An Introduction to MEMOIR

Tommy McMichen

Advanced Topics in Compilers

Northwestern University

ARCANA
Rethinks Compiler Abstractions for New Applications

CONSTELLATION PROJECT

m&moir

Overview

Bird's Eye View

MEMOIR is a

Bird's Eye View

MEMOIR is a

Compiler Intermediate Representation

Bird's Eye View

MEMOIR is a

Compiler Intermediate Representation

for

Data Collections and Objects

Bird's Eye View

MEMOIR is a

Compiler Intermediate Representation

for

Data Collections and Objects

in an

SSA Form

Outline

What is a Data Collection?

Outline

What is a Data Collection?

What is SSA?

Outline

What is a Data Collection?

What is SSA?

How can I analyze it?

Outline

What is a Data Collection?

What is SSA?

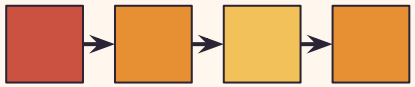
How can I analyze it?

How can I transform it?

Data Collection

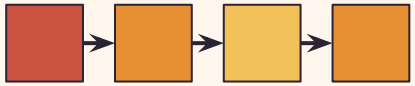
A logical organization of data

Examples

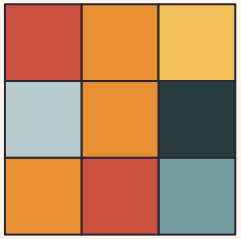


List

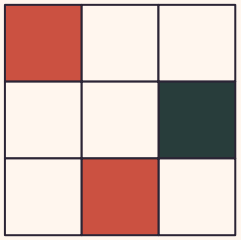
Examples



List

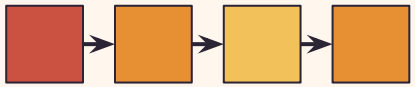


Dense Array

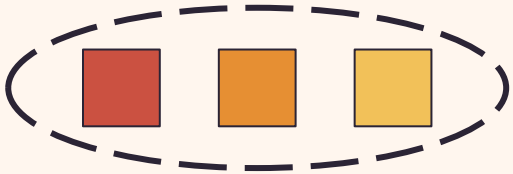


Sparse Array

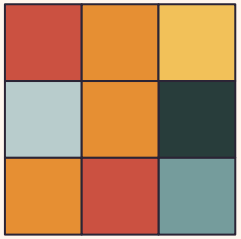
Examples



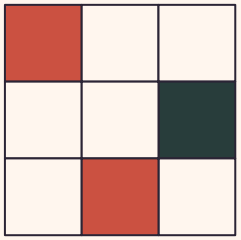
List



Set

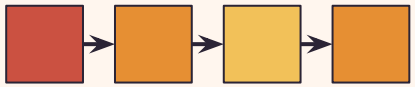


Dense Array

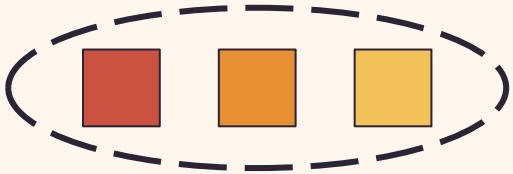


Sparse Array

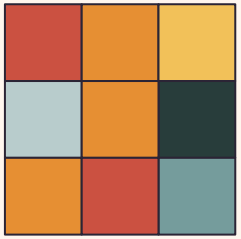
Examples



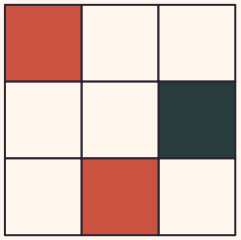
List



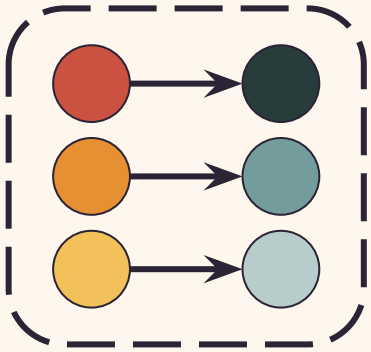
Set



Dense Array

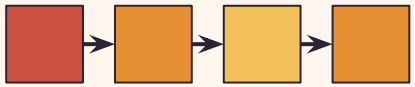


Sparse Array

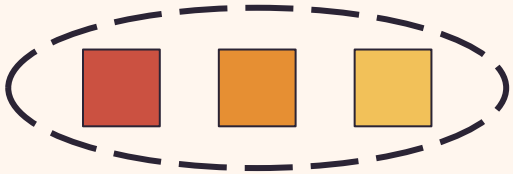


Map

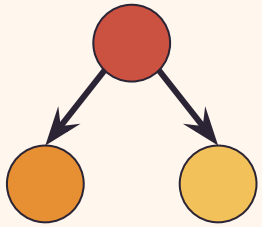
Examples



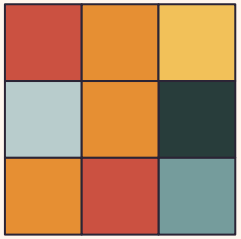
List



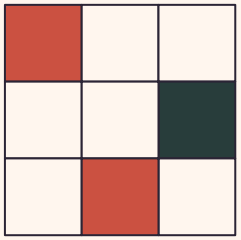
Set



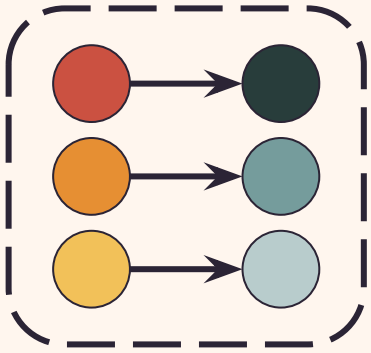
Tree



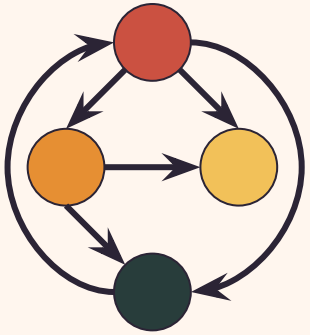
Dense Array



Sparse Array

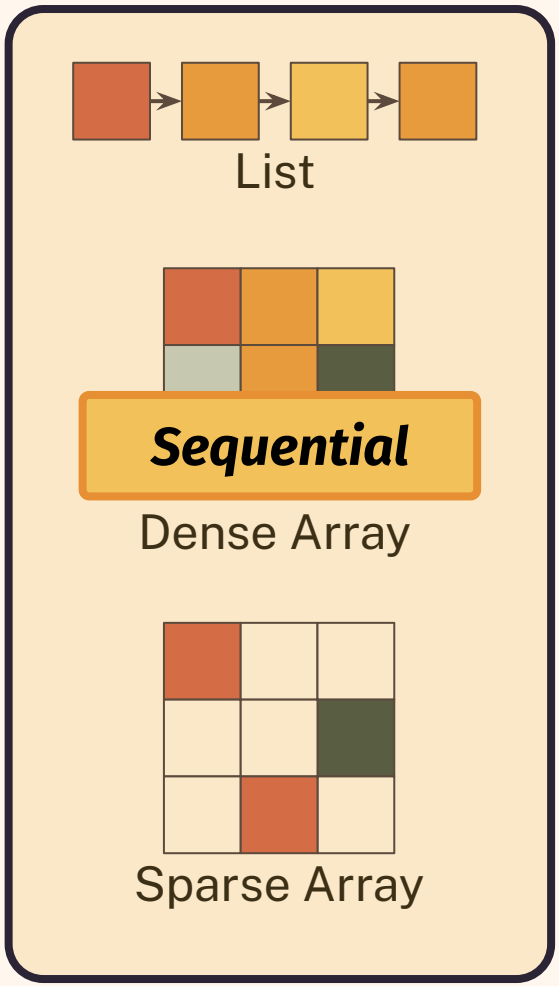


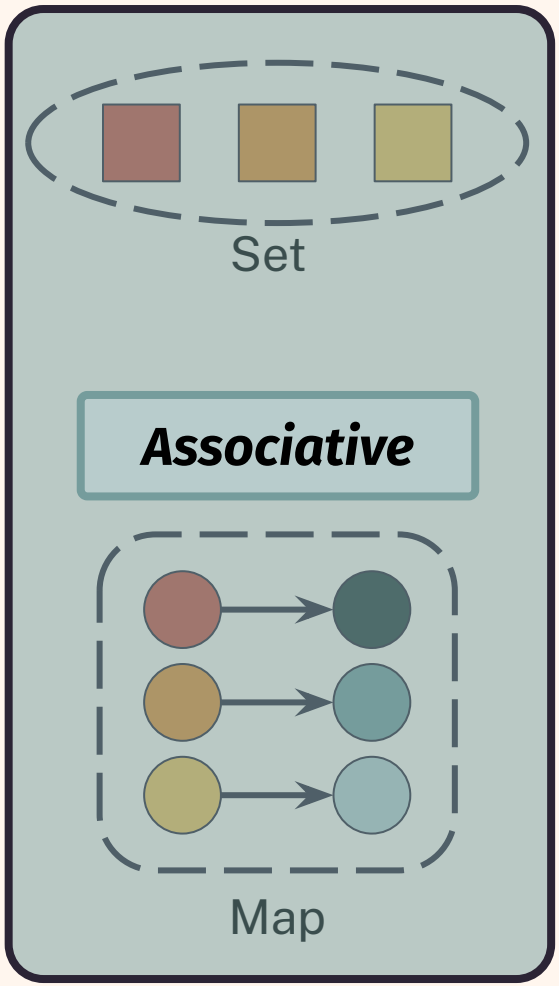
Map

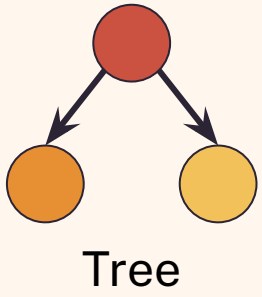


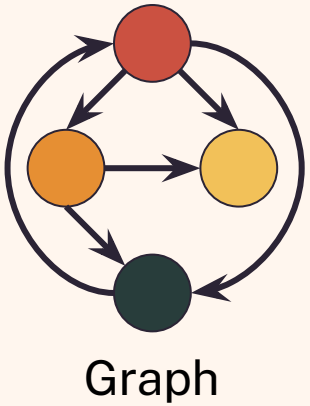
Graph

Examples

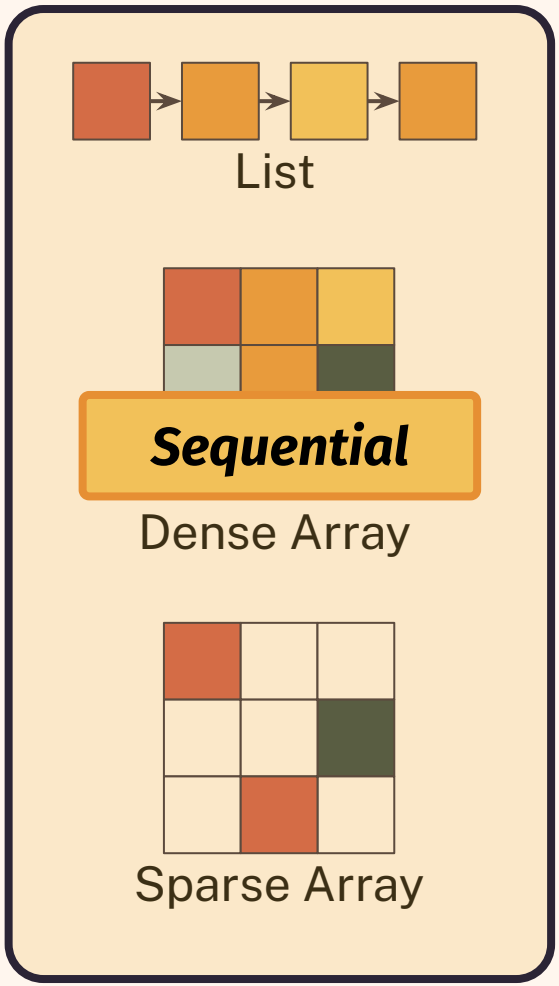

List
Sequential
Dense Array
Sparse Array

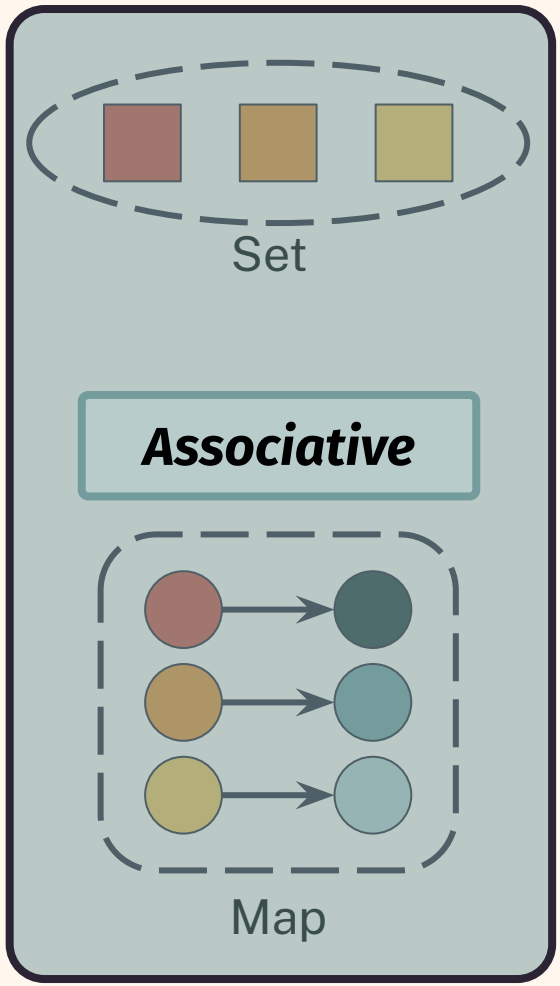

Set
Associative
Map

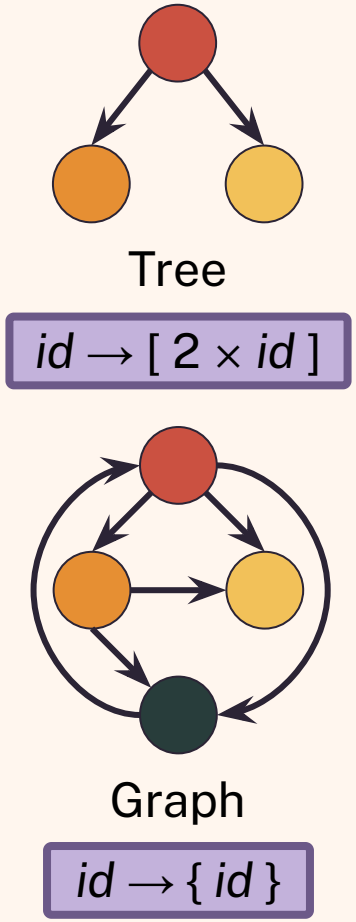

Tree


Graph

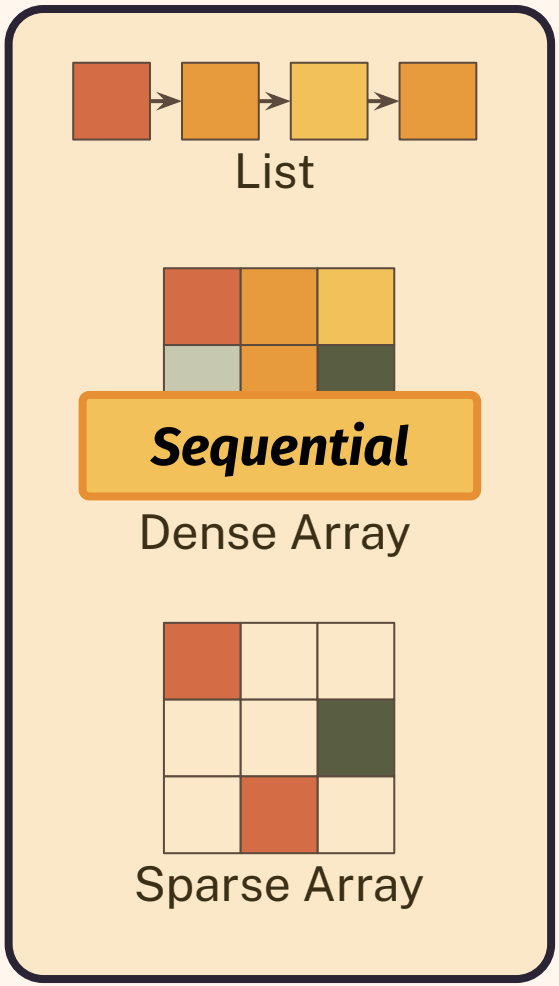
Examples

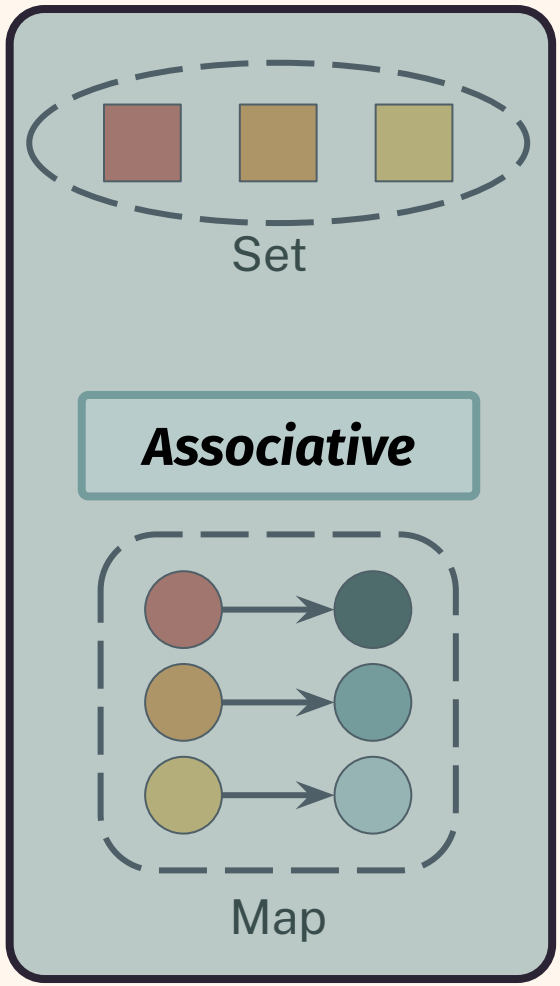

List
Sequential
Dense Array
Sparse Array

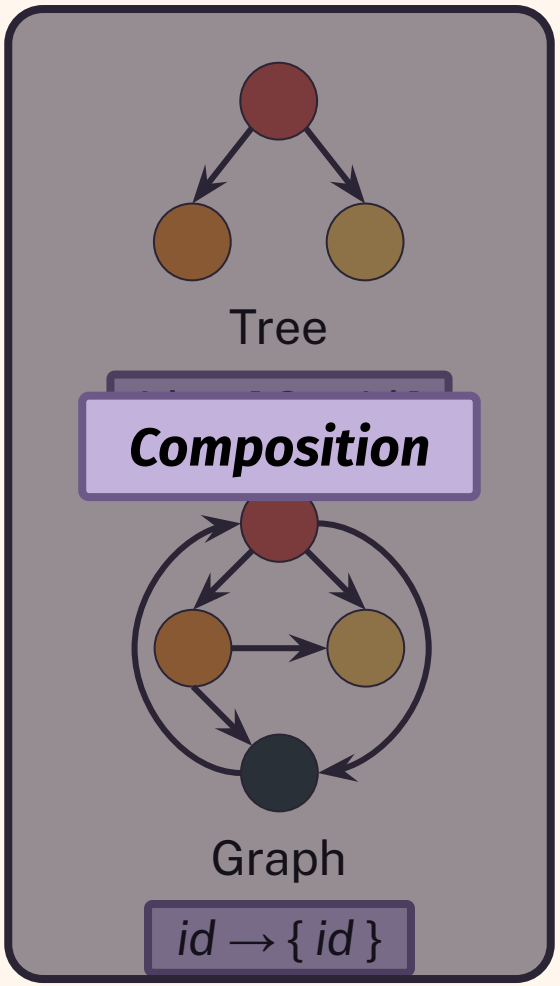

Set
Associative
Map


Tree
$id \rightarrow [\ 2 \times id\ ]$
Graph
$id \rightarrow \{\ id\ \}$

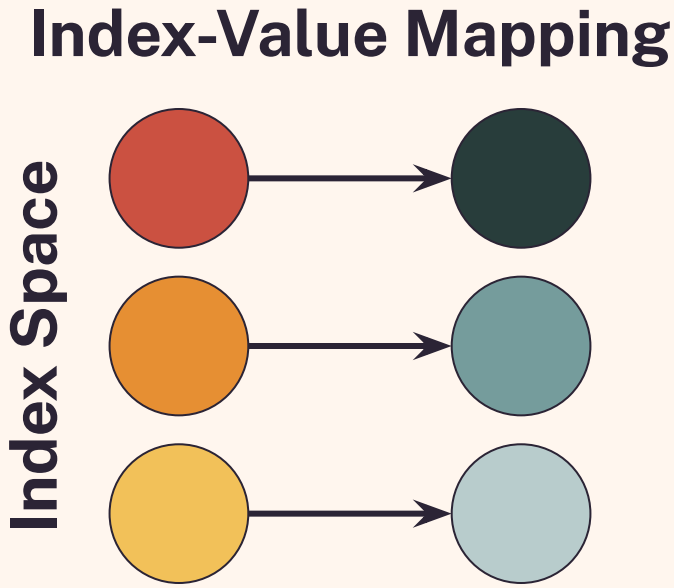
Examples


List
Sequential
Dense Array
Sparse Array

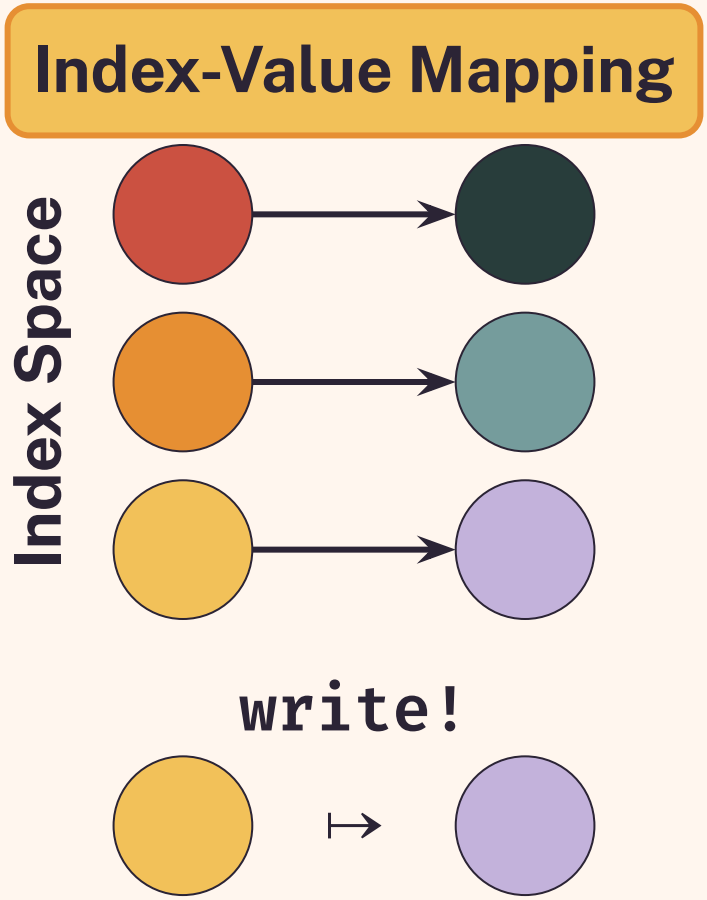

Set
Associative
Map


Tree
Composition
Graph
id → { id }

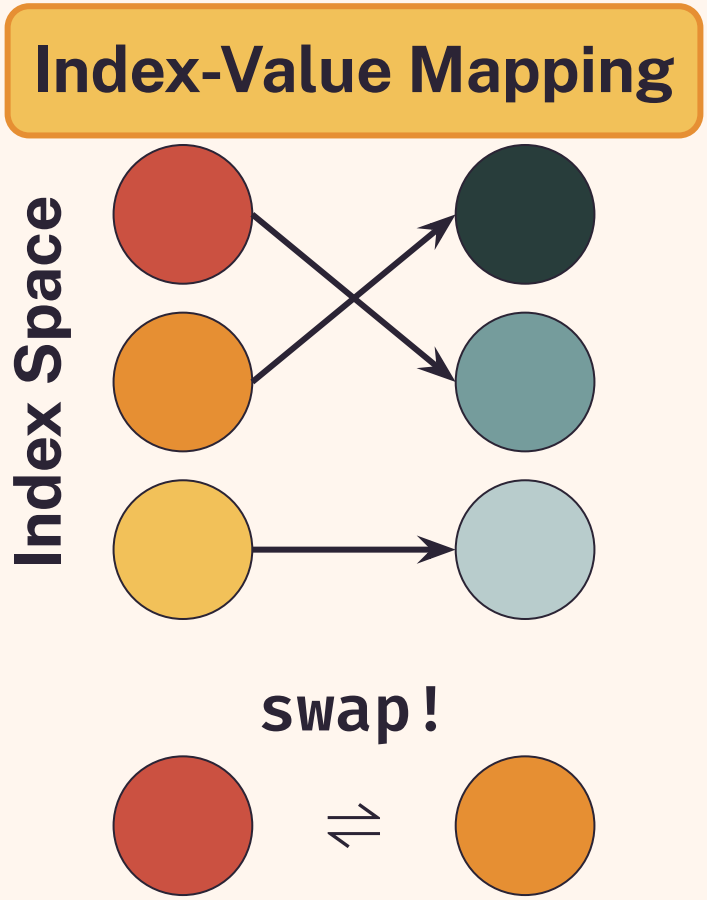
Representation



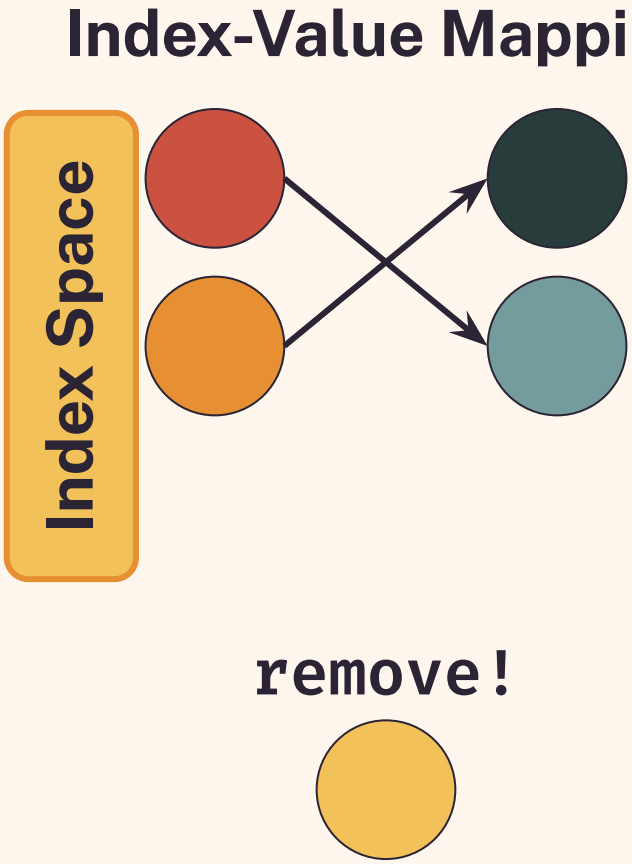
Operations on the *Index-Value Mapping*



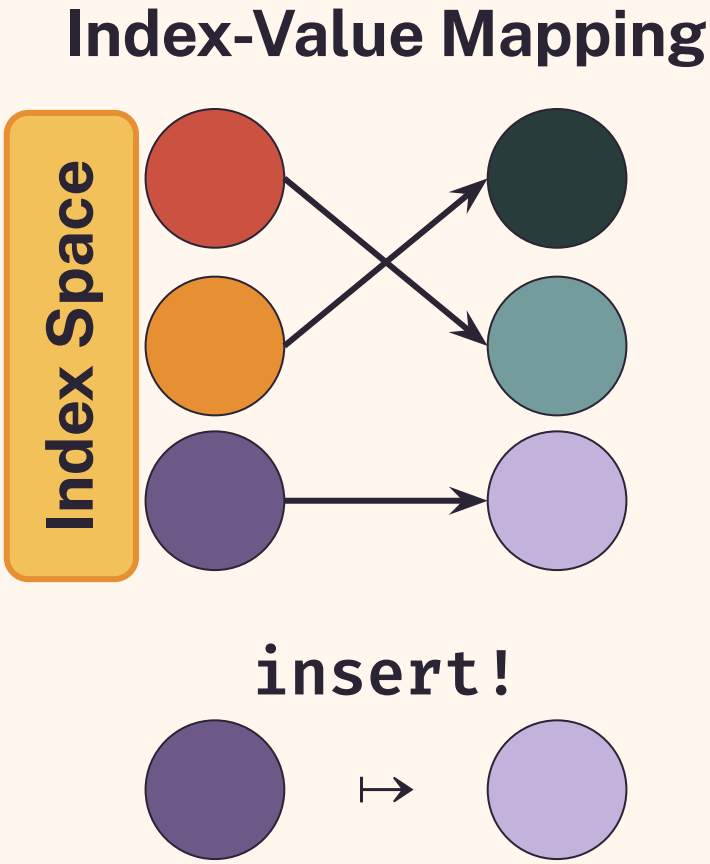
Operations on the *Index-Value Mapping*



Operations on the *Index Space*



Operations on the *Index Space*



Static Single Assignment

A language constraint, where *each variable* has a *single definition* in the static program.

Why SSA?

↓

Referential Transparency

Referential Transparency

Replacing a *subexpression* with an *equivalent one* produces an *equivalent expression*.

Referential Transparency

She lives in Chicago

$\asymp$

She lives in the largest city in Illinois

Referential Opacity

‘*Chicago*’ contains seven letters

$\not\asymp$

‘*The largest city in Illinois*’ contains seven letters

Benefits of Referential Transparency

A variable's value is ***independent of its position in the program***

Benefits of Referential Transparency

A variable's value is ***independent of its position in the program***

Information attached to the ***definition of a variable*** is true for ***all uses of the variable***

So, what about data collections?

```
write!(c, i, v)
```

Semantics: Following this operation, `c[i] = v`

So, what about data collections?

`write!(c, i, v)`

Semantics: Following this operation, `c[i] = v`

But, is `read(c, i)` $\asymp$ `v`?

Not necessarily! Depends on its position in the program.

So, what about data collections?

```
write!(c, i, v)
```

```
read(c, i)
```

Yep! The use of **`c`** is dominated by the **`write!`** and is not dominated by any other **`write!`** to **`c`**

So, what about data collections?

```
write!(c, i, v)

write!(c, i, w)

read(c, i)
```

Nope! The use of **`c`** is dominated by the **`write!`**, but is dominated by another **`write!`** to **`c`**

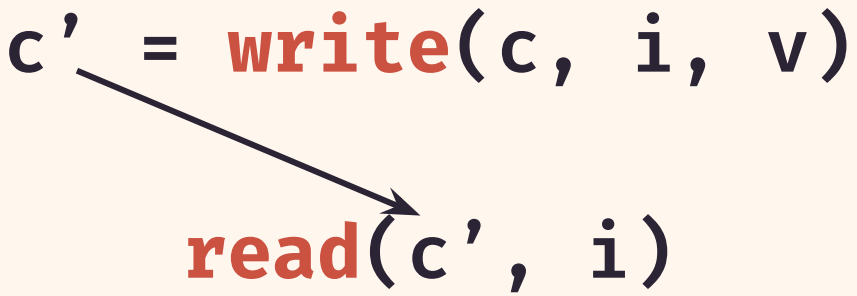
So, what about data collections?

```
read(c, i)
```

```
write!(c, i, v)
```

Nope! The use of **`c`** is not dominated by the **`write!`**

So, how do we fix this? SSA Construction.



So, what about MEMOIR?

Each operation on a collection produces a new collection

(except for `read`)

```
write!(c, i, v)    → c' = write(c, i, v)
```

So, what about MEMOIR?

Each operation on a collection produces a new collection

(except for `read`)

```
write!(c, i, v)    → c' = write(c, i, v)
swap!(c, i, j)     → c' = swap(c, i, j)
```

So, what about MEMOIR?

Each operation on a collection produces a new collection

(except for `read`)

```
write!(c, i, v)    → c' = write(c, i, v)
swap!(c, i, j)     → c' = swap(c, i, j)
remove!(c, i)      → c' = remove(c, i)
insert!(c, i, v)   → c' = insert(c, i, v)
```

So, what about MEMOIR?

Each operation on a collection produces a new collection

(except for `read`)

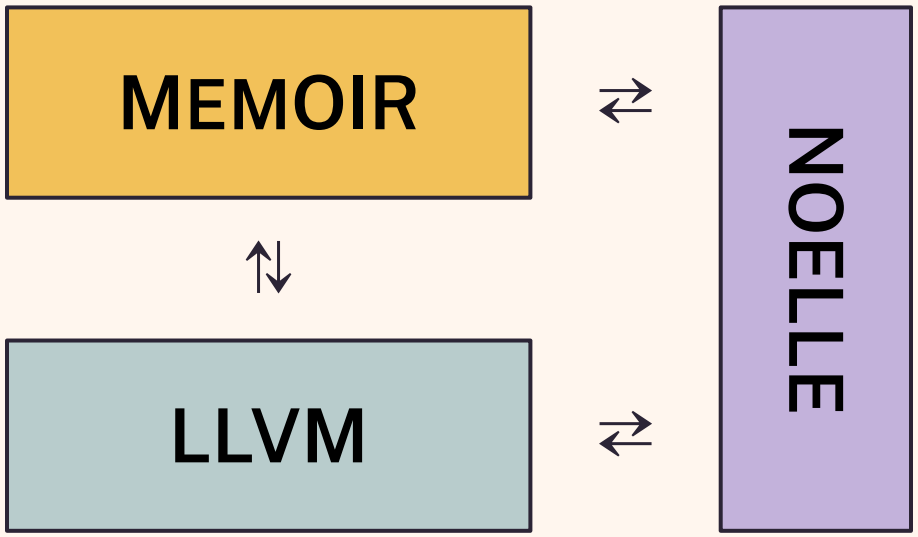
```
write!(c, i, v)     → c' = write(c, i, v)
swap!(c, i, j)      → c' = swap(c, i, j)
remove!(c, i)       → c' = remove(c, i)
insert!(c, i, v)    → c' = insert(c, i, v)
read(c, i)          → v  = read(c, i)
```

So, what about MEMOIR?

Other, useful query operations can be easily performed:

```
n = size(c)     ▷ # of elements in c
h = has(c, i)   ▷ does c have index i?
ks = keys(c)    ▷ sequence of keys in c
```

The MEMOIR Compiler



Step-by-step instructions are available

Writing a pass:

mcmichen.cc/memoir-docs/user/writing_a_pass

Writing a program:

mcmichen.cc/memoir-docs/user/writing_a_program

Additional Resources

Developer manual:

mcmichen.cc/memoir-docs

Doxygen:

mcmichen.cc/memoir-doxygen

The CGO'24 paper:

mcmichen.cc/files/MEMOIR_CGO_2024.pdf

Yippee!

Live Coding Time