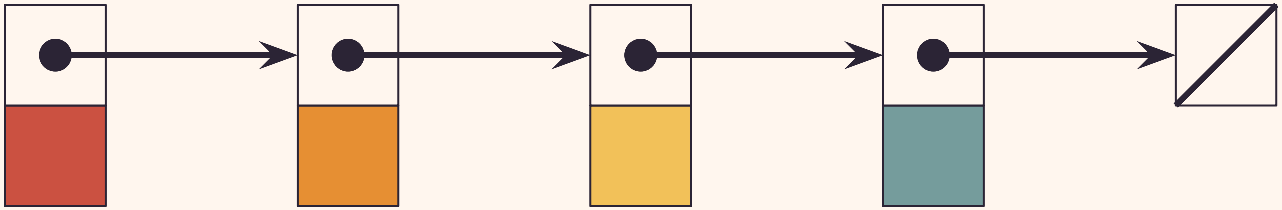
General-Purpose Data Collections



Representation

Decoupling Data from its Logical Organization

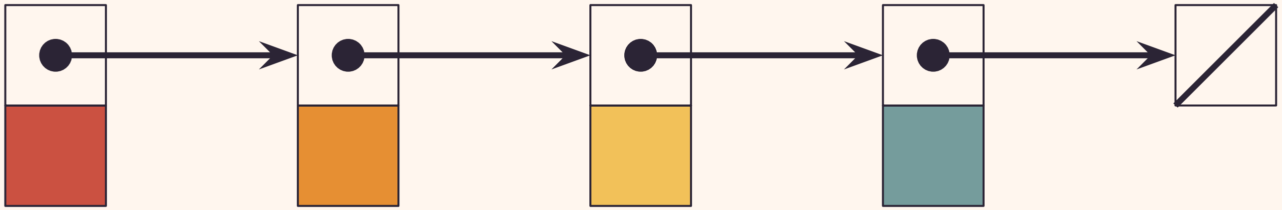
Example: Linked List



Representation

Decoupling Data from its Logical Organization

Example: Linked List

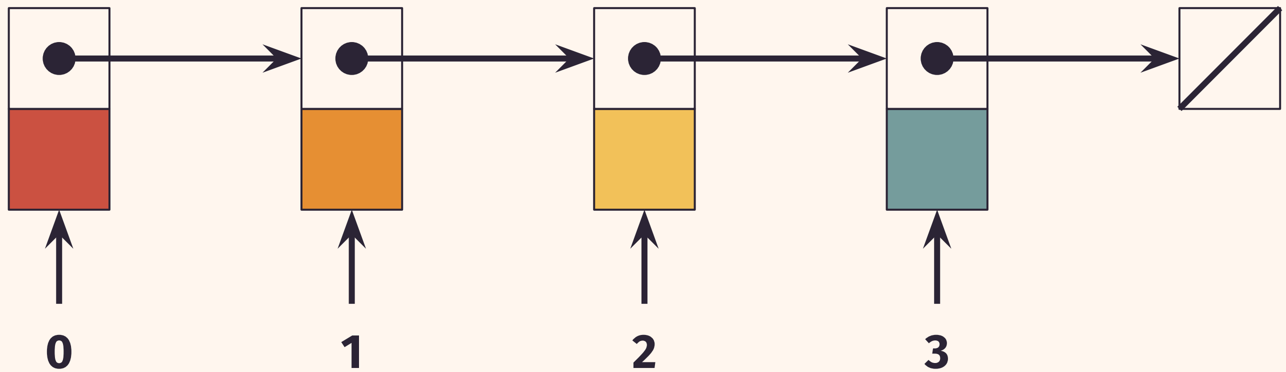


Index space: **0** **1** **2** **3**

Representation

Decoupling Data from its Logical Organization

Example: Linked List



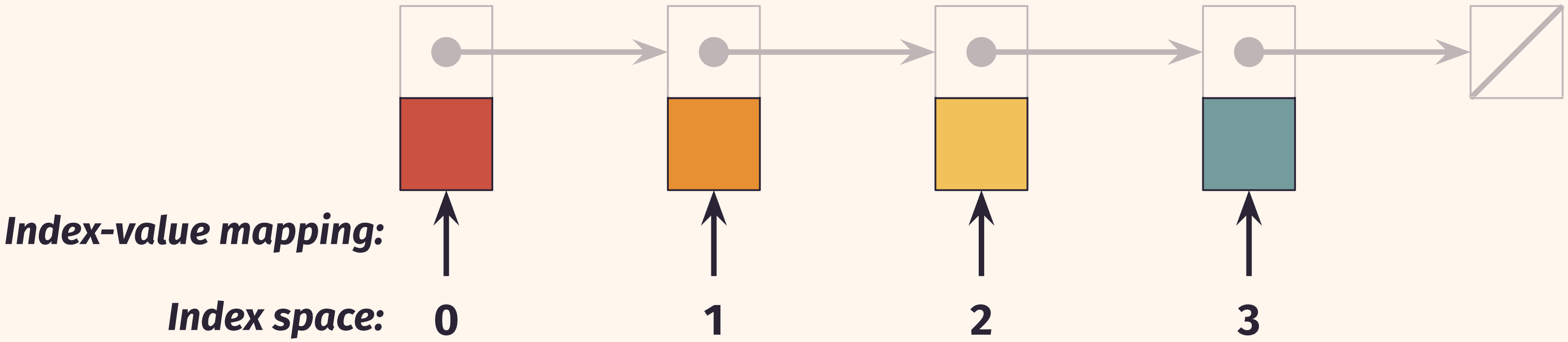
Index-value mapping:

Index space:

Representation

Decoupling Data from its Logical Organization

Example: Linked List



Abstract away the memory used to ***logically organize*** the collection.

Representation

Abstraction of Logical Organization

Associative

Uniqueness in index space

$\underbrace{\hphantom{xxxxxxxxxx}}$

Map *keys* to values

Sequential

Contiguous in index space

$\underbrace{\hphantom{xxxxxxxxxx}}$

Map *indices* to values

Abstraction of Logical Organization

Associative
Uniqueness in index space

$\supset$

Sequential
Contiguous in index space

Additional properties of index space

Representation

Takeaway

Multiple layouts → single abstraction

+

Structure and properties of data organization

=

General-purpose

Overview

Representing Data Collections in the Compiler

General-purpose ✓

Amenable to Analysis

Amenable to Transformation

Overview

Representing Data Collections in the Compiler

General-purpose

Amenable to Analysis

Amenable to Transformation

Analysis

Enabling Analysis with Intermediate Representations

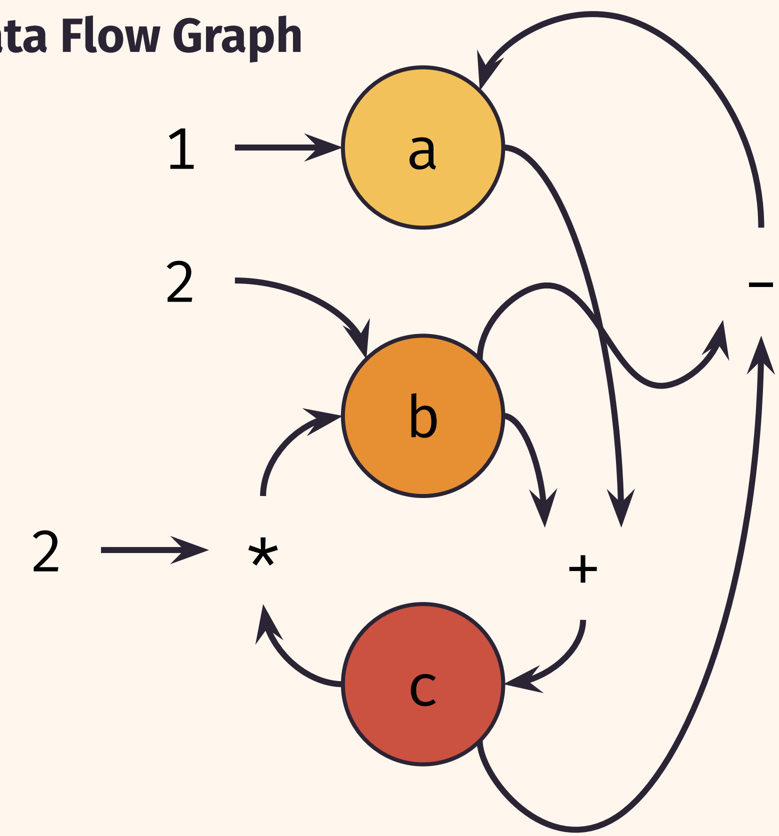
Impact of Program Representation on Analysis

Program

```
a = 1
b = 2
c = a + b
b = c * 2
a = b - c
```

⇒

Data Flow Graph



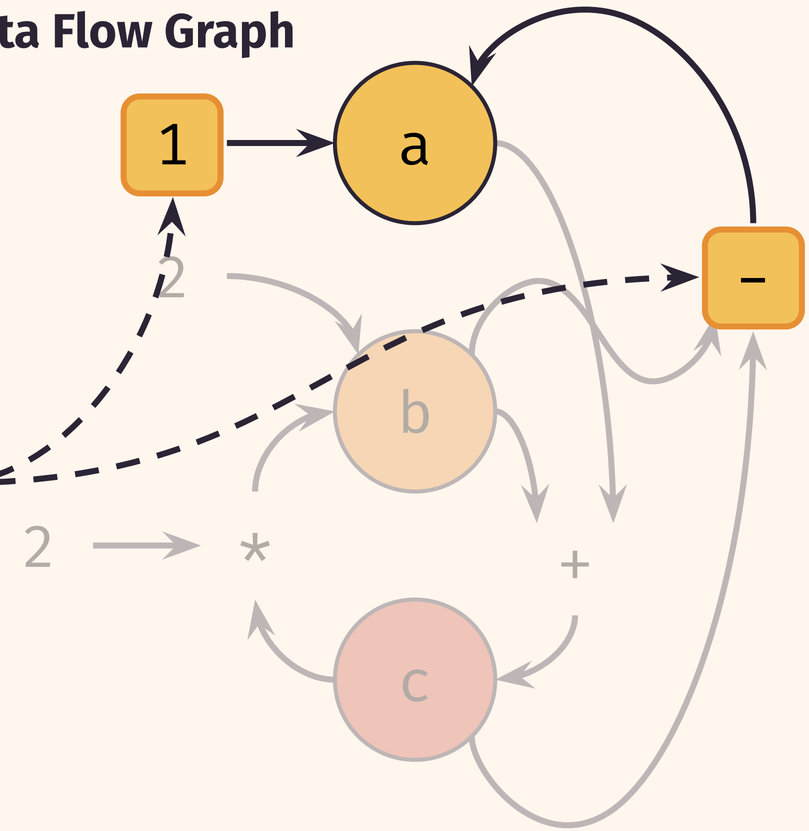
Impact of Program Representation on Analysis

Program

```
a = 1
b = 2
c = a + b
b = c * 2
a
```

⇒

Data Flow Graph



a can be defined by *multiple operations*

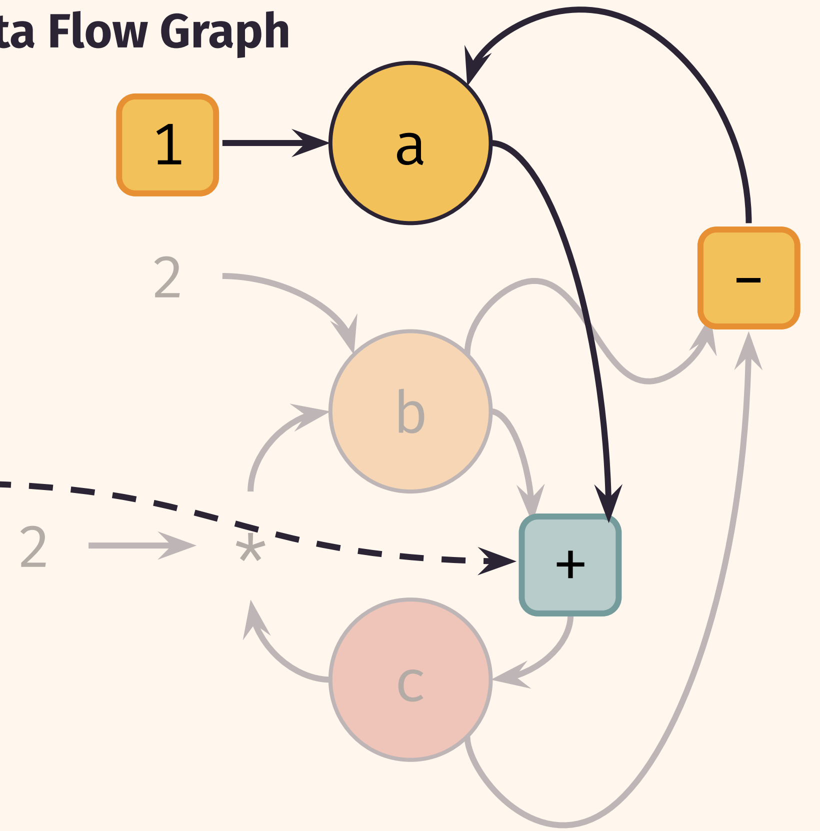
Impact of Program Representation on Analysis

Program

```
a = 1
b = 2
c = a + b
b = c * 2
```

⇒

Data Flow Graph



Which assignment to **a** is used by the **+** operation?

Analysis

Enabling Analysis with IR Design

Intermediate representations (IR) *simplify analysis and transformation*.

Enabling Analysis with IR Design

Intermediate representations (IR) ***simplify analysis and transformation***.

Example: **Static Single Assignment (SSA)**

Analysis

Static Single Assignment (SSA)

Each variable in program has a ***single definition***

```
a = 1
b = 2
c = a + b
b = c * 2
a = b - c
```

$\Rightarrow$

```
a = 1
b = 2
c = a + b
b' = c * 2
a' = b' - c
```

Analysis

Static Single Assignment (SSA)

Each variable in program has a ***single definition***

```
a = 1            a  = 1
b = 2            b  = 2
c = a + b   ⇒    c  = a + b
b = c * 2        b' = c * 2
a = b - c        a' = b' - c
```

Each variable use has ***referential transparency:***
The variable can be replaced with its definition.

Analysis

Static Single Assignment (SSA)

Each variable in program has a ***single definition***

```
a = 1
b = 2
c = a + b
b = c * 2
a = b - c
```

⇒

```
a = 1
b = 2
c = a + b
b' = c * 2
a' = b' - c
```

We can easily resolve the ambiguous use of **a** by the **+** operation now!

Example: Constant Propagation

Each variable in program has a ***single definition***

```
a = 1                    a  = 1
b = 2                    b  = 2
c = a + b       ⇒        c  = a + b
b = c * 2                b' = c * 2
a = b - c                a' = b' - c
```

With referential transparency, **constant propagation becomes *trivial***

Analysis

Example: Constant Propagation

Each variable in program has a ***single definition***

```
a = 1
b = 2
c = a + b
b = c * 2
a = b - c
```

⇒

```
a  = 1
b  = 2
c  = a + b
b' = c * 2
a' = b' - c
```

With referential transparency, **constant propagation becomes *trivial***

Example: Constant Propagation

Each variable in program has a ***single definition***

```
a = 1                a  = 1
b = 2                b  = 2
c = a + b     ⇒      c  = 1 + 2
b = c * 2            b' = c * 2
a = b - c            a' = b' - c
```

With referential transparency, **constant propagation becomes *trivial***

Data Collections

An SSA collection variable is the *only reference* to that collection.

Data Collections

An SSA collection variable is the *only reference* to that collection.

SSA collections are *immutable* for their static lifetime.
A collection variable *is* a collection

MEMOIR

Element-Level Operations

Conventional

`allocate(# of bytes)`

Collection-Aware

`allocate(# of elements)`

Reason about ***elements*** instead of ***bytes***.

MEMOIR

Element-Level Operations

Conventional

`allocate(# of bytes)` → `allocate(# of elements)`

`reallocate(pointer, # of bytes)` → `insert(collection, index, ...)`

→ `remove(collection, index)`

Collection-Aware

Whether a collection is growing *or* shrinking is ***explicit***.

MEMOIR

Element-Level Operations

Conventional		***Collection-Aware***
`allocate(# of bytes)`	→	`allocate(# of elements)`
`reallocate(pointer, # of bytes)`	→	`insert(collection, index)`
	→	`remove(collection, index)`
`access(pointer, offset in bytes)`	→	`access(collection, index)`

Access ***explicitly references collection and element***.

Analysis

Where *scalar* analysis and transformation *fails*.

LLVM IR, etc.

Constant *Scalar* Propagation

```
... table = ... ;


table[0] = 10;
table[1] = 20;
print(table[0]);
```

Analysis

Element-Level **analysis and transformation can *prevail*.**

LLVM IR, etc.

Constant *Scalar* Propagation

```
... table = ... ;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

MEMOIR

Constant *Element* Propagation

```
... table = ... ;

table[0] = 10;
table[1] = 20;
print(10);
```

Analysis

Construct a MEMOIR Program

LLVM IR, etc. **MEMOIR**

Constant *Scalar* Propagation → **Constant *Element* Propagation**

```
... table = ... ;  →  Assoc<int,int> table = ... ;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

Construct a MEMOIR Program

LLVM IR, etc.

MEMOIR

Constant *Scalar* Propagation

Constant *Element* Propagation

```
...  table  =  ... ;   →   Assoc&lt;int,int&gt; table =  ... ;

table[0] = 10;         →   table1 = INSERT(table,  0, 10);
table[1] = 20;
print(table[0]);
```

Analysis

Construct a MEMOIR Program

LLVM IR, etc. **MEMOIR**

Constant *Scalar* Propagation **Constant *Element* Propagation**

```
... table = ... ;  →  Assoc<int,int> table = ... ;

table[0] = 10;     →  table1 = INSERT(table, 0, 10);
table[1] = 20;
print(table[0]);
```

Analysis

Construct a MEMOIR Program

LLVM IR, etc. → **MEMOIR**

Constant *Scalar* Propagation → **Constant *Element* Propagation**

```
... table = ... ;          →  Assoc<int,int> table = ... ;

table[0] = 10;             →  table1 = INSERT(table,  0, 10);
table[1] = 20;             →  table2 = INSERT(table1, 1, 20);
print(table[0]);
```

Analysis

Construct a MEMOIR Program

LLVM IR, etc.

Constant *Scalar* Propagation

MEMOIR

Constant *Element* Propagation

```
... table = ... ;            →  Assoc<int,int> table = ... ;

table[0] = 10;               →  table1 = INSERT(table,  0, 10);
table[1] = 20;               →  table2 = INSERT(table1, 1, 20);
print(table[0]);
```

Analysis

Construct a MEMOIR Program

LLVM IR, etc. | **MEMOIR**

Constant *Scalar* Propagation | **Constant *Element* Propagation**

```
... table = ... ;          →  Assoc&lt;int,int> table = ... ;

table[0] = 10;             →  table1 = INSERT(table,  0, 10);
table[1] = 20;             →  table2 = INSERT(table1, 1, 20);
print(table[0]);           →  r = READ(table2, 0)
                           →  print(r);
```

Analysis

Construct a MEMOIR Program

LLVM IR, etc.

Constant *Scalar* Propagation

```
... table = ... ;
table[0] = 10;
table[1] = 20;
print(table[0]);
```

MEMOIR

Constant *Element* Propagation

```
Assoc<int,int> table = ... ;
table1 = INSERT(table,  0, 10);
table2 = INSERT(table1, 1, 20);
r = READ(table2, 0)
print(r);
```

Analysis

Construct a MEMOIR Program

Benchmark	*Number of Collections* Source	SSA
`mcf`	5	13
`deepsjeng`	2	14
`LLVM opt`	8	37

SSA Construction introduces new collections.

Analysis

Construct a MEMOIR Program

LLVM IR | MEMOIR

Benchmark	Number of Collections		
	Source	SSA	Binary
`mcf`	5	13	5
`deepsjeng`	2	14	2
`LLVM opt`	8	37	8

SSA Destruction eliminates them, with *no new copies!*

Analysis

Performing a *sparse data flow analysis* on collections.

LLVM IR, etc.

Constant *Scalar* Propagation

MEMOIR

Constant *Element* Propagation

```
            { } ◁ Assoc<int,int> table = ... ;

      { 0→10 } ◁ table1 = INSERT(table,  0, 10);
{ 0→10, 1→20 } ◁ table2 = INSERT(table1, 1, 20);
                 r = READ(table2, 0)
                 print(r);
```

Analysis

Propagate *element-level* constants to optimize the program.

LLVM IR, etc. — **Constant *Scalar* Propagation**

MEMOIR — **Constant *Element* Propagation**

```
{ } ◁ Assoc<int,int> table = ... ;

{ 0→10 } ◁ table1 = INSERT(table,  0, 10);
{ 0→10, 1→20 } ◁ table2 = INSERT(table1, 1, 20);
                  r = READ(table2, 0)
                  print(r);
```

Analysis

Propagate *element-level* constants to optimize the program.

LLVM IR, etc.

MEMOIR

Constant *Scalar* Propagation

Constant *Element* Propagation

```
                 { } ◁ Assoc<int,int> table = ... ;

           { 0→10 } ◁ table1 = INSERT(table,  0, 10);
      { 0→10, 1→20 } ◁ table2 = INSERT(table1, 1, 20);
                       r = 10;
                       print(r);
```

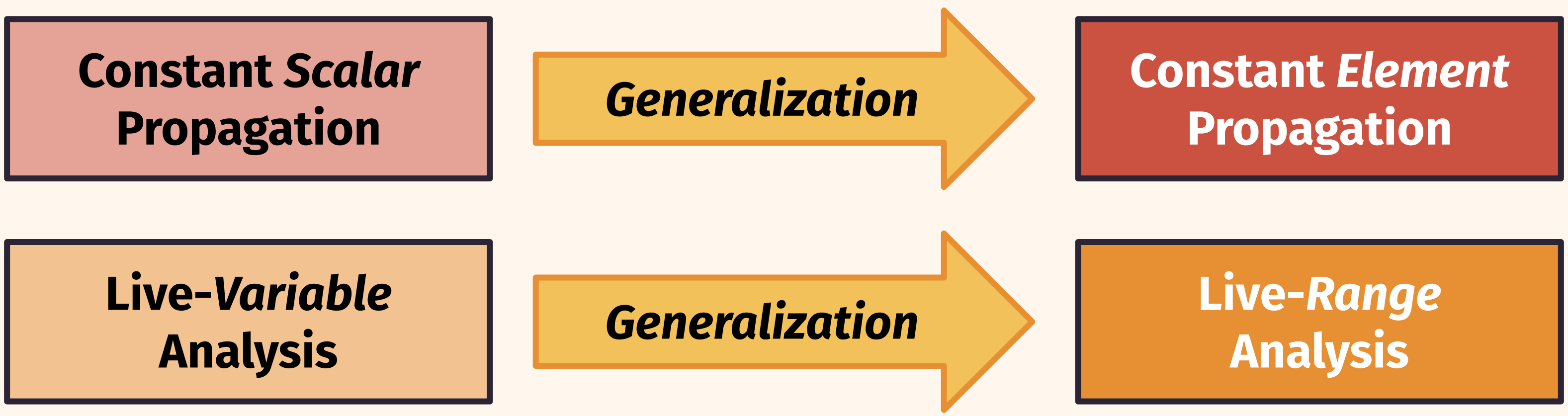
Analysis

Generalizing scalar optimizations to operate on *elements*

Analysis

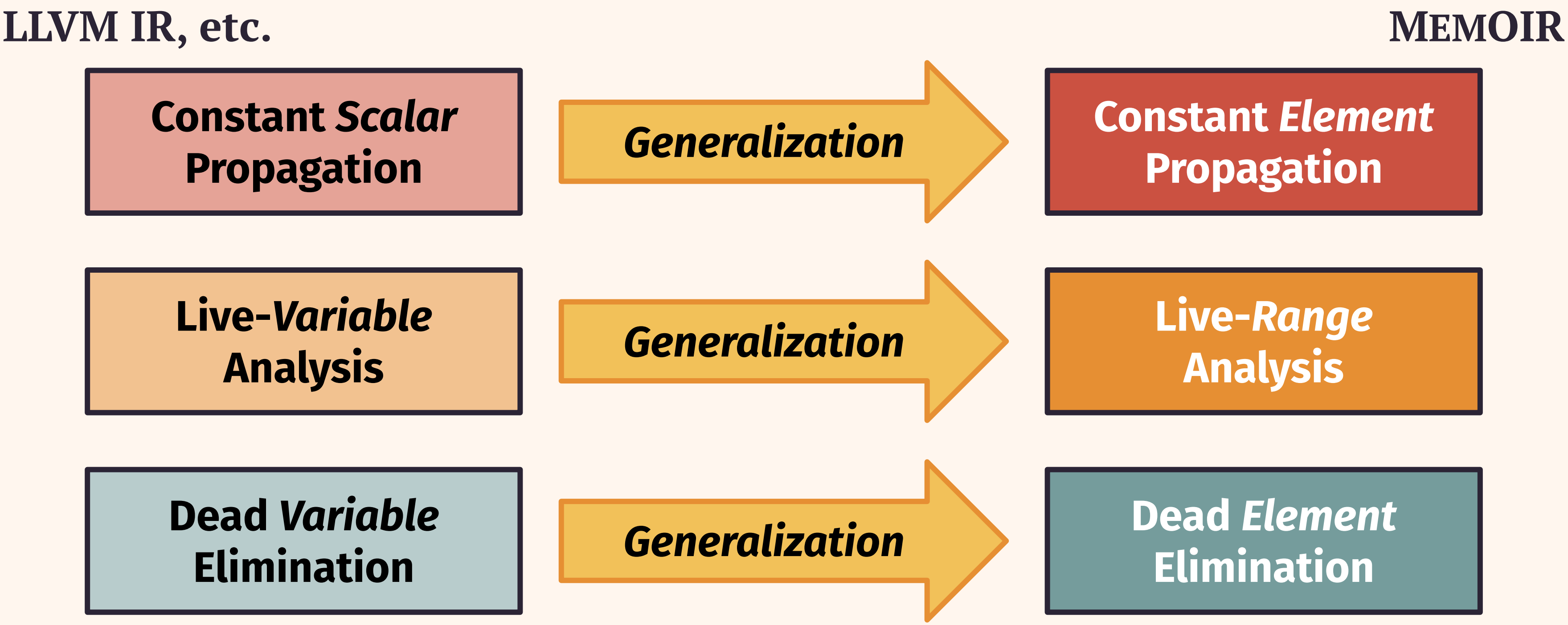
Generalizing scalar optimizations **to operate on** ***elements***

LLVM IR, etc. | MEMOIR



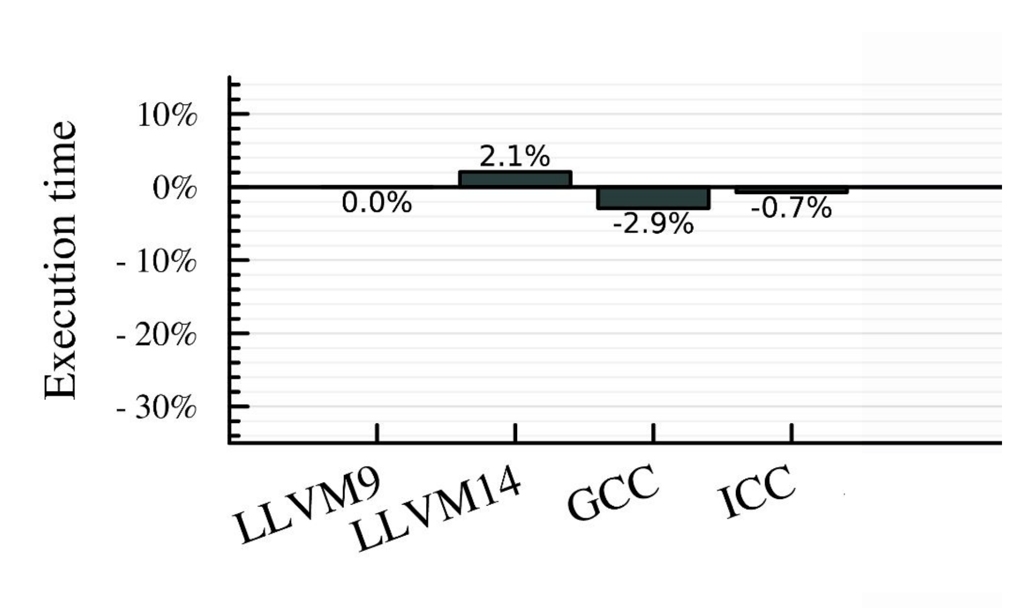
Analysis

Generalizing scalar optimizations to operate on *elements*



Evaluation

Production compilers provide negligible performance improvements on mcf_s.

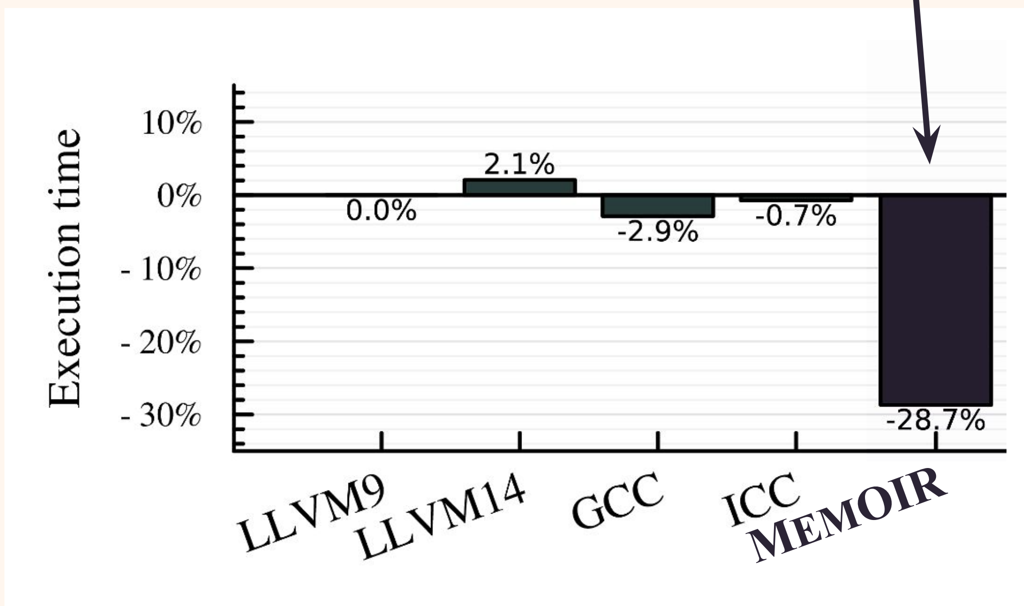


Lower is better.

Figure: Execution time of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

Evaluation

MEMOIR provides significant performance improvements with several optimizations.



Lower is better.

Figure: Execution time of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

Evaluation

Example Application: mcf_s from SPEC2017

Quick sort accounts for ***~40%*** of exec. time

```
Seq<T> sorted = qsort(in);

for (i = 0 to K)
    v = READ(sorted, i);
    if (v > threshold)
        use(v);
```

Evaluation

Example Application: mcf_s from SPEC2017

Quick sort accounts for ***~40%*** of exec. time

```
Seq<T> sorted = qsort(in);                    [0, end)

for (i = 0 to K)
    v = READ(sorted, i);
    if (v > threshold)                        [0, K)
        use(v);
```

Evaluation

Live Range Analysis **propagates liveness information**

```
Seq<T> sorted = qsort(in);

for (i = 0 to K)
    v = READ(sorted, i);
    if (v > threshold)
        use(v);
```

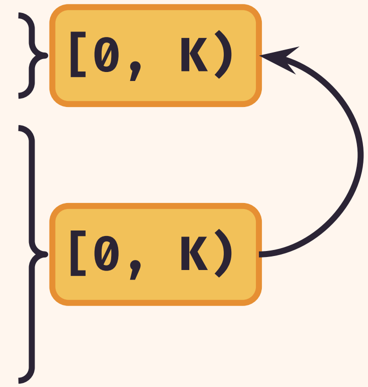
[0, end)

[0, K)

Evaluation

Live Range Analysis **propagates liveness information**

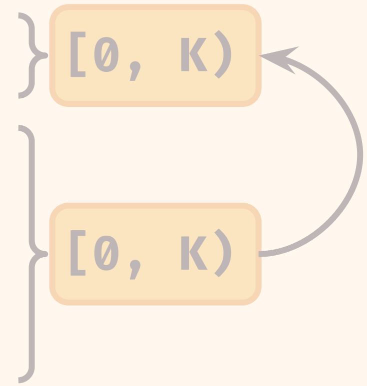
```
Seq<T> sorted = qsort(in);

for (i = 0 to K)
    v = READ(sorted, i);
    if (v > threshold)
        use(v);
```



Evaluation

Dead Element Elimination **converts sort → partial sort!**

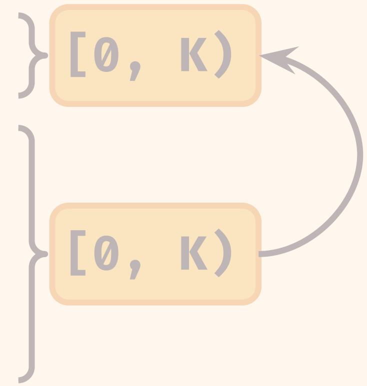
```
Seq<T> sorted = qsort'(in, 0, K);

for (i = 0 to K)
    v = READ(sorted, i);
    if (v > threshold)
        use(v);
```



Evaluation

Dead Element Elimination **converts sort → partial sort!**

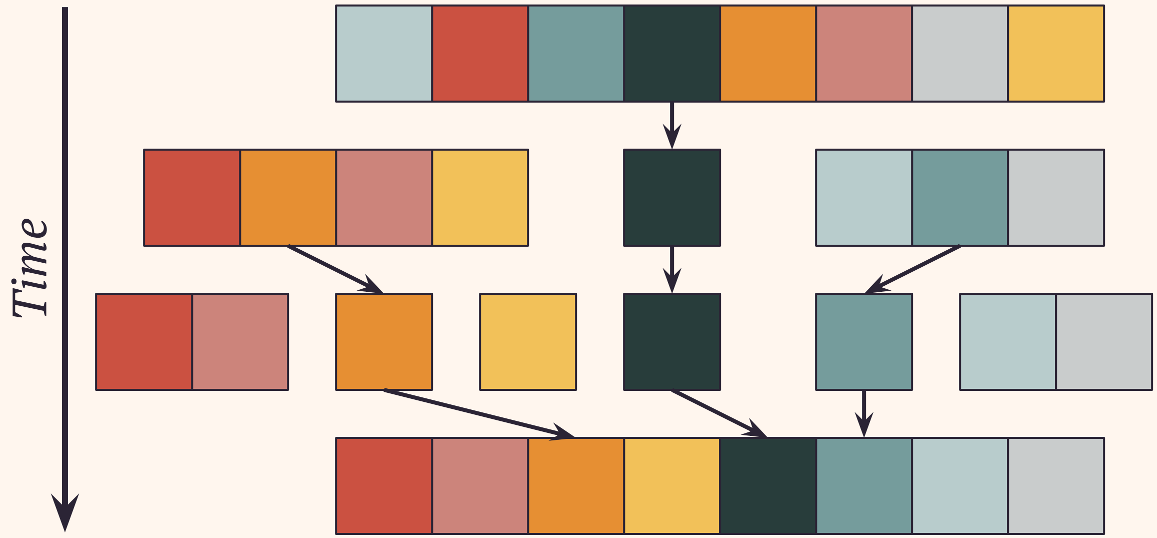
```
Seq<T> sorted = qsort'(in, 0, K);

for (i = 0 to K)
    v = READ(sorted, i);
    if (v > threshold)
        use(v);
```



Let's zoom into the sort to see how this works.

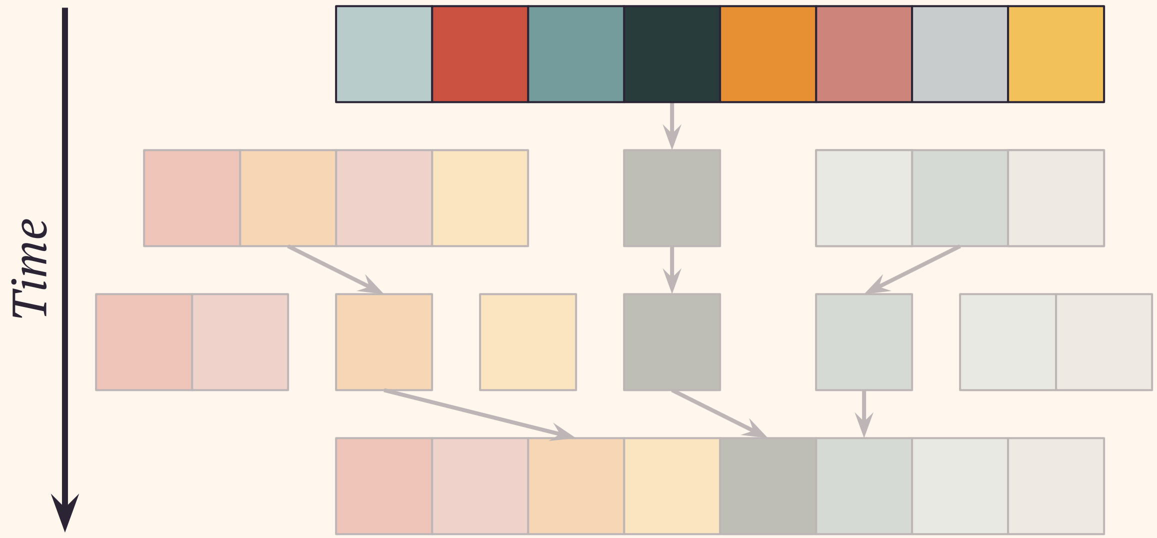
Evaluation

Dead Element Elimination converts sort → partial sort!



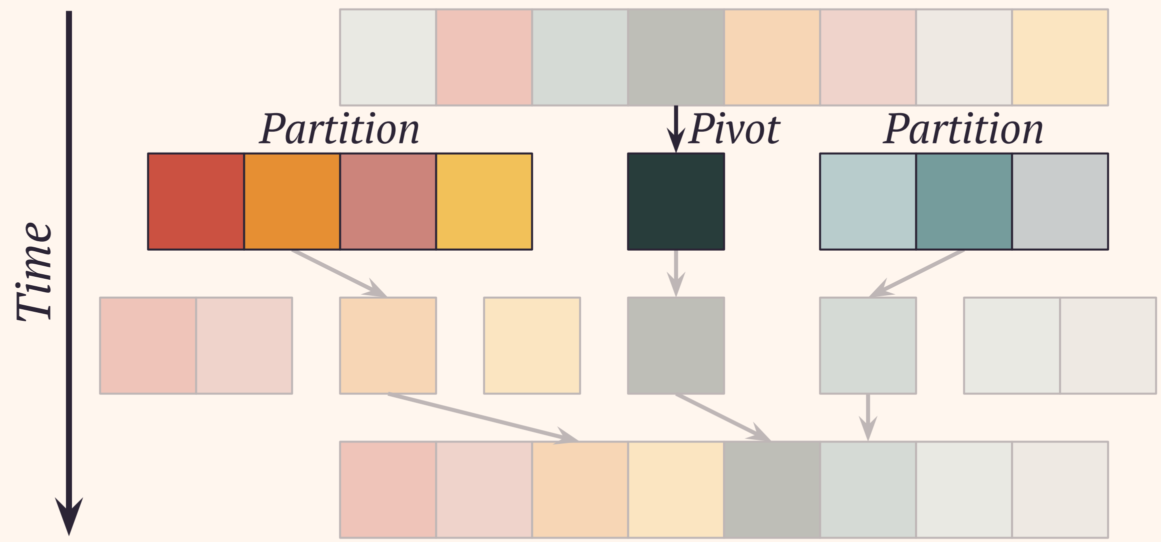
Evaluation

***Dead Element Elimination* converts sort → partial sort!**



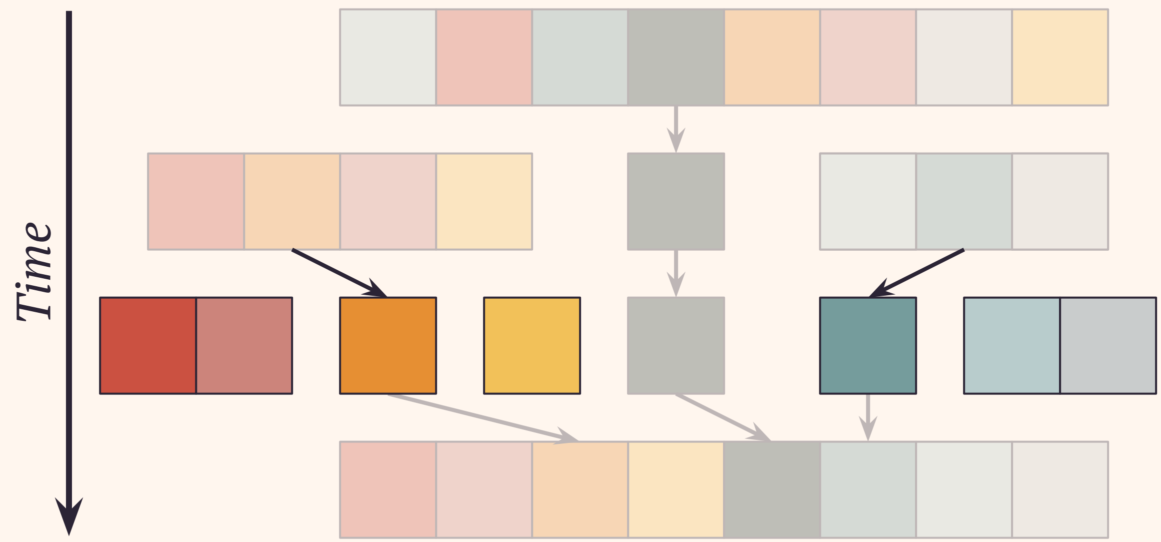
Evaluation

Dead Element Elimination converts sort → partial sort!



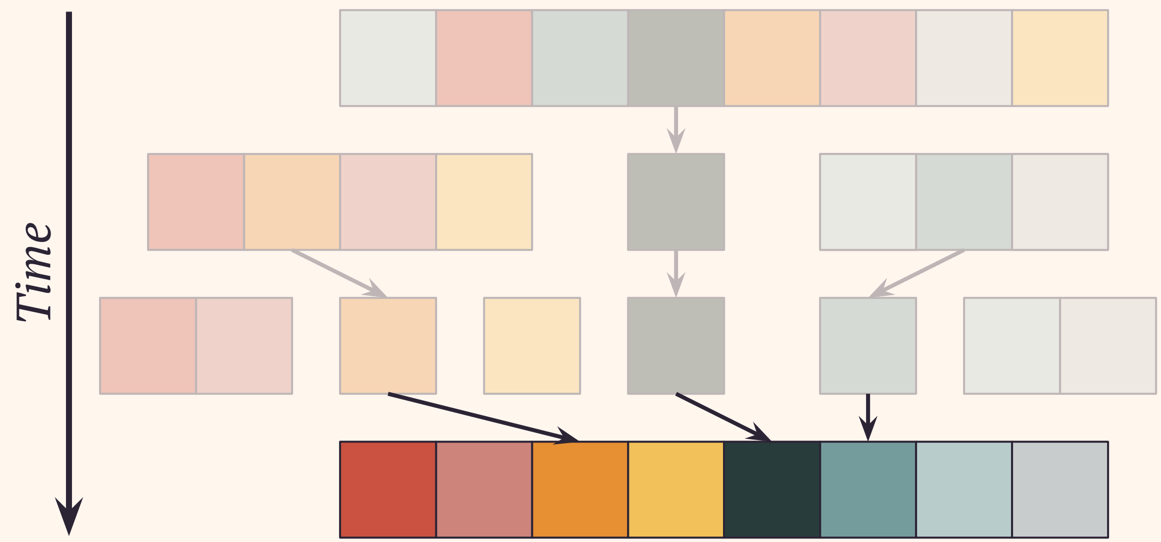
Evaluation

Dead Element Elimination **converts sort → partial sort!**



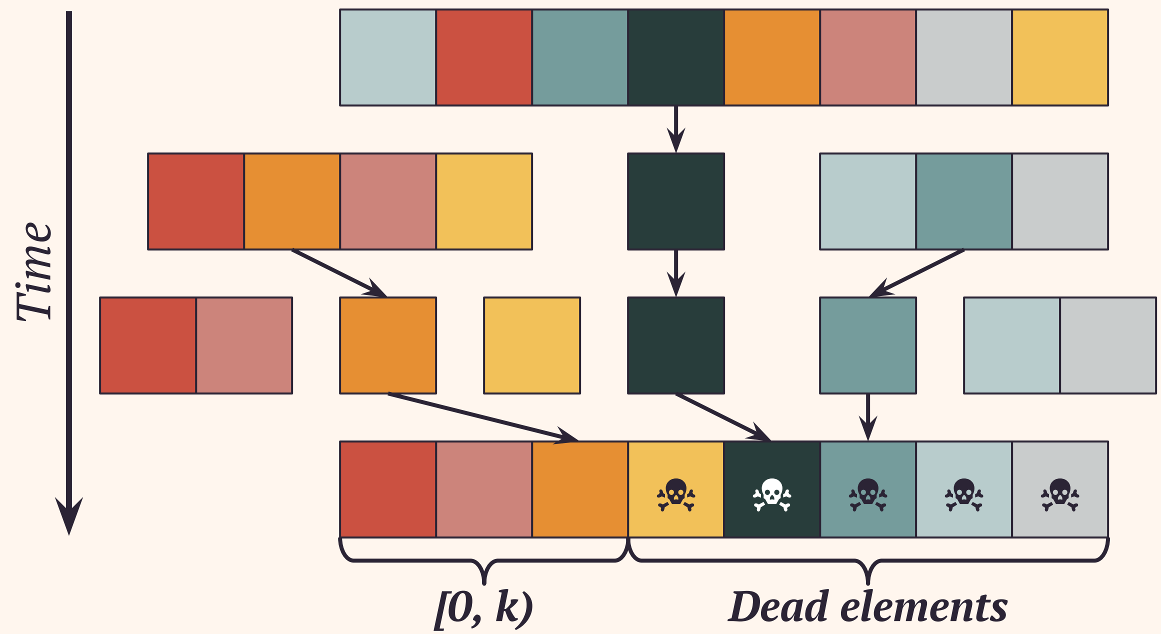
Evaluation

Dead Element Elimination **converts sort → partial sort!**

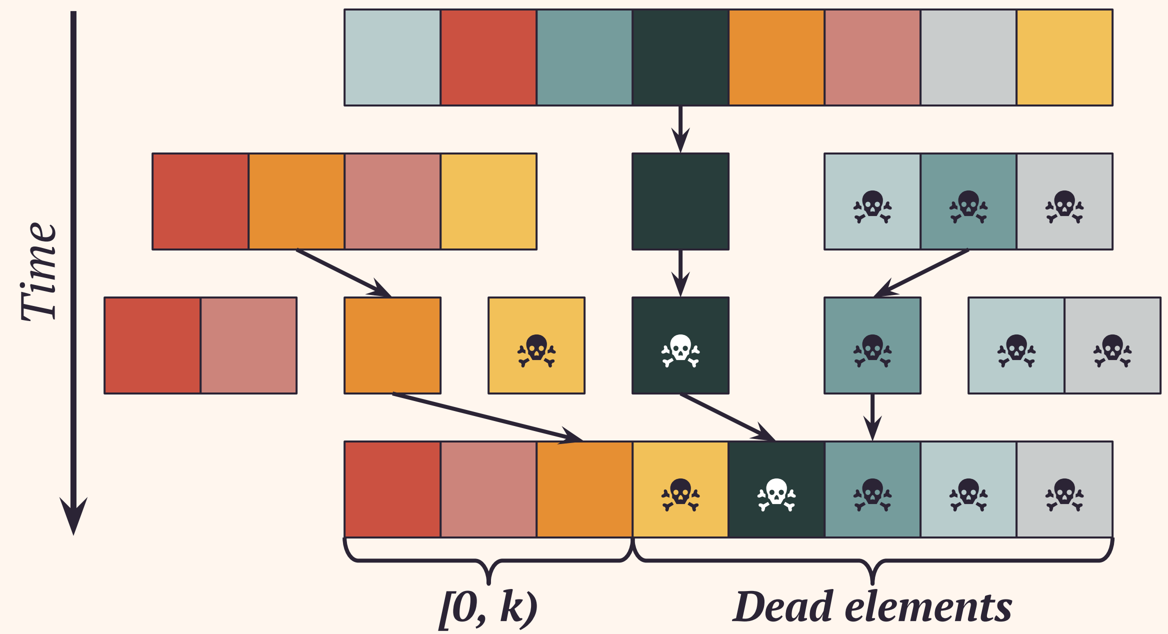


Evaluation

Dead Element Elimination converts sort → partial sort!



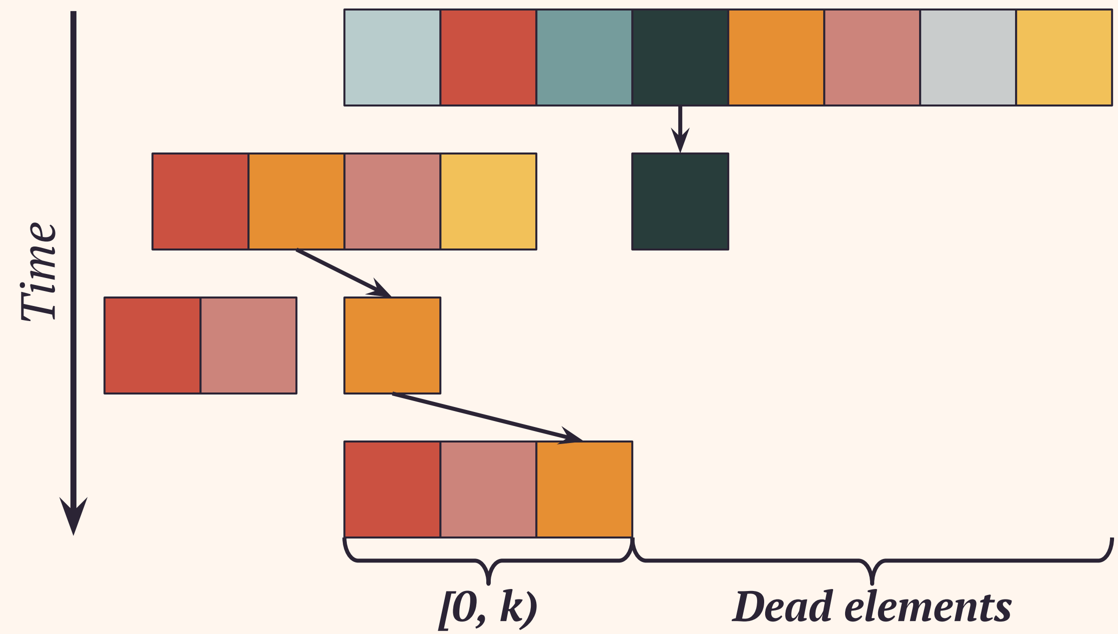
Dead Element Elimination converts sort → partial sort!



Identify and writes to dead elements

Evaluation

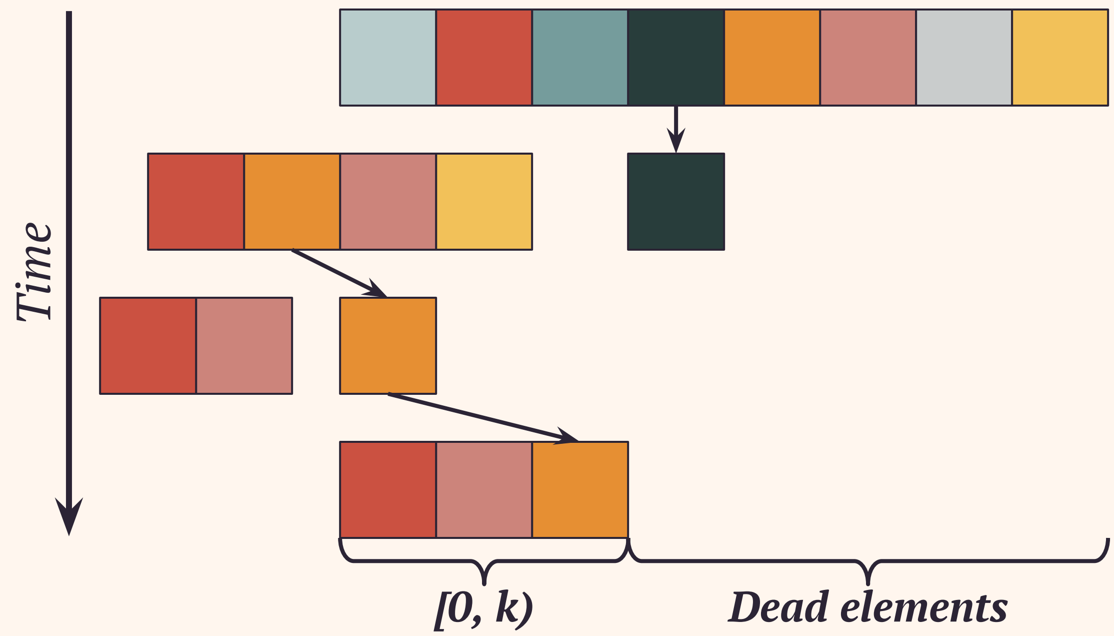
Dead Element Elimination **converts sort → partial sort!**



Eliminate writes to dead elements

Evaluation

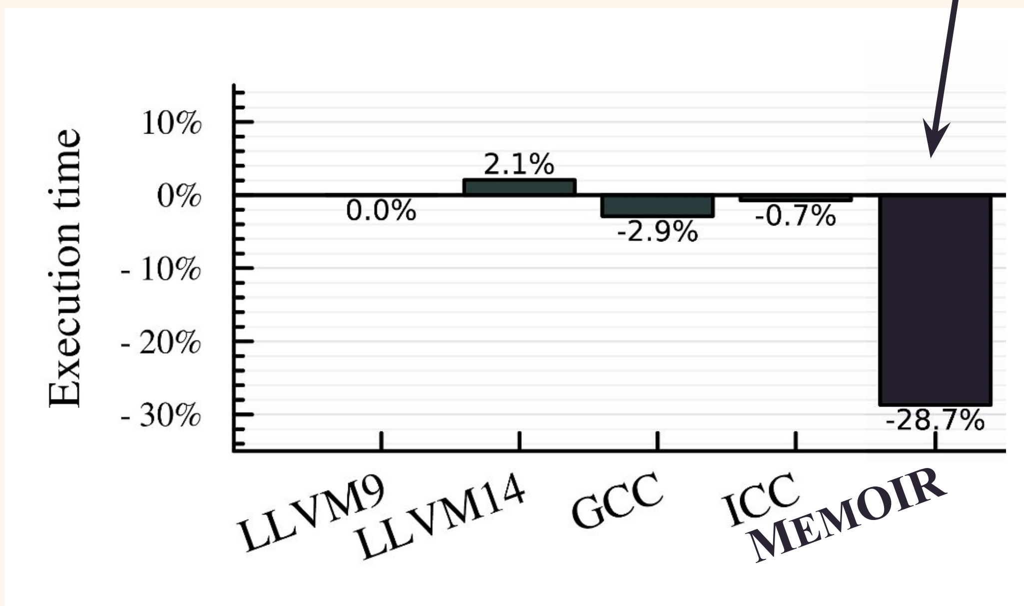
***Dead Element Elimination* converts sort → partial sort!**



No primitive knowledge of sort!

Evaluation

MEMOIR reduces *O(nlogn)* operation to $\underbrace{O(n+klogk)}$, *k << n*



Lower is better.

Figure: Execution time of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

Analysis

Takeaway

Single Reference

+

Immutability

=

Amenable to Analysis

Transformation

Enabling Transformations on Data Organization

Transformation

Data Organization Optimizations

Dead Field Elimination

Reduce memory usage

Transformation

Data Organization Optimizations

Dead Field Elimination

Reduce memory usage

Associative collection → Sequential collection

Reduce memory usage, faster access

Transformation

Data Organization Optimizations

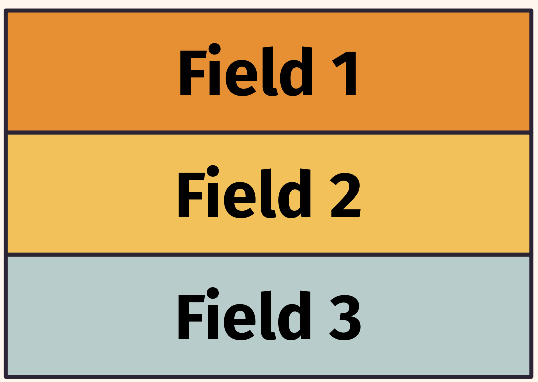
Dead Field Elimination
Reduce memory usage

Associative collection → Sequential collection
Reduce memory usage, faster access

Field(s) of objects → Associative collection
Reduce memory usage, improve locality

Optimization

Field Elision




```
type T = { a: i32, b: i32,
           c: i32 }
x = new Seq<T>(N)
for (i=0..N) {
  x[i].a = ...
  x[i].b = ...
  if ( ... )
    x[i].c = ...
  }
}
```

Optimization

Field Elision

```
type T = { a: i32, b: i32 }
c = new Assoc<&T, i32>(N)
x = new Seq<T>(N)
for (i=0..N) {
  x[i].a = ...
  x[i].b = ...
  if ( ... )
    c[x[i]] = ...
}
```

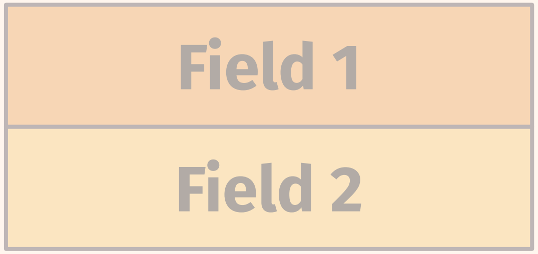
Field 1

Field 2

Replace the field with an *associative collection*

Optimization

Field Elision




```
type T = { a: i32, b: i32 }
c = new Assoc<&T, i32>(N)
x = new Seq<T>(N)
for (i=0..N) {
  x[i].a = ...
  x[i].b = ...
  if ( ... )
    c' = INSERT(c, x[i])
}
```

Update the field uses

Optimization

Field Elision

Field 1

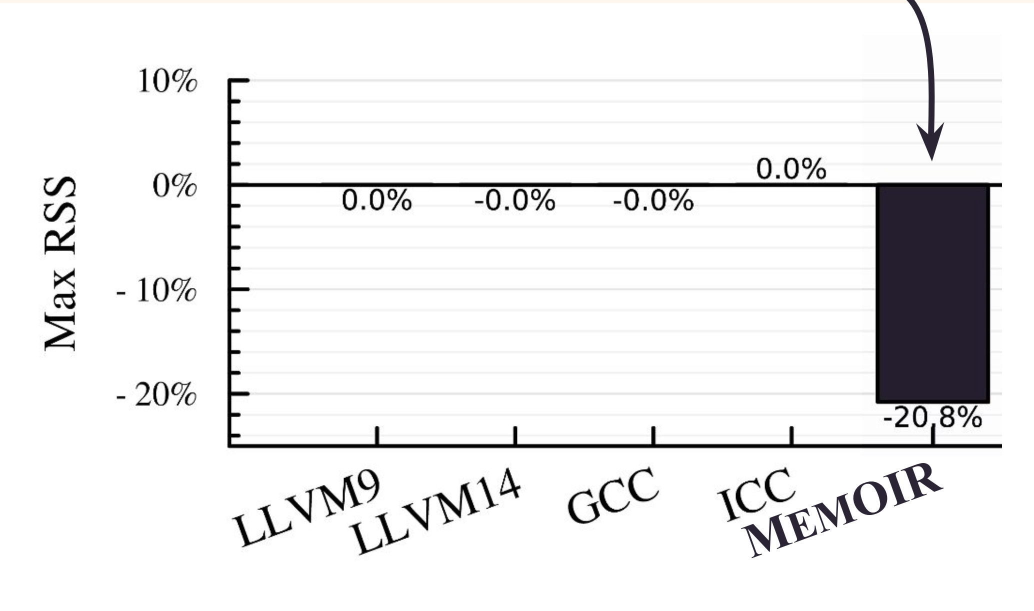
Field 2

```
type T = { a: i32, b: i32 }
c = new Assoc<&T, i32>(N)
x = new Seq<T>(N)
for (i=0..N) {
  x[i].a = ...
  x[i].b = ...
  if (...)
    c' = INSERT(c, x[i])
}
```

Benefit:* Field is only allocated *if needed!

Evaluation

MEMOIR optimizations ***reduce memory usage!***



Lower is better.

Figure: Maximum resident set size of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

Takeaway

Decoupling storage from organization

+

Referential Transparency

=

Amenable to Transformation

Overview

Representing Data Collections in the Compiler

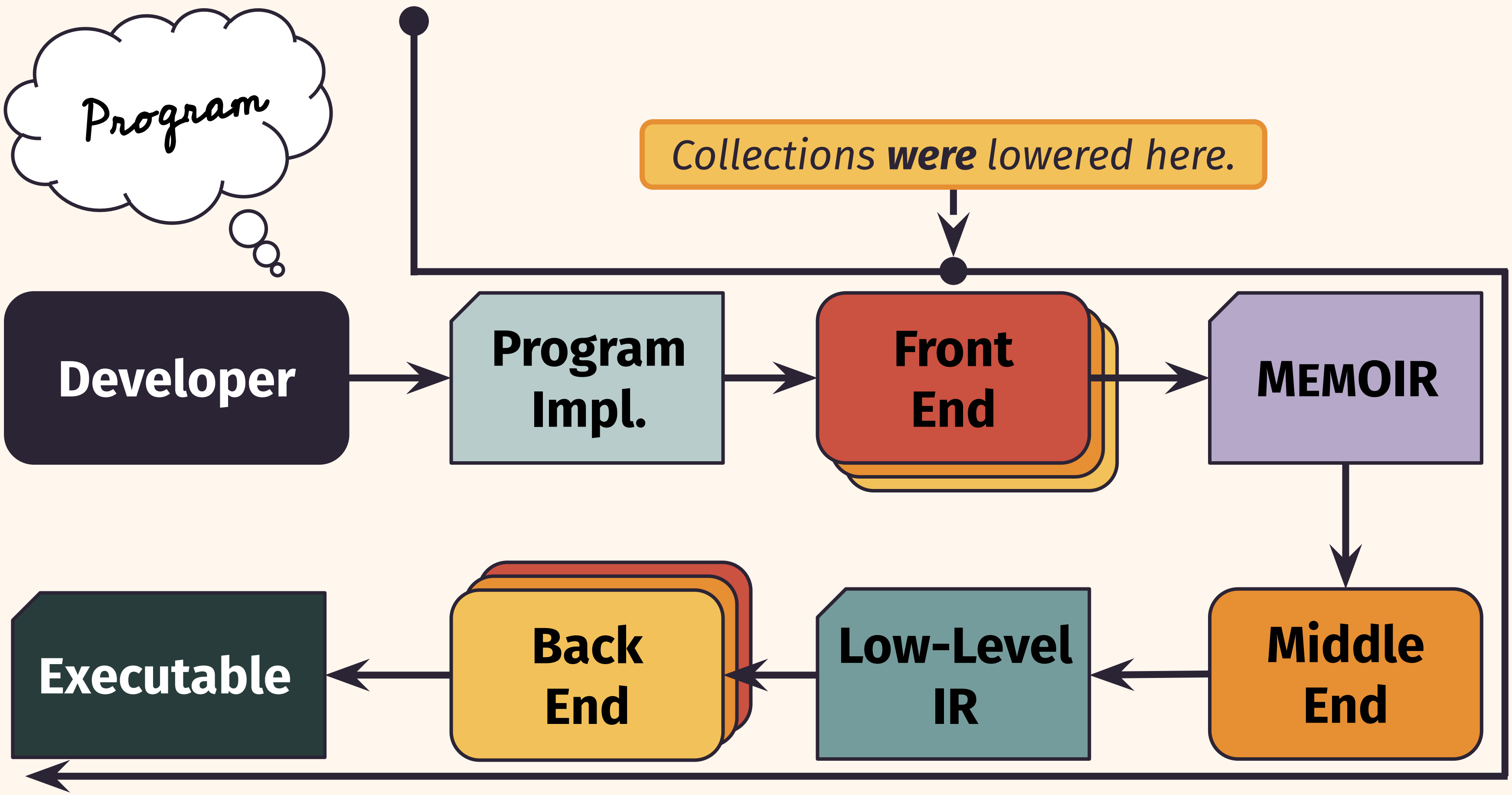
General-purpose ✓

Amenable to Analysis ✓

Amenable to Transformation ✓

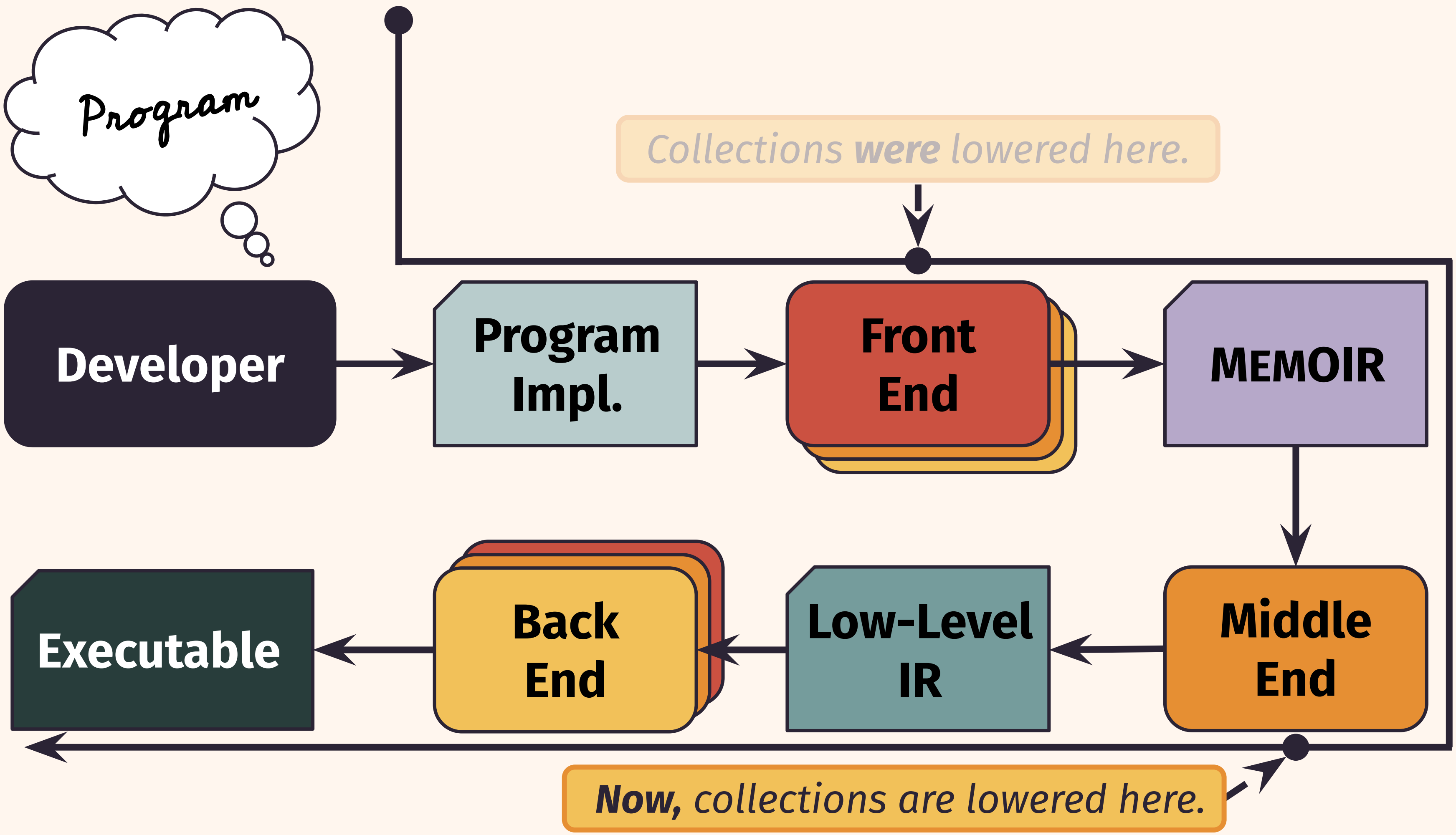
Conclusion

What is a *Collection-Oriented Definition* of Compilation?



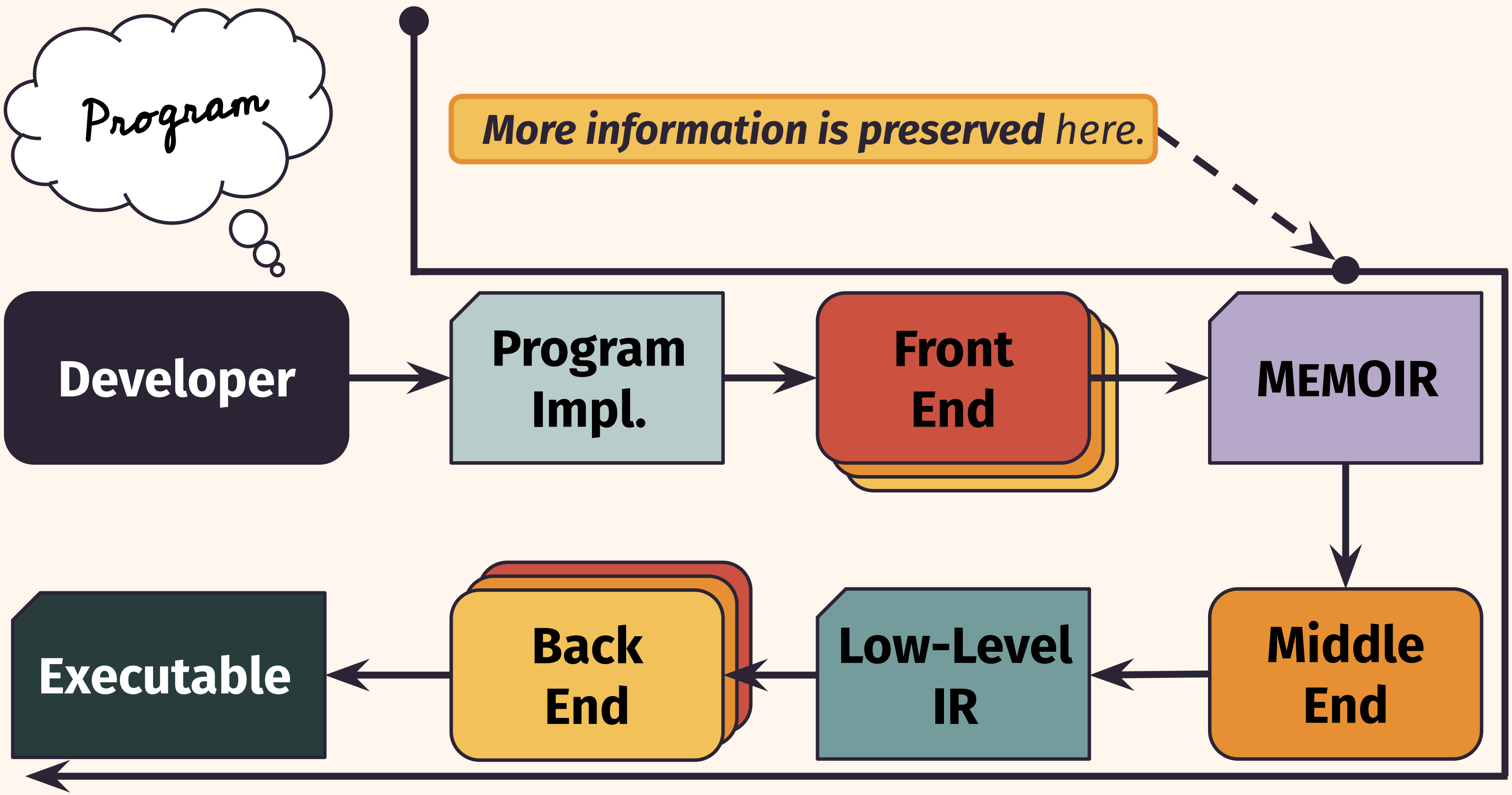
Conclusion

What is a *Collection-Oriented Definition* of Compilation?



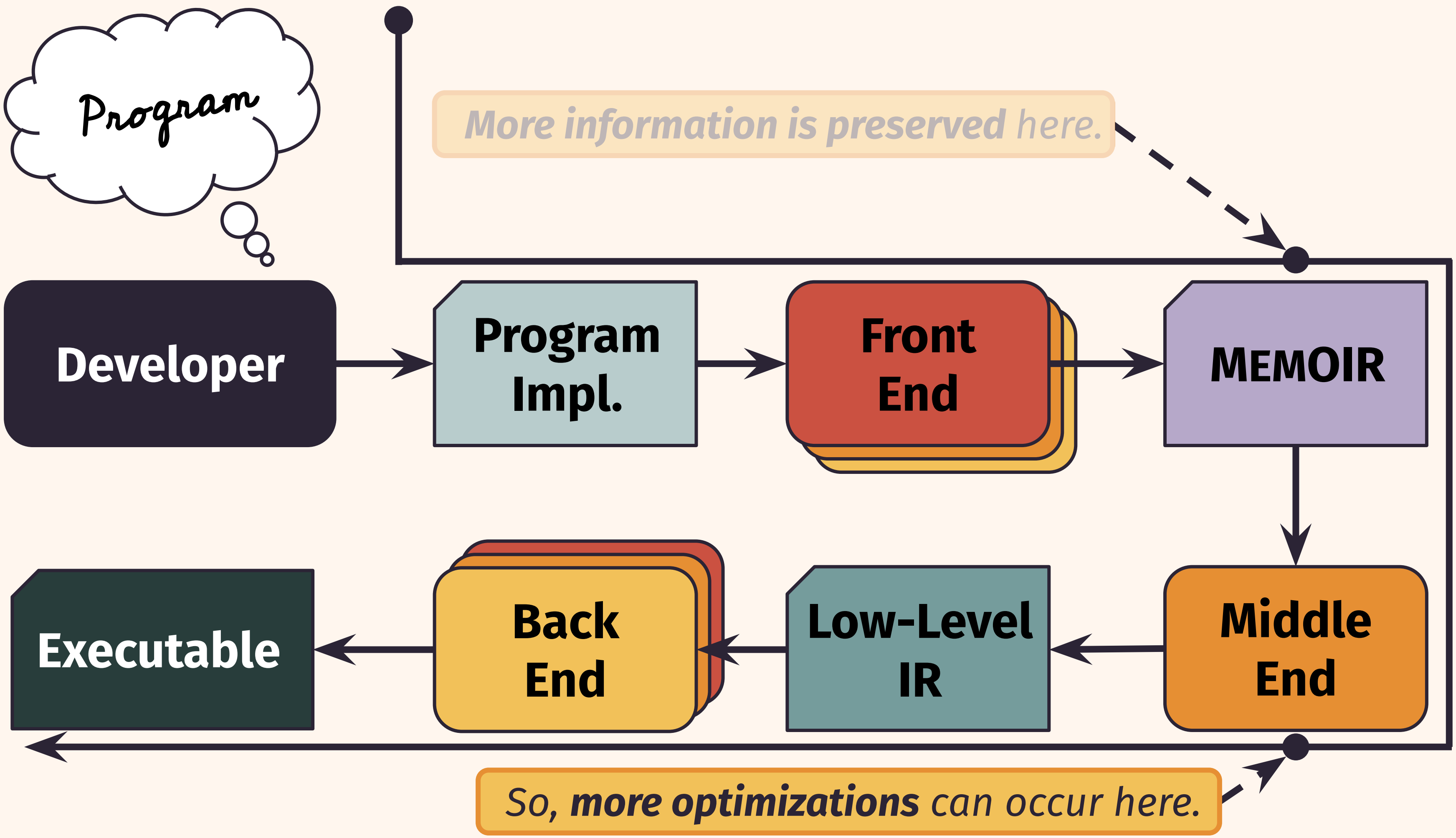
Conclusion

What is a *Collection-Oriented Definition* of Compilation?



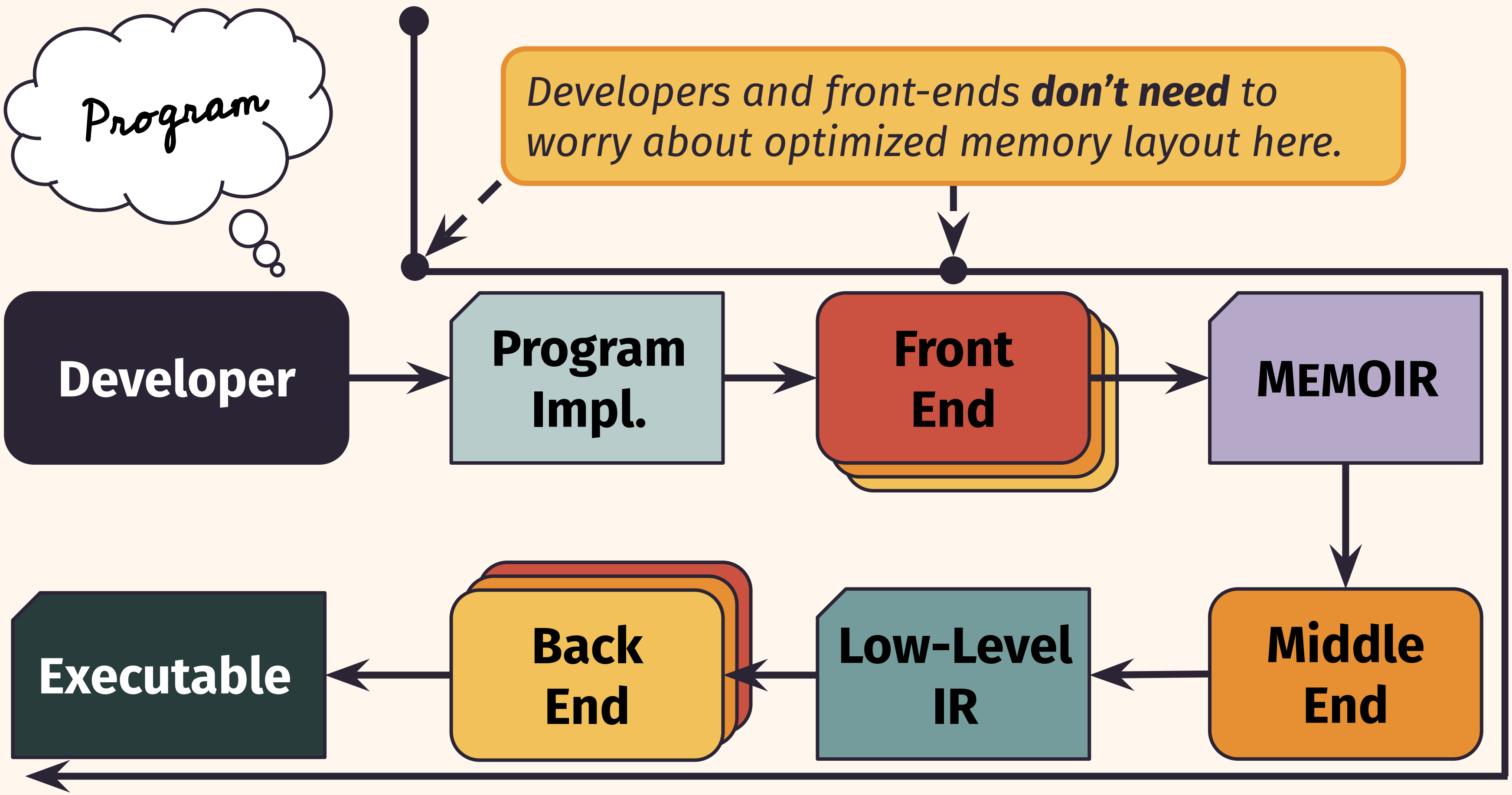
Conclusion

What is a ***Collection-Oriented Definition*** of Compilation?



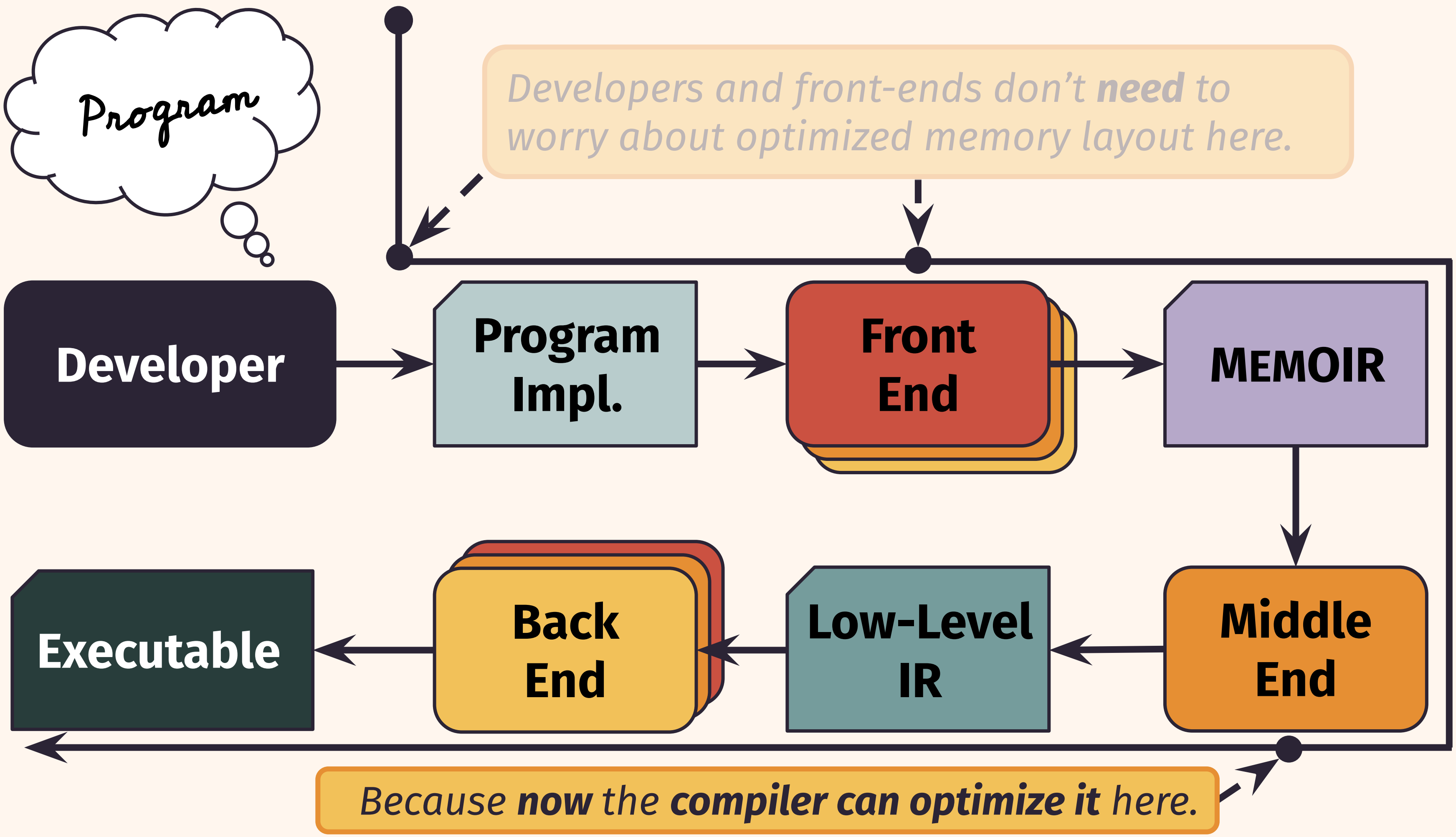
Conclusion

What is a *Collection-Oriented Definition* of Compilation?



Conclusion

What is a *Collection-Oriented Definition* of Compilation?



Conclusion

How can I use MEMOIR *today*?

github.com/arcana-lab/memoir

Write a pass with our *open source* compiler

LLVM

NOELLE

Conclusion

How can I use MEMOIR *today*?

github.com/arcana-lab/memoir

Write a pass with our *open source* compiler

- **LLVM**
- **NOELLE**

Write a program using the MEMOIR toolchain

- **C**
- **C++**

Conclusion

How can I use MEMOIR *today*?

github.com/arcana-lab/memoir

Write a pass with our *open source* compiler

- LLVM
- NOELLE

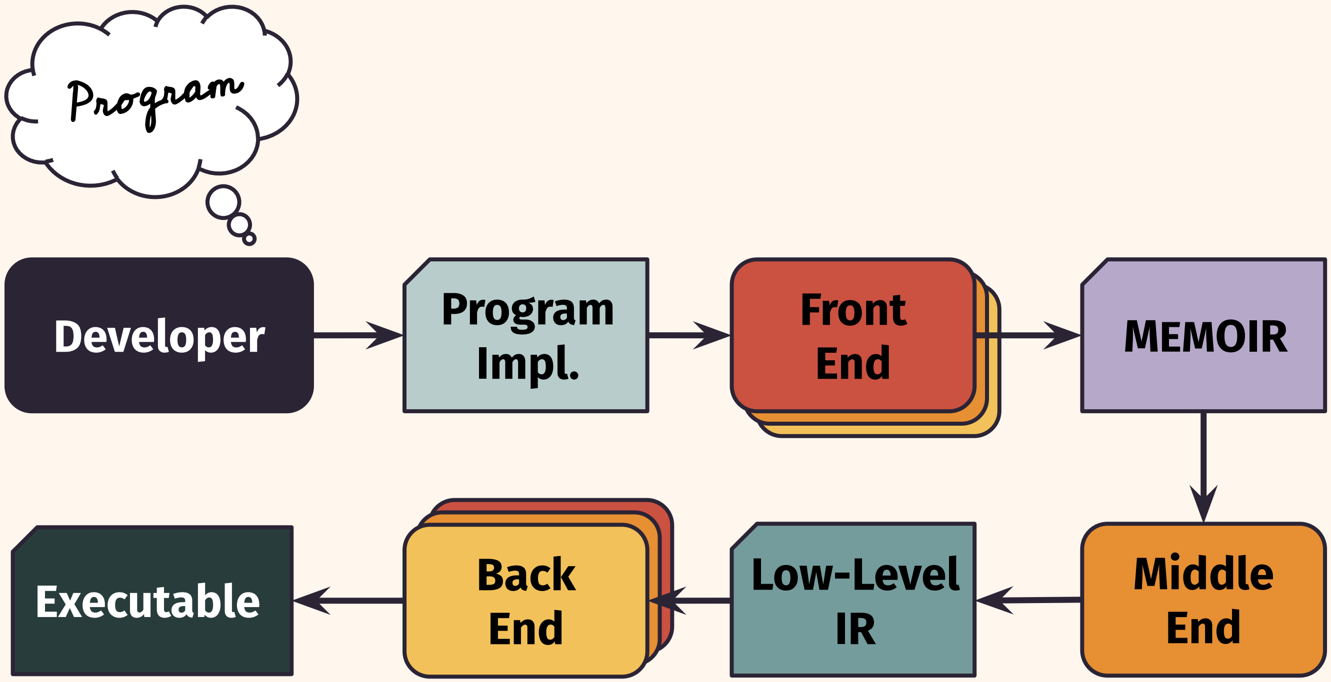
Write a program using the MEMOIR toolchain

- C
- C++
- Rust
- Mojo
- …

More to come!

Representing Data Collections for Analysis and Transformation

Tommy McMichen



www.mcmichen.cc

MEMOIR Repository

Northwestern University

ARCANA
Rethinks Compiler Abstractions for New Applications

memoir

Motivations

Most Heap Memory is for Structured Data

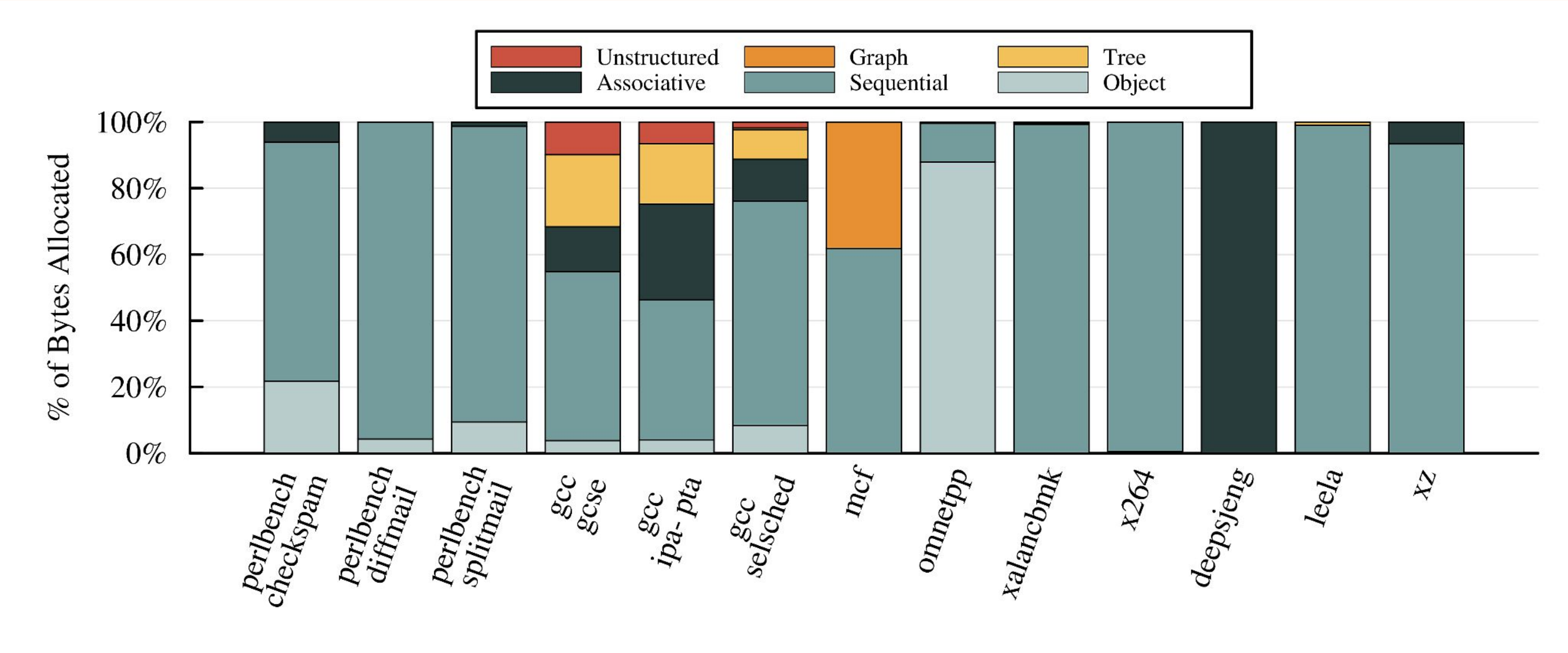


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

Motivations

Most Reads from Heap Memory are for Structured Data

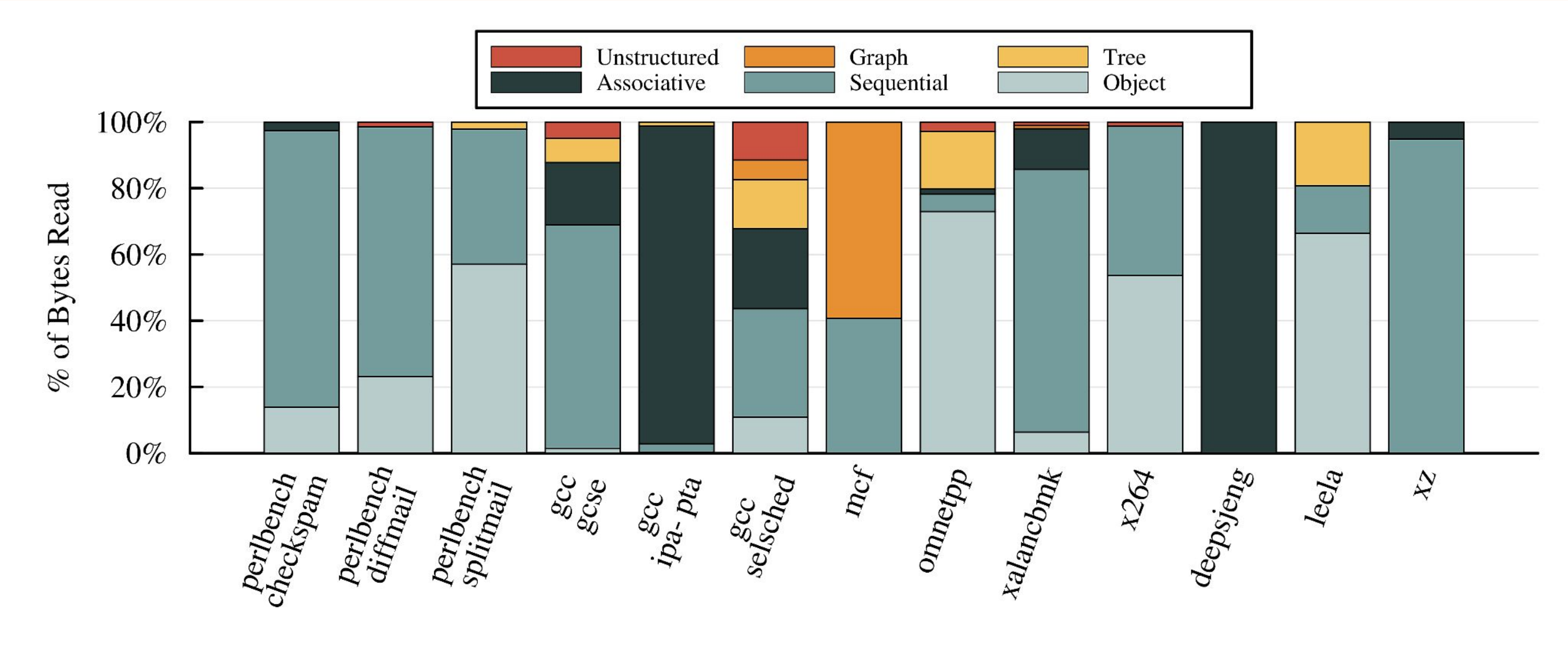


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

Motivations

Most Writes to Heap Memory are for Structured Data

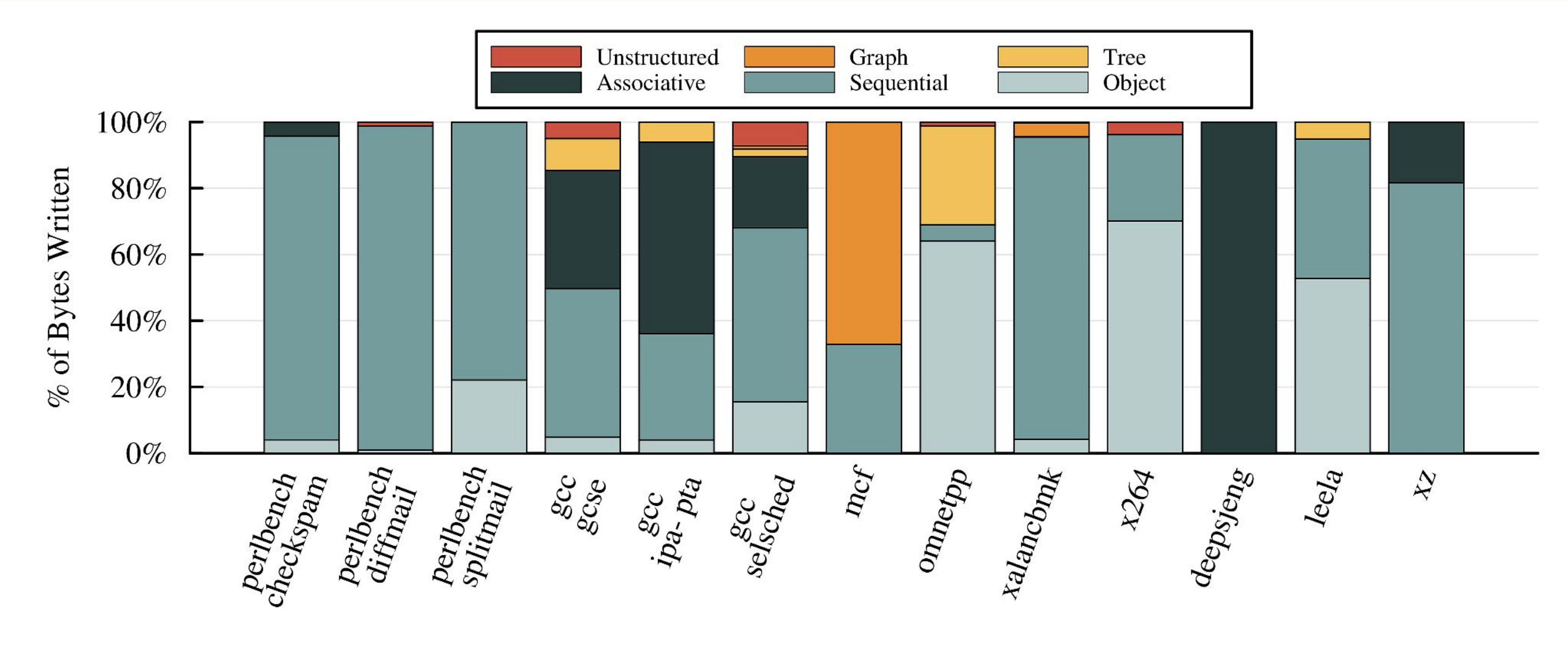


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

MEMOIR requires reasonable compilation time

	Compile Time (ms)			
	MEMOIR		LLVM	
Benchmark	`-O0`	`-O3`	`-O0`	`-O3`
`mcf`	70.6	776.4	20.9	663.2
`deepsjeng`	246.0	1867.6	34.8	852.8
`LLVM opt`	225.9	668.4	52.0	414.7

mcf_s Execution Time with Pass Breakdown

Lower is better.

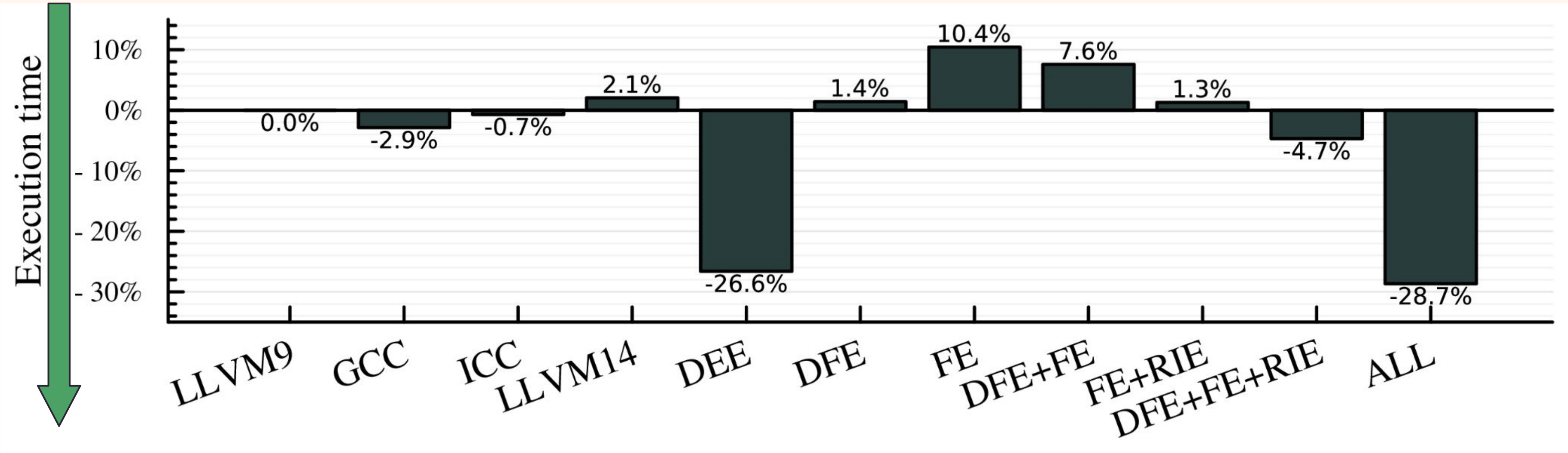


Figure 3: Execution time of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

mcf_s Max RSS with Pass Breakdown

Lower is better.

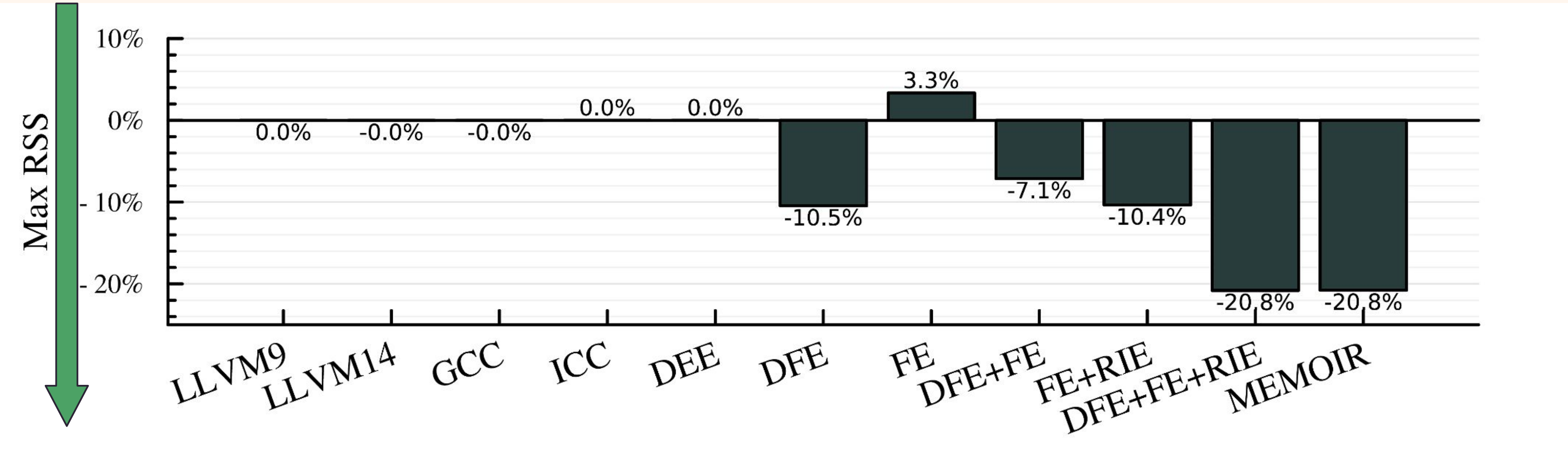
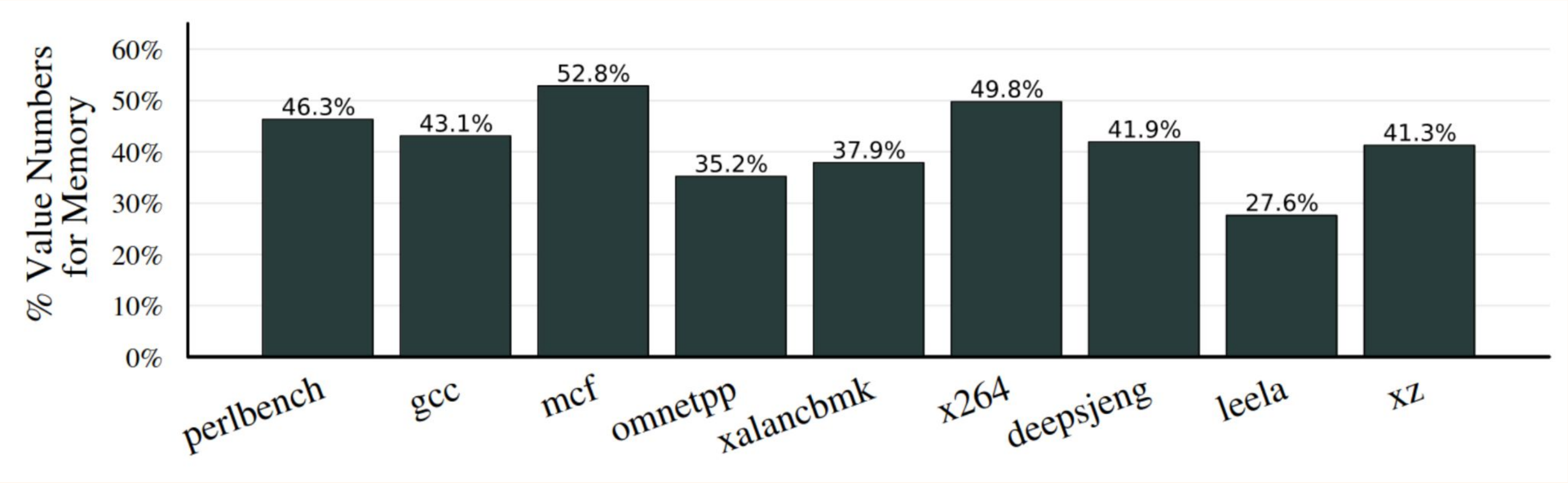
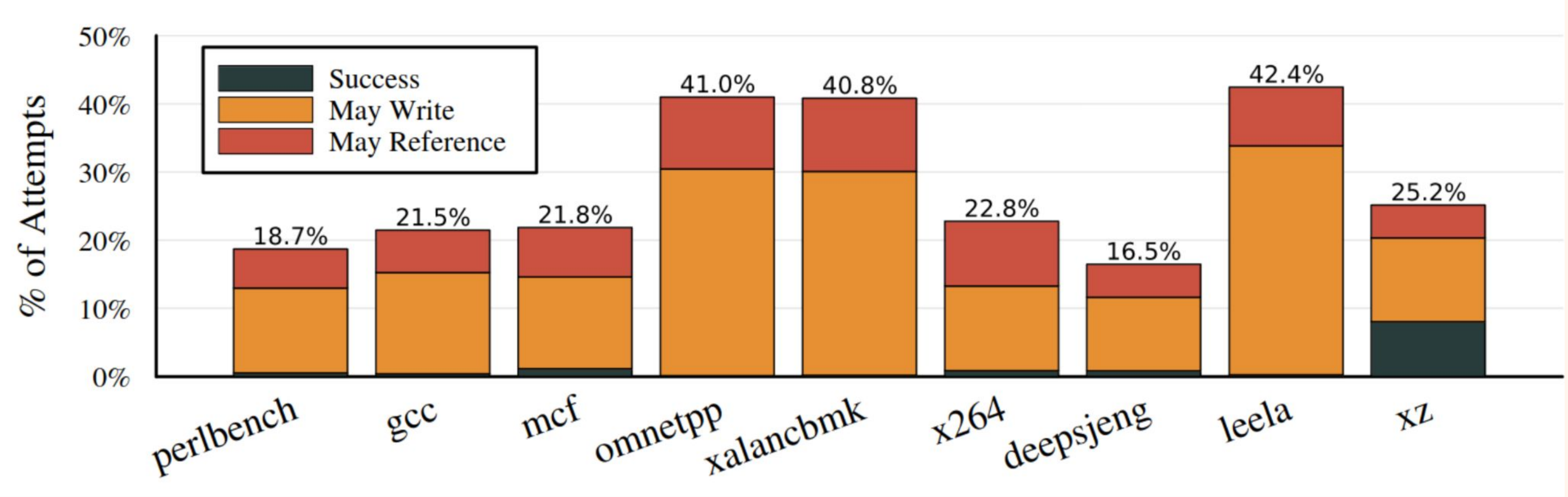


Figure: Maximum resident set size usage of `mcf_s` with `refspeed` input. 10 trials. Normalized to `LLVM9`.

Global Value Numbering is conservative because of memory operations



Sink is commonly blocked by memory operations



mcf_s parallel speedup with DEE optimization