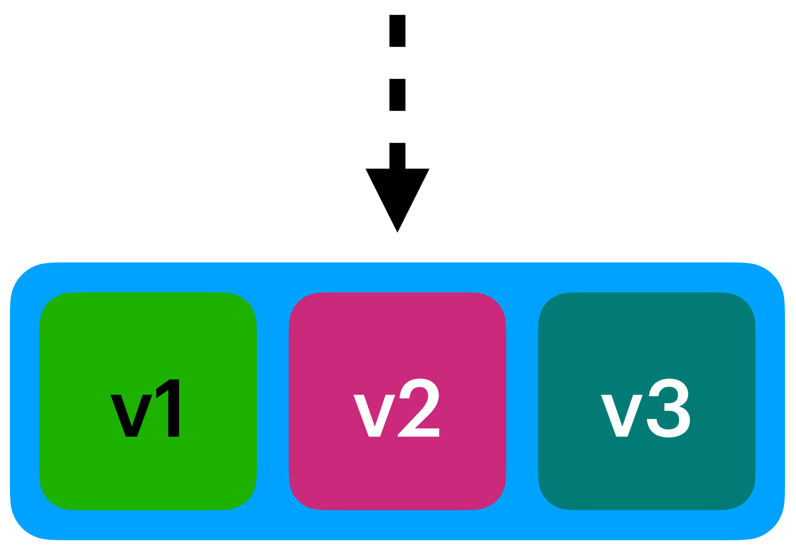
Mutable Cells for Memoized Lexing

Verified Memoization Framework for Verified Invertible Lexing

Samuel Chassot - 16.03.2026

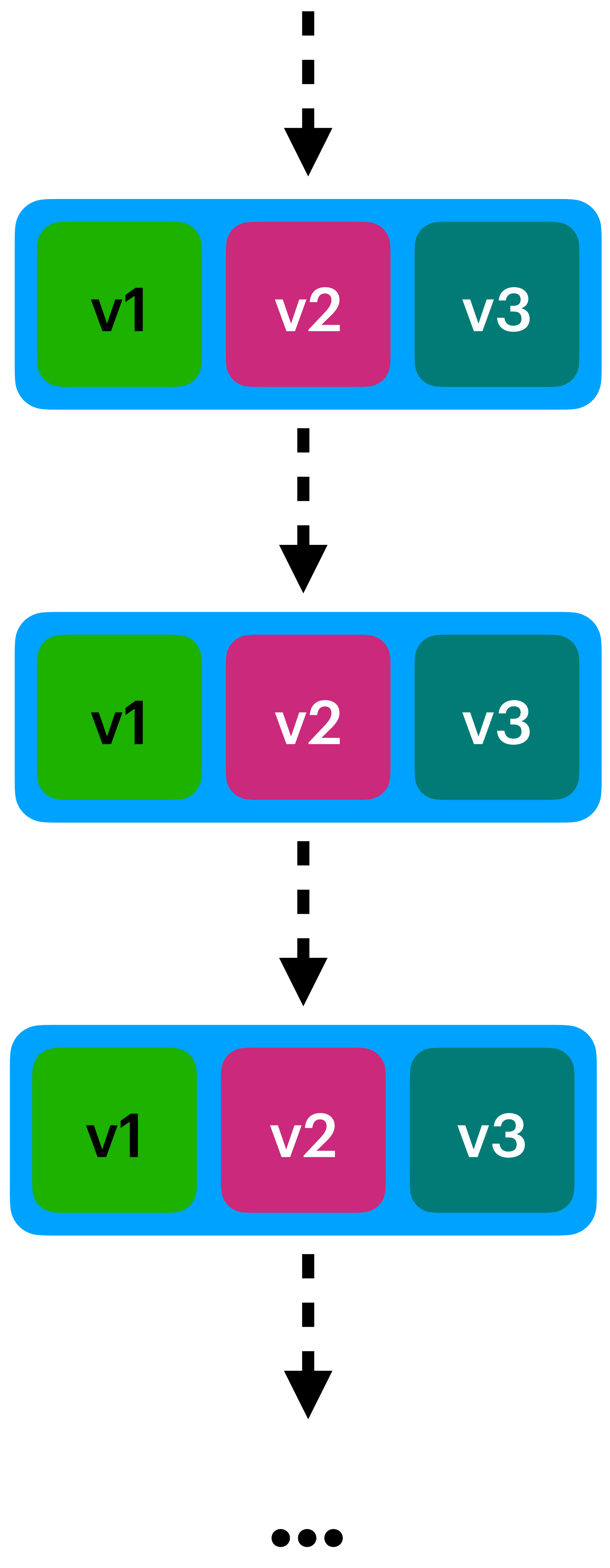
Introduction

Reasoning with Mutation and Aliasing



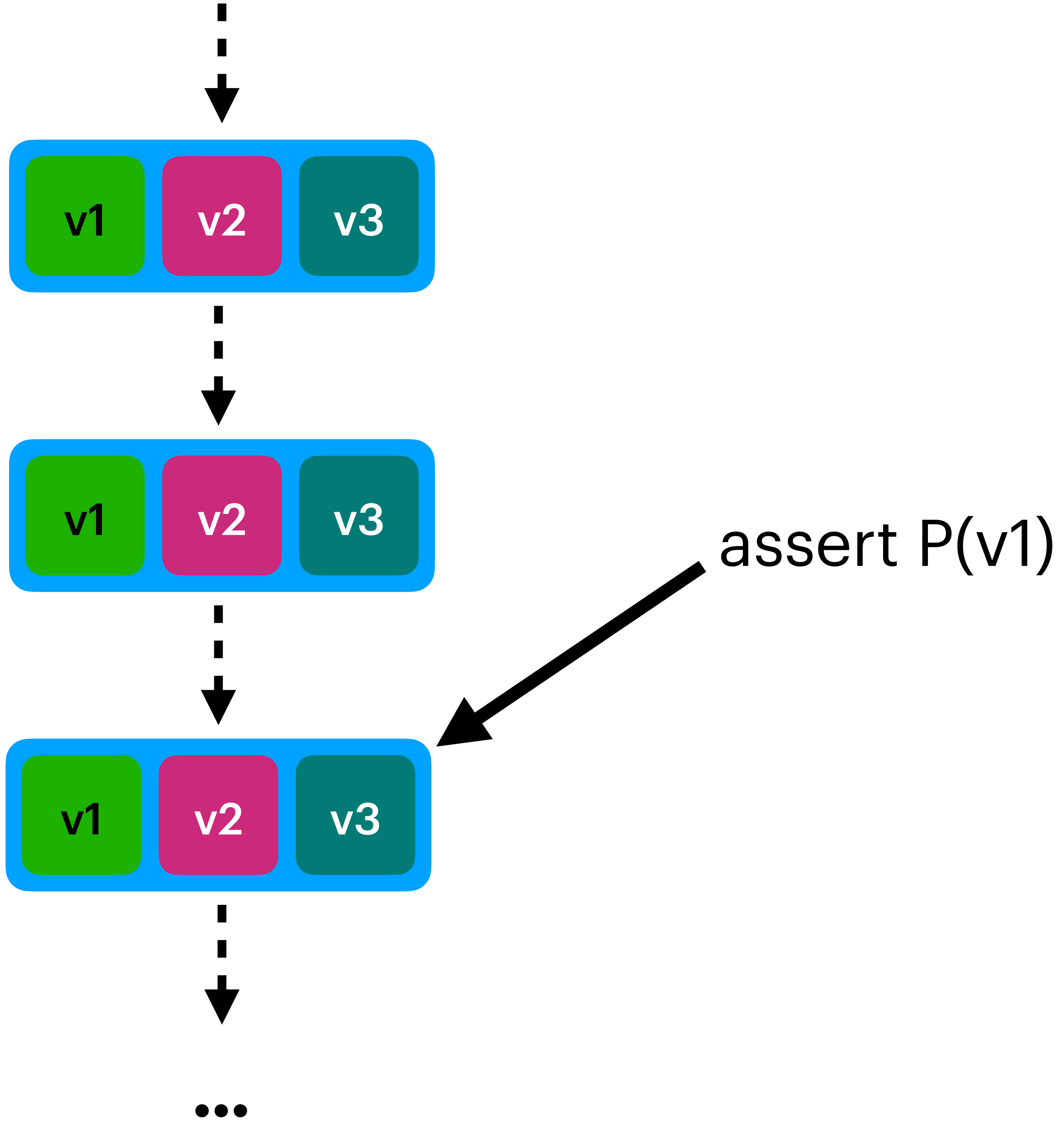
Introduction

Reasoning with Mutation and Aliasing



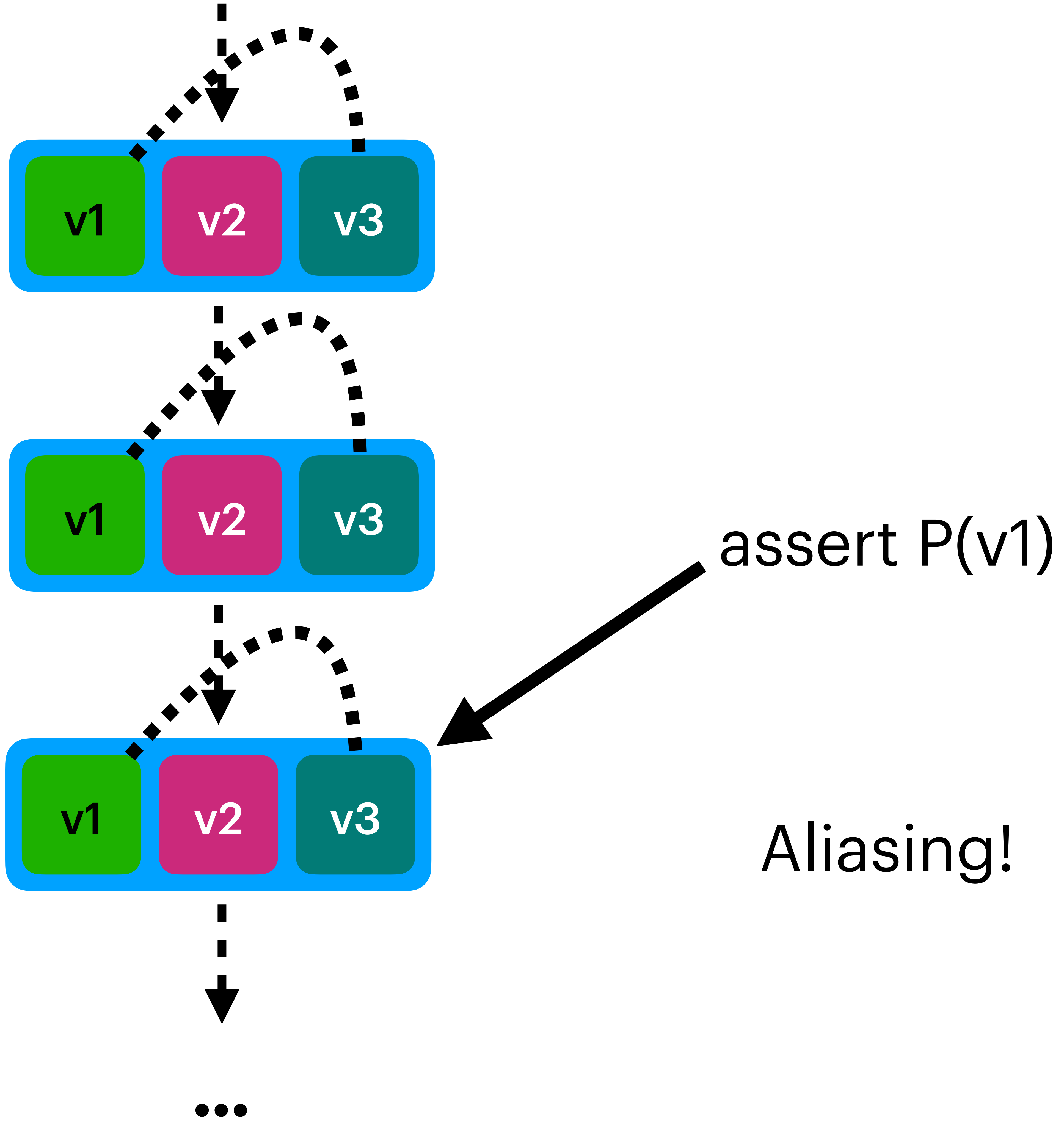
Introduction

Reasoning with Mutation and Aliasing



Introduction

Reasoning with Mutation and Aliasing



Aliasing!

Introduction

Mutation and Reasoning

Introduction

Mutation and Reasoning

Mutation **practical** to **efficiently** write **efficient** programs

Introduction

Mutation and Reasoning

Mutation **practical** to **efficiently** write **efficient** programs

Expensive to **analyze** (for tools and humans) because of **aliasing**

Introduction

Mutation and Reasoning

Mutation **practical** to **efficiently** write **efficient** programs

Expensive to **analyze** (for tools and humans) because of **aliasing**

Verification

Introduction

Mutation and Reasoning

Mutation **practical** to **efficiently** write **efficient** programs

Expensive to **analyze** (for tools and humans) because of **aliasing**

Verification

Dafny: heap encoding $\Rightarrow$ **expensive** & **brittle** reasoning

Introduction

Mutation and Reasoning

Mutation **practical** to **efficiently** write **efficient** programs

Expensive to **analyze** (for tools and humans) because of **aliasing**

Verification

- **Dafny**: heap encoding $\Rightarrow$ **expensive** & **brittle** reasoning
- What about **Stainless?**

Introduction

Mutation and Reasoning

Mutation **practical** to **efficiently** write **efficient** programs

Expensive to **analyze** (for tools and humans) because of **aliasing**

Verification

- **Dafny**: heap encoding ⇒ **expensive** & **brittle** reasoning
- What about **Stainless?**

Aliasing makes code analysis hard for tools and humans

Outline

Can we recover the **efficiency** benefits of **mutable memoization** to achieve verified and **efficient programs** using Stainless **without** reintroducing the verification burden of **aliasing**?

Outline

Can we recover the **efficiency** benefits of **mutable memoization** to achieve verified and **efficient programs** using Stainless **without** reintroducing the verification burden of **aliasing**?

- **Stainless** introduction

Outline

Can we recover the **efficiency** benefits of **mutable memoization** to achieve verified and **efficient programs** using Stainless **without** reintroducing the verification burden of **aliasing**?

- **Stainless** introduction
- **Memoization**

Outline

Can we recover the **efficiency** benefits of **mutable memoization** to achieve verified and **efficient programs** using Stainless **without** reintroducing the verification burden of **aliasing**?

- **Stainless** introduction
- **Memoization**
- Mutable **hash tables** with **mutable cells**

Outline

Can we recover the **efficiency** benefits of **mutable memoization** to achieve verified and **efficient programs** using Stainless **without** reintroducing the verification burden of **aliasing**?

- **Stainless** introduction
- **Memoization**
- Mutable **hash tables** with **mutable cells**
- Linear-time invertible **lexing** with memoization

Stainless Verifier

Stainless: Automated Proof

Verification Framework for Scala

```scala
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {

  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
           Cons((x, y), zip(xs0, ys0))

    case _ => Nil()
}
```

Stainless: Automated Proof

Verification Framework for Scala

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {

  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))

    case _ => Nil()
}.ensuring (res => res.map(p => p._1) == xs)
```

Stainless: Automated Proof

Verification Framework for Scala

```
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {

  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))

    case _ => Nil()
}.ensuring (res => res.map(p => p._1) == xs)
```

```
warning: Found counter-example:
warning:   xs: List[Int]          -> Cons[Int](0, Nil[Int]())
           ys: List[Boolean] -> Nil[Boolean]()
```

Stainless: Automated Proof

Verification Framework for Scala

```scala
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {
  require(xs.size <= ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))

    case _ => Nil()
}.ensuring (res => res.map(p => p._1) == xs)
```

Stainless: Automated Proof

Verification Framework for Scala

```scala
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {
  require(xs.size <= ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
          Cons((x, y), zip(xs0, ys0))

    case _ => Nil()
}.ensuring (res => res.map(p => p._1) == xs)
```

```
stainless summary

zip.scala:5:5:     zip  non-negative measure                                  valid  U:smt-cvc5  0.0
zip.scala:9:11:    zip  postcondition                                          valid  U:smt-cvc5  0.1
zip.scala:9:24:    zip  measure decreases                                      valid  U:smt-cvc5  0.0
zip.scala:9:24:    zip  precond. (call zip((scrut._1.t): @DropVCs , (scrut._...)  valid  U:smt-cvc5  0.1
zip.scala:11:15:   zip  postcondition                                          valid  U:smt-cvc5  0.0
-----------------------------------------------------------------------------------------------------
total: 5    valid: 5    (0 from cache, 0 trivial) invalid: 0    unknown: 0    time:    0.29
```

Stainless

Mutation and Aliasing Policies

Stainless

Mutation and Aliasing Policies

Unrestricted aliasing of **immutable** types

Stainless

Mutation and Aliasing Policies

Unrestricted aliasing of **immutable** types

No aliasing of **mutable** types

Stainless

Mutation and Aliasing Policies

Unrestricted aliasing of **immutable** types

No aliasing of **mutable** types

Transform programs to **purely functional**

Stainless

Mutation and Aliasing Policies

Unrestricted aliasing of **immutable** types

No aliasing of **mutable** types

Transform programs to **purely functional**

Unlike others e.g., Rust, transformation **only for reasoning**

Stainless

Mutation and Aliasing Policies

Unrestricted aliasing of **immutable** types

No aliasing of **mutable** types

Transform programs to **purely functional**

Unlike others e.g., Rust, transformation **only for reasoning**

  Runtime still relies on **garbage collection**

Stainless

Mutation and Aliasing Policies

Unrestricted aliasing of **immutable** types

No aliasing of **mutable** types

Transform programs to **purely functional**

Unlike others e.g., Rust, transformation **only for reasoning**

Runtime still relies on **garbage collection**

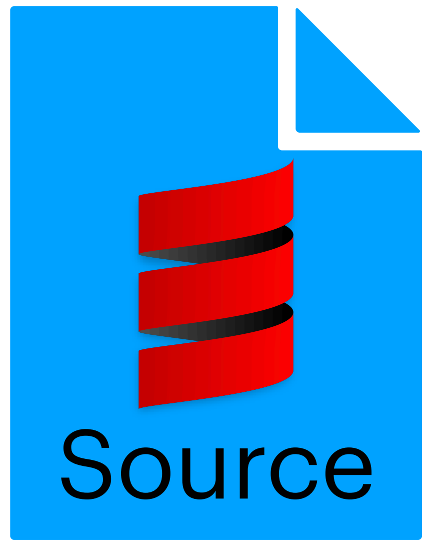
Mutation without aliasing reduces to purely functional
but requires checking its absence

Stainless

Verification framework for Scala

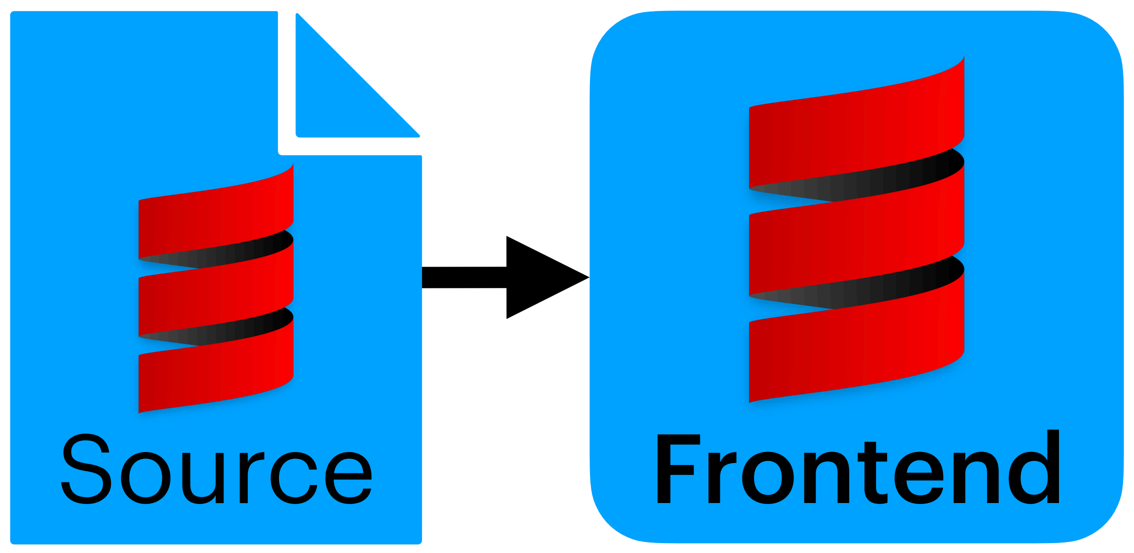
Stainless

Verification framework for Scala



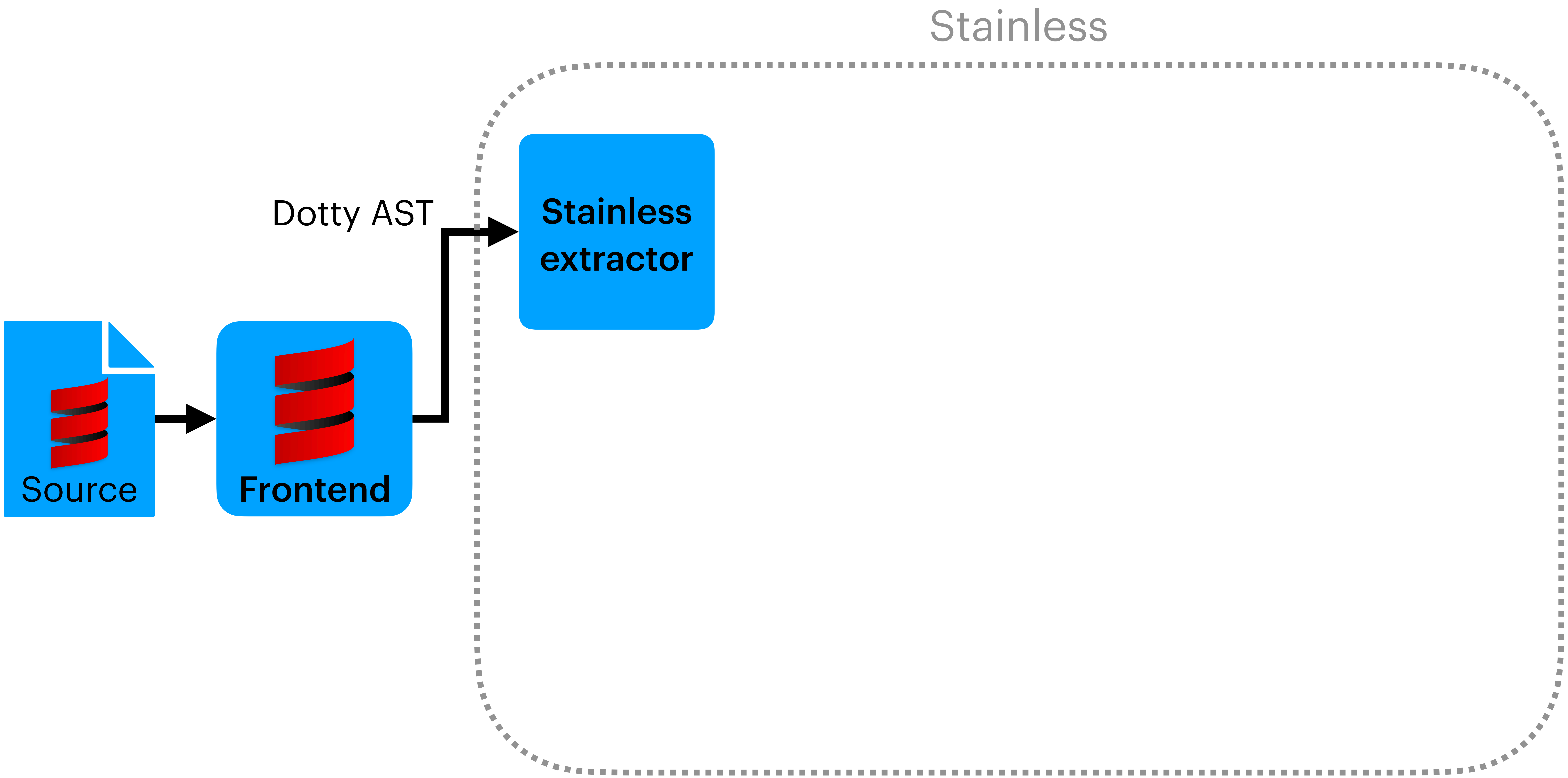
Stainless

Verification framework for Scala



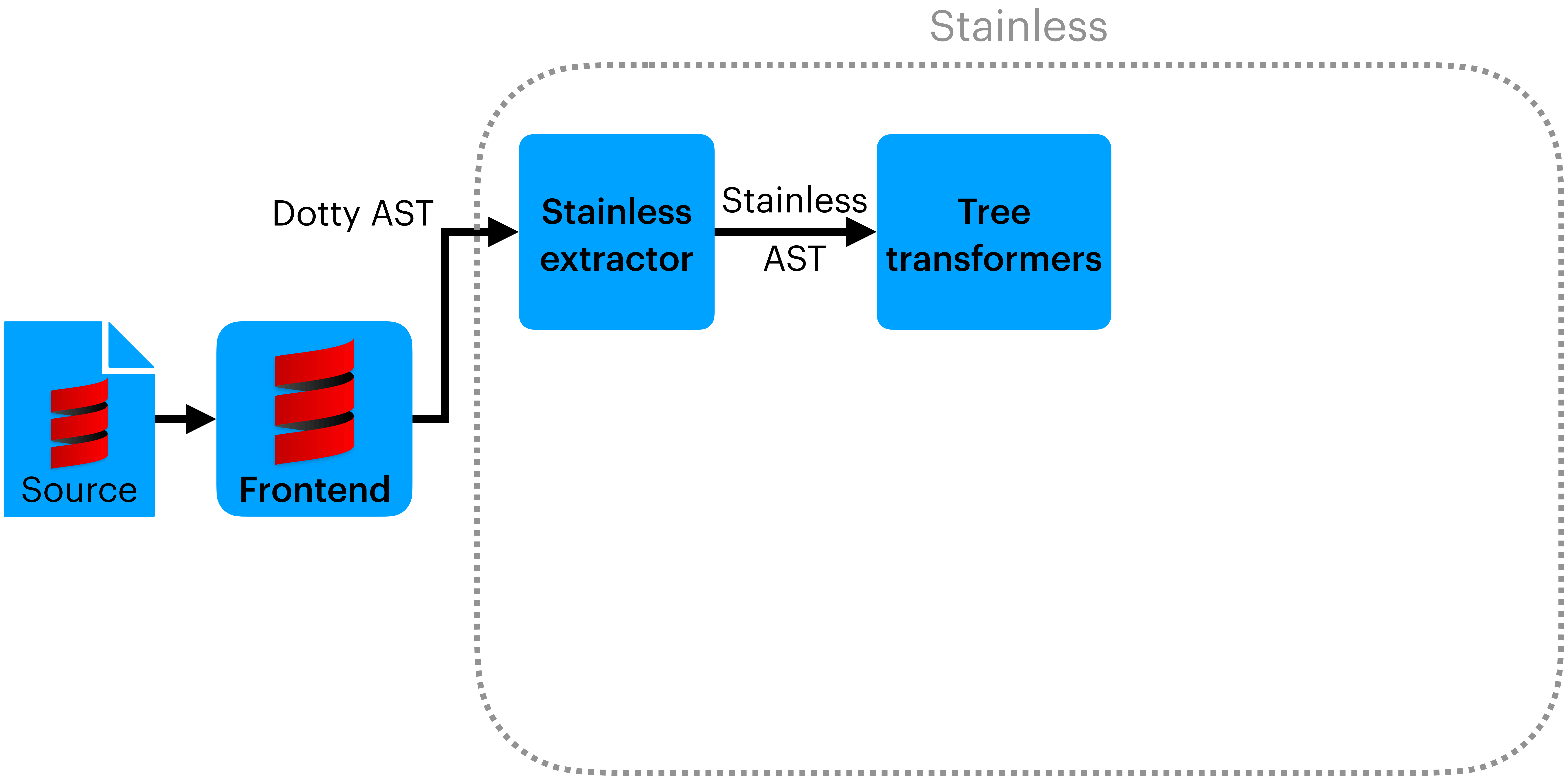
Stainless

Verification framework for Scala



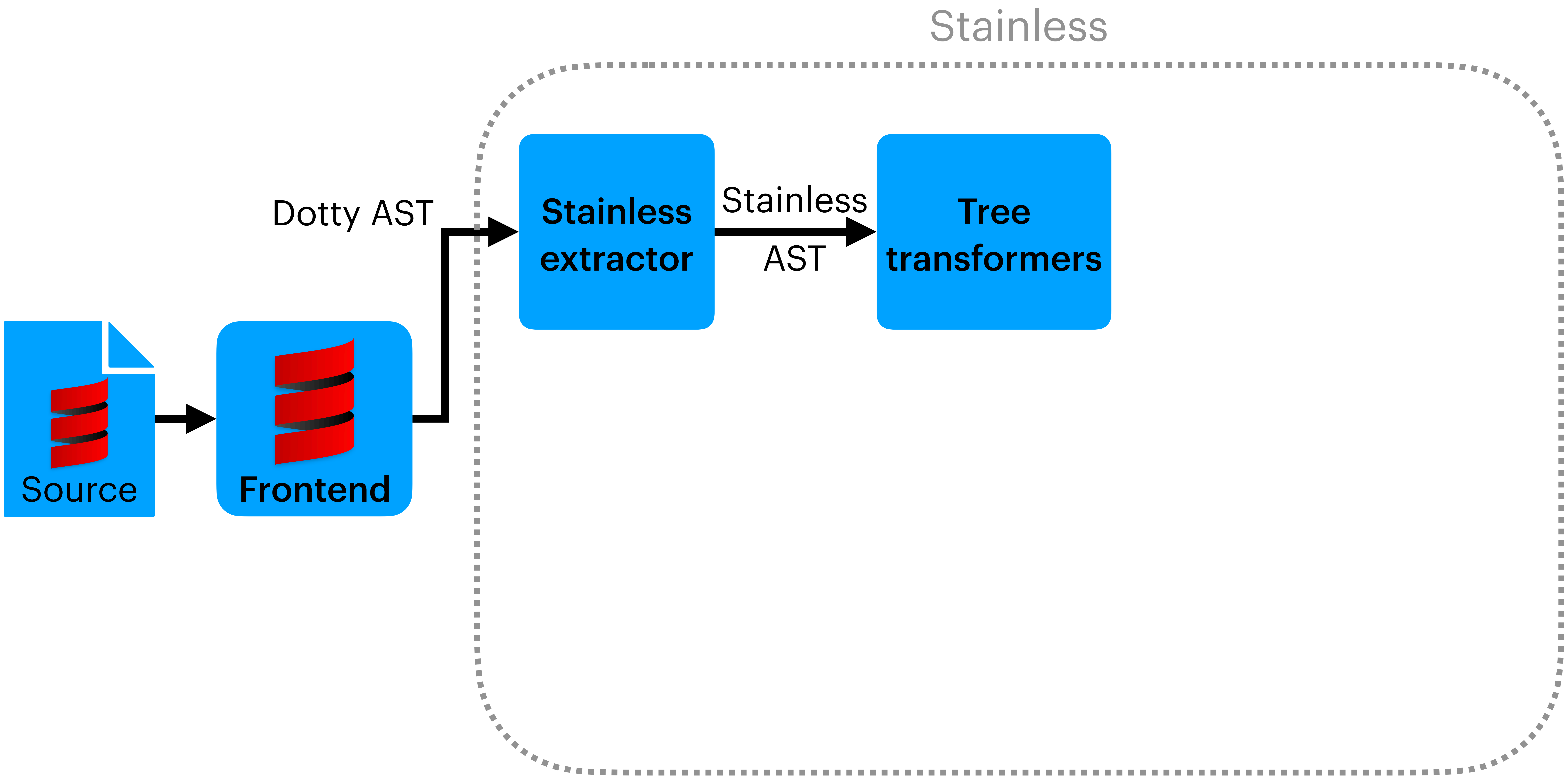
Stainless

Verification framework for Scala



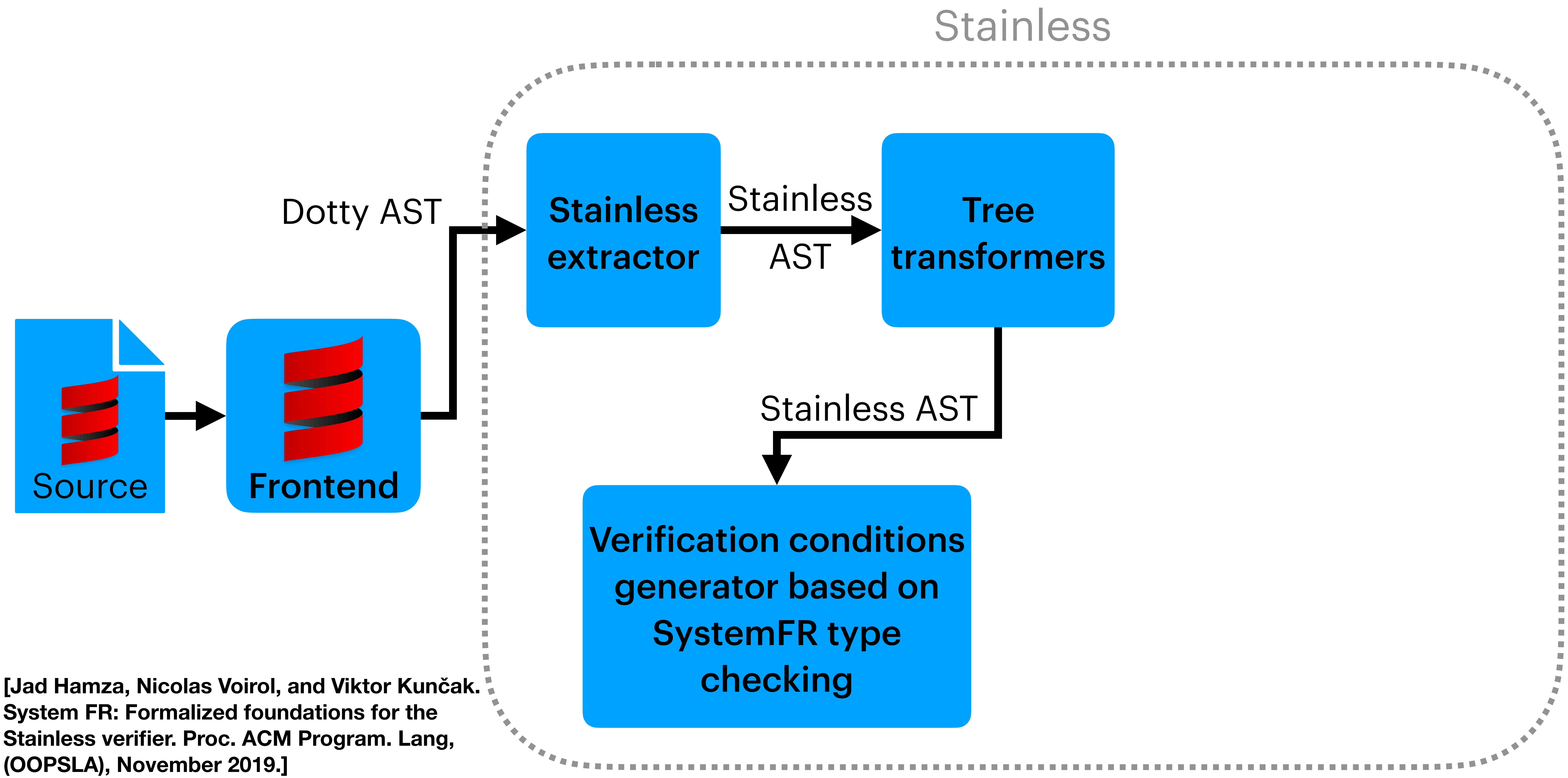
Stainless

Verification framework for Scala



Stainless

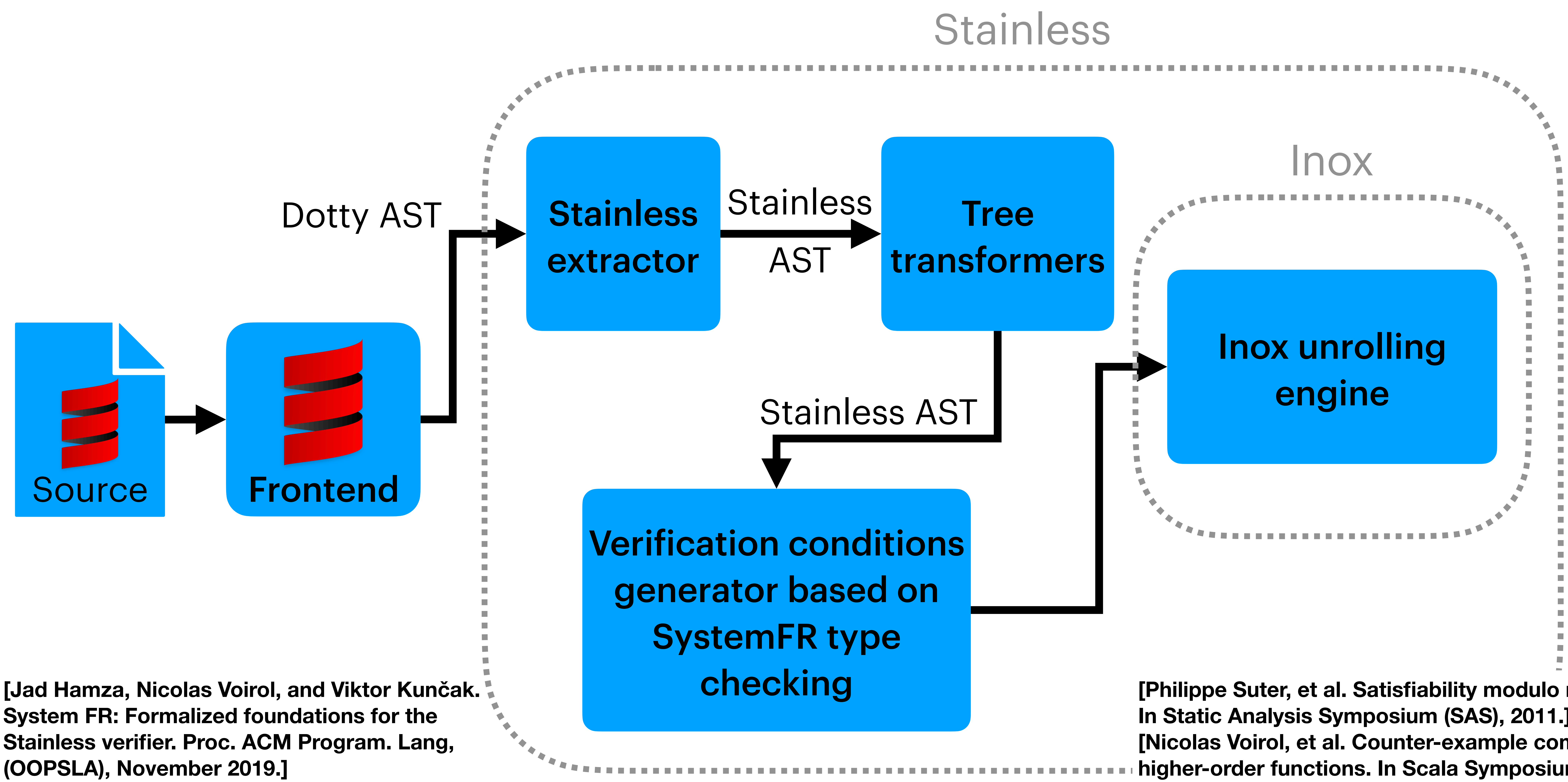
Verification framework for Scala



[Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang, (OOPSLA), November 2019.]

Stainless

Verification framework for Scala



[Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang, (OOPSLA), November 2019.]

[Philippe Suter, et al. Satisfiability modulo recursive programs. In Static Analysis Symposium (SAS), 2011.]
[Nicolas Voirol, et al. Counter-example complete verification for higher-order functions. In Scala Symposium, 2015.]

Stainless

Verification framework for Scala

[Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang, (OOPSLA), November 2019.]

[Philippe Suter, et al. Satisfiability modulo recursive programs. In Static Analysis Symposium (SAS), 2011.]
[Nicolas Voirol, et al. Counter-example complete verification for higher-order functions. In Scala Symposium, 2015.]

Stainless

Verification framework for Scala

[Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang, (OOPSLA), November 2019.]

[Philippe Suter, et al. Satisfiability modulo recursive programs. In Static Analysis Symposium (SAS), 2011.]
[Nicolas Voirol, et al. Counter-example complete verification for higher-order functions. In Scala Symposium, 2015.]

Stainless

Verification framework for Scala

[Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang, (OOPSLA), November 2019.]

[Philippe Suter, et al. Satisfiability modulo recursive programs. In Static Analysis Symposium (SAS), 2011.]
[Nicolas Voirol, et al. Counter-example complete verification for higher-order functions. In Scala Symposium, 2015.]

Stainless

Imperative Elimination Phase

```scala
case class MutA(var x: BigInt)

def incr(a: MutA): Unit =
  a.x = a.x + 1

def main(): Unit =
  val a: MutA = MutA(0)
  incr(a)
  assert(a.x == 1)
```

Stainless

Imperative Elimination Phase

```scala
case class MutA(var x: BigInt)

def incr(a: MutA): Unit =
  a.x = a.x + 1

def main(): Unit =
  val a: MutA = MutA(0)
  incr(a)
  assert(a.x == 1)
```

➡

Imperative Elimination

Stainless

Imperative Elimination Phase

```scala
case class MutA(var x: BigInt)

def incr(a: MutA): Unit =
  a.x = a.x + 1

def main(): Unit =
  val a: MutA = MutA(0)
  incr(a)
  assert(a.x == 1)
```

Imperative Elimination

```scala
def incr(a: MutA): (Unit, MutA) =
  val a_1: MutA = a
  val a_2: MutA = MutA(a_1.x + 1)
  ((), a_2)

def main(): Unit =
  val a: MutA = MutA(0)
  val res: (Unit, MutA) = incr(a)
  val a_1: MutA = res._2
  val a_2: MutA = a_1
  assert(a_2.x == 1)
  ()
```

Stainless

Imperative Elimination Phase

```scala
case class MutA(var x: BigInt)

def incr(a: MutA): Unit =
  a.x = a.x + 1

def main(): Unit =
  val a: MutA = MutA(0)
  incr(a)
  assert(a.x == 1)
```

Imperative Elimination

```scala
def incr(a: MutA): (Unit, MutA) =
  val a_1: MutA = a
  val a_2: MutA = MutA(a_1.x + 1)
  ((), a_2)

def main(): Unit =
  val a: MutA = MutA(0)
  val res: (Unit, MutA) = incr(a)
  val a_1: MutA = res._2
  val a_2: MutA = a_1
  assert(a_2.x == 1)
  ()
```

Stainless

Imperative Elimination Phase

```scala
case class MutA(var x: BigInt)

def incr(a: MutA): Unit =
  a.x = a.x + 1

def main(): Unit =
  val a: MutA = MutA(0)
  incr(a)
  assert(a.x == 1)
```

Imperative Elimination

```scala
def incr(a: MutA): (Unit, MutA) =
  val a_1: MutA = a
  val a_2: MutA = MutA(a_1.x + 1)
  ((), a_2)

def main(): Unit =
  val a: MutA = MutA(0)
  val res: (Unit, MutA) = incr(a)
  val a_1: MutA = res._2
  val a_2: MutA = a_1
  assert(a_2.x == 1)
  ()
```

Verified and Efficient programs

Verified and Efficient

Memoization

Verified and Efficient

Memoization

Correctness is crucial

Verified and Efficient

Memoization

Correctness is crucial

But **efficiency** is also central for adoption

Verified and Efficient

Memoization

Correctness is crucial

But **efficiency** is also central for adoption

$\Rightarrow$ **Memoization**

Verified and Efficient

Memoization

Correctness is crucial

But **efficiency** is also central for adoption

$\Rightarrow$ **Memoization**

We want efficient, verified, and reusable memoization

Memoization

General Framework

Memoization

General Framework

$$f : A \to B$$

Memoization

General Framework

$$f : A \to B$$

$$cache : A \to option[B]$$

Memoization

General Framework

$$f : A \to B \qquad\qquad cache : A \to option[B]$$

$$valid(cache)$$

Memoization

General Framework

$$f : A \to B \qquad cache : A \to option[B]$$

$$valid(cache) = \forall a : A$$

Memoization

General Framework

$$f : A \to B \qquad\qquad cache : A \to option[B]$$

$$valid(cache) = \forall a : A \;\; cache(a) = some(b) \implies$$

Memoization

General Framework

$$f : A \to B \qquad\qquad cache : A \to option[B]$$

$$valid(cache) = \forall a : A \;\; cache(a) = some(b) \implies b = f(a)$$

Memoization

General Framework

$$f : A \to B \qquad\qquad cache : A \to option[B]$$

$$valid(cache) = \forall a : A \;\; cache(a) = some(b) \implies b = f(a)$$

Efficient implementation?

Memoization

General Framework

$$f : A \to B \qquad\qquad cache : A \to option[B]$$

$$valid(cache) = \forall a : A \;\; cache(a) = some(b) \implies b = f(a)$$

Efficient implementation? $\Rightarrow$ **Hash Table**

Memoization

General Framework

$$f : A \to B \qquad\qquad cache : A \to option[B]$$

$$valid(cache) = \forall a : A \;\; cache(a) = some(b) \implies b = f(a)$$

Efficient implementation? $\Rightarrow$ **Hash Table**

Need mutability?

Memoization

Recursion and Immutable Cache

```
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

Memoization

Recursion and Immutable Cache

```
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```
type ImmutableCache = Map[BigInt, BigInt]
```

Memoization

Recursion and Immutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type ImmutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt, cache: ImmutableCache): (BigInt, ImmutableCache) =
  cache.get(n) match
    case Some(value) => (value, cache)
    case None =>
      if n <= 1 then (n, cache + (n -> n))
      else
        val (fibN1, cache1) = memFib(n - 1, cache)
        val (fibN2, cache2) = memFib(n - 2, cache1)
        val result = fibN1 + fibN2
        (result, cache2 + (n -> result))
```

Memoization

Recursion and Immutable Cache

```
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```
type ImmutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt, cache: ImmutableCache)  (BigInt, ImmutableCache) =
  cache.get(n) match
    case Some(value) => (value, cache)
    case None =>
      if n <= 1 then (n, cache + (n -> n))
      else
        val (fibN1, cache1) = memFib(n - 1, cache)
        val (fibN2, cache2) = memFib(n - 2, cache1)
        val result = fibN1 + fibN2
        (result, cache2 + (n -> result))
```

Memoization

Recursion and Immutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)

type ImmutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt, cache: ImmutableCache): (BigInt, ImmutableCache) =
  cache.get(n) match
    case Some(value) => (value, cache)
    case None =>
      if n <= 1 then (n, cache + (n -> n))
      else
        val (fibN1, cache1) = memFib(n - 1, cache)
        val (fibN2, cache2) = memFib(n - 2, cache1)
        val result = fibN1 + fibN2
        (result, cache2 + (n -> result))
```

Memoization

Recursion and Immutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type ImmutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt, cache: ImmutableCache): (BigInt, ImmutableCache) =
  cache.get(n) match
    case Some(value) => (value, cache)
    case None =>
      if n <= 1 then (n, cache + (n -> n))
      else
        val (fibN1, cache1) = memFib(n - 1, cache)
        val (fibN2, cache2) = memFib(n - 2, cache1)
        val result = fibN1 + fibN2
        (result, cache2 + (n -> result))
```

Memoization

Recursion and Immutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type ImmutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt, cache: ImmutableCache): (BigInt, ImmutableCache) =
  cache.get(n) match
    case Some(value) => (value, cache)
    case None =>
      if n <= 1 then (n, cache + (n -> n))
      else
        val (fibN1, cache1) = memFib(n - 1, cache)
        val (fibN2, cache2) = memFib(n - 2, cache1)
        val result = fibN1 + fibN2
        (result, cache2 + (n -> result))
```

Memoization

Recursion and Immutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type ImmutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt, cache: ImmutableCache): (BigInt, ImmutableCache) =
  cache.get(n) match
    case Some(value) => (value, cache)
    case None =>
      if n <= 1 then (n, cache + (n -> n))
      else
        val (fibN1, cache1) = memFib(n - 1, cache)
        val (fibN2, cache2) = memFib(n - 2, cache1)
        val result = fibN1 + fibN2
        (result, cache2 + (n -> result))
```

Memoization

Recursion and Immutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type ImmutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt, cache: ImmutableCache): (BigInt, ImmutableCache) =
  cache.get(n) match
    case Some(value) => (value, cache)
    case None =>
      if n <= 1 then (n, cache + (n -> n))
      else
        val (fibN1, cache1) = memFib(n - 1, cache)
        val (fibN2, cache2) = memFib(n - 2, cache1)
        val result = fibN1 + fibN2
        (result, cache2 + (n -> result))
```

Memoization

Recursion and Immutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type ImmutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt, cache: ImmutableCache): (BigInt, ImmutableCache) =
  cache.get(n) match
    case Some(value) => (value, cache)
    case None =>
      if n <= 1 then (n, cache + (n -> n))
      else
        val (fibN1, cache1) = memFib(n - 1, cache)
        val (fibN2, cache2) = memFib(n - 2, cache1)
        val result = fibN1 + fibN2
        (result, cache2 + (n -> result))
```

Requires rewriting the program with this monadic construct

Memoization

Recursion and Mutable Cache

```
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

Memoization

Recursion and Mutable Cache

```
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```
type MutableCache = Map[BigInt, BigInt]
```

Memoization

Recursion and Mutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type MutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt)(using cache: MutableCache): BigInt =
  cache.get(n) match
    case Some(value) => value
    case None =>
      val result =
        if n <= 1 then n
        else memFib(n - 1) + memFib(n - 2)
      cache.update(n, result)
      result
```

Memoization

Recursion and Mutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type MutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt)(using cache: MutableCache): BigInt =
  cache.get(n) match
    case Some(value) => value
    case None =>
      val result =
        if n <= 1 then n
        else memFib(n - 1) + memFib(n - 2)
      cache.update(n, result)
      result
```

Memoization

Recursion and Mutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type MutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt)(using cache: MutableCache): BigInt =
  cache.get(n) match
    case Some(value) => value
    case None =>
      val result =
        if n <= 1 then n
        else memFib(n - 1) + memFib(n - 2)
      cache.update(n, result)
      result
```

Memoization

Recursion and Mutable Cache

```scala
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```scala
type MutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt)(using cache: MutableCache): BigInt =
  cache.get(n) match
    case Some(value) => value
    case None =>
      val result =
        if n <= 1 then n
        else memFib(n - 1) + memFib(n - 2)
      cache.update(n, result)
      result
```

Memoization

Recursion and Mutable Cache

```
def fib(n: BigInt): BigInt =
    if n <= 1 then n
    else fib(n - 1) + fib(n - 2)
```

```
type MutableCache = Map[BigInt, BigInt]

def memFib(n: BigInt)(using cache: MutableCache): BigInt =
  cache.get(n) match
    case Some(value) => value
    case None =>
      val result =
        if n <= 1 then n
        else memFib(n - 1) + memFib(n - 2)
      cache.update(n, result)
      result
```

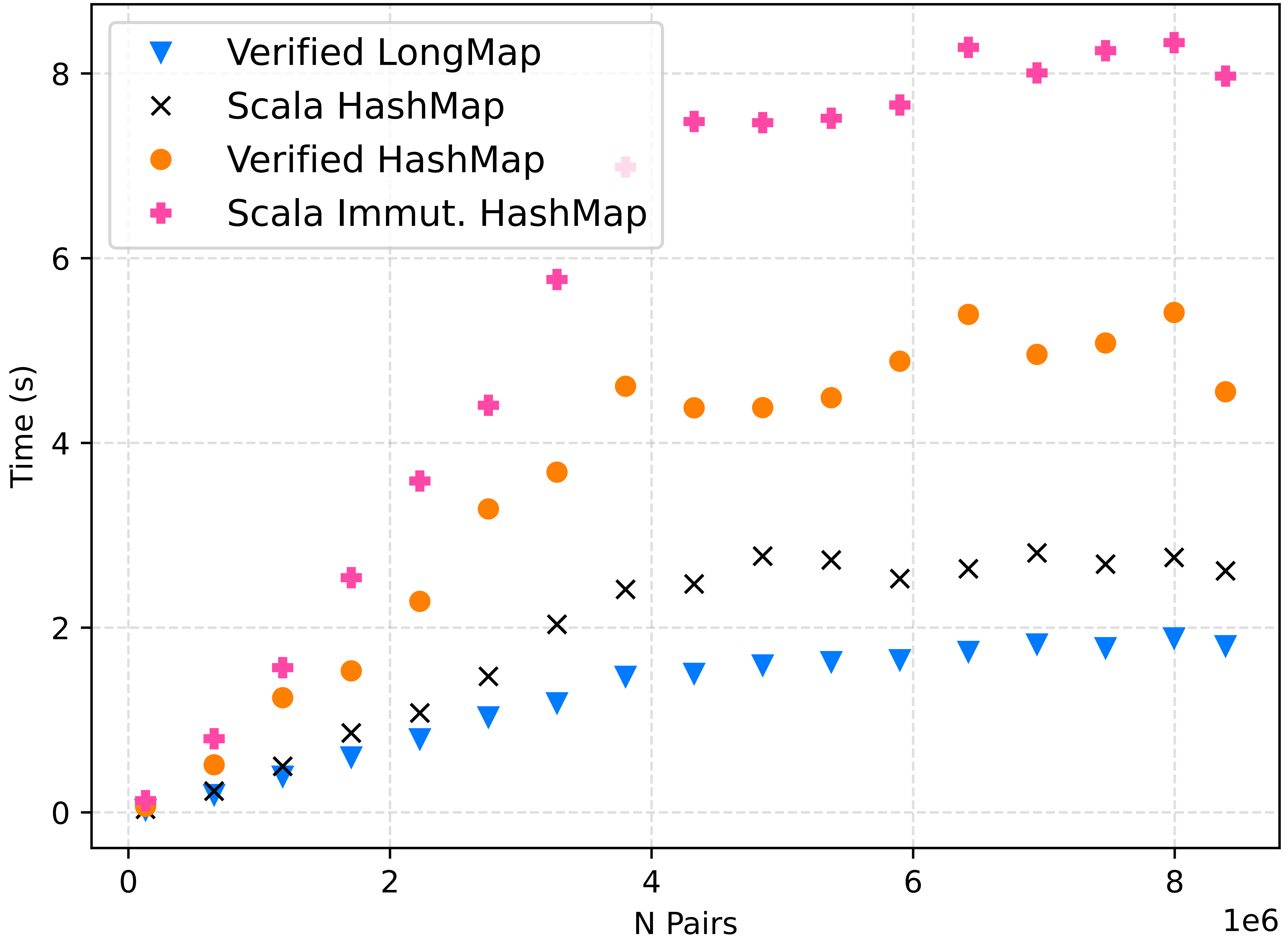
Only one more (implicit) parameter, same specification

Maps Performance

Why not `var map = new immutable.Map`?

Maps Performance

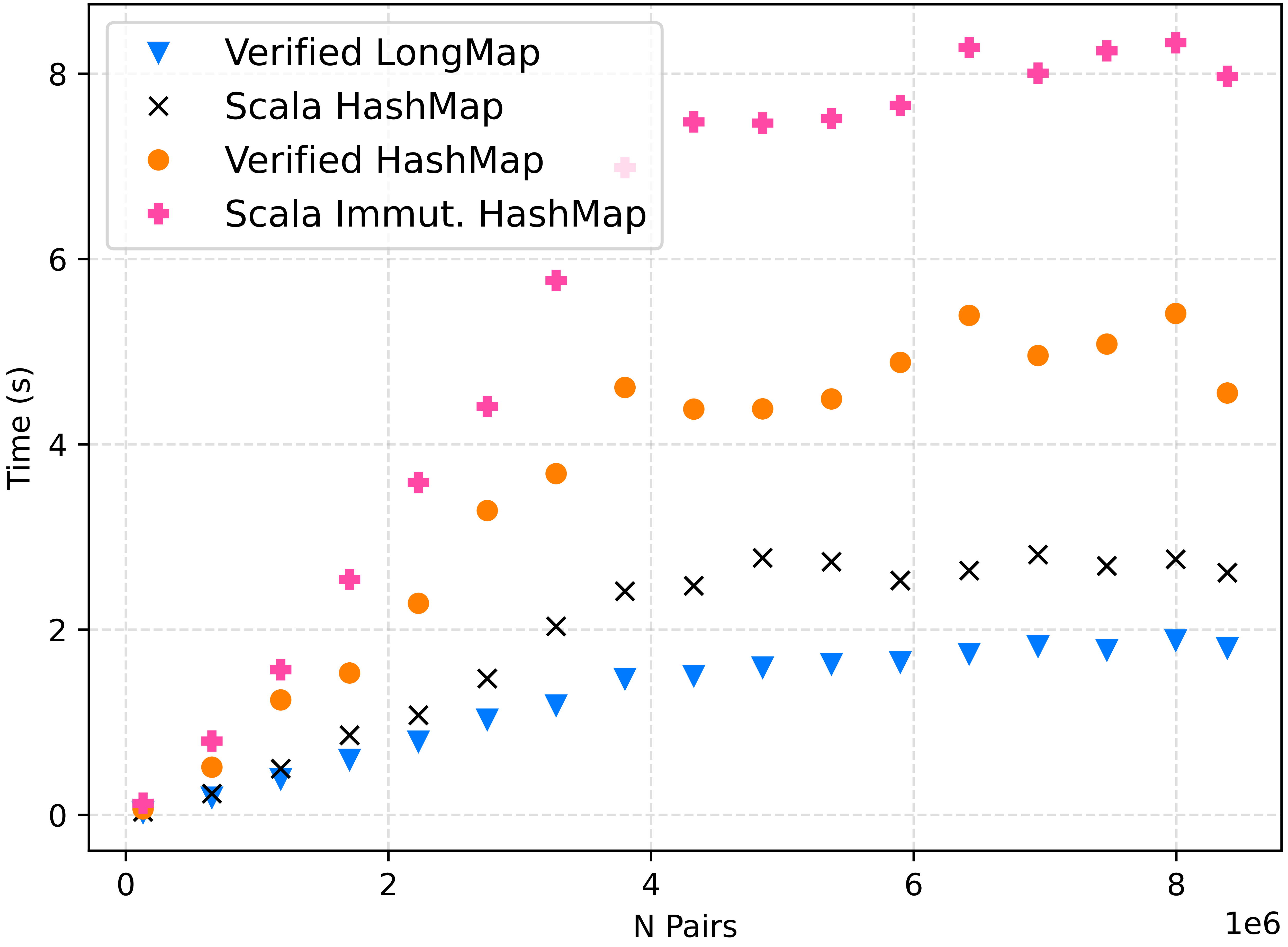
Why not var map = new immutable.Map?



Insert N pairs, remove N/2, insert N

Maps Performance

Why not var map = new immutable.Map?



⇒ Our **verified mutable** hash table outperforms the Scala library **unverified immutable** map

Insert N pairs, remove N/2, insert N

Verified Mutable Hash Table without Aliasing using Mutable Cells

LongMap[1]

64-bit Keys Hash Table

[1]**<u>Chassot, S.</u>**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

LongMap[1]

64-bit Keys Hash Table

- **Mutable** Hash Table

[1]**Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

LongMap[1]

64-bit Keys Hash Table

- **Mutable** Hash Table
- **Open addressing**, non-linear probing

[1]**Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

LongMap[1]

64-bit Keys Hash Table

- **Mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

LongMap[1]

64-bit Keys Hash Table

- **Mutable** Hash Table
- **Open addressing**, non-linear probing
- From Scala standard library

```
trait LongMap[V]:
  def contains(key: Long): Boolean
  def apply(key: Long): V           // Lookup
  def update(key: Long, v: V): Boolean
  def remove(key: Long): Boolean
  def repack(): Boolean
```

[1] **Chassot, S.**, Kunčak, V. (2024). **Verifying a Realistic Mutable Hash Table.** In: IJCAR 2024. Lecture Notes in Computer Science, vol 14739. Springer

LongMap

Decorator Pattern

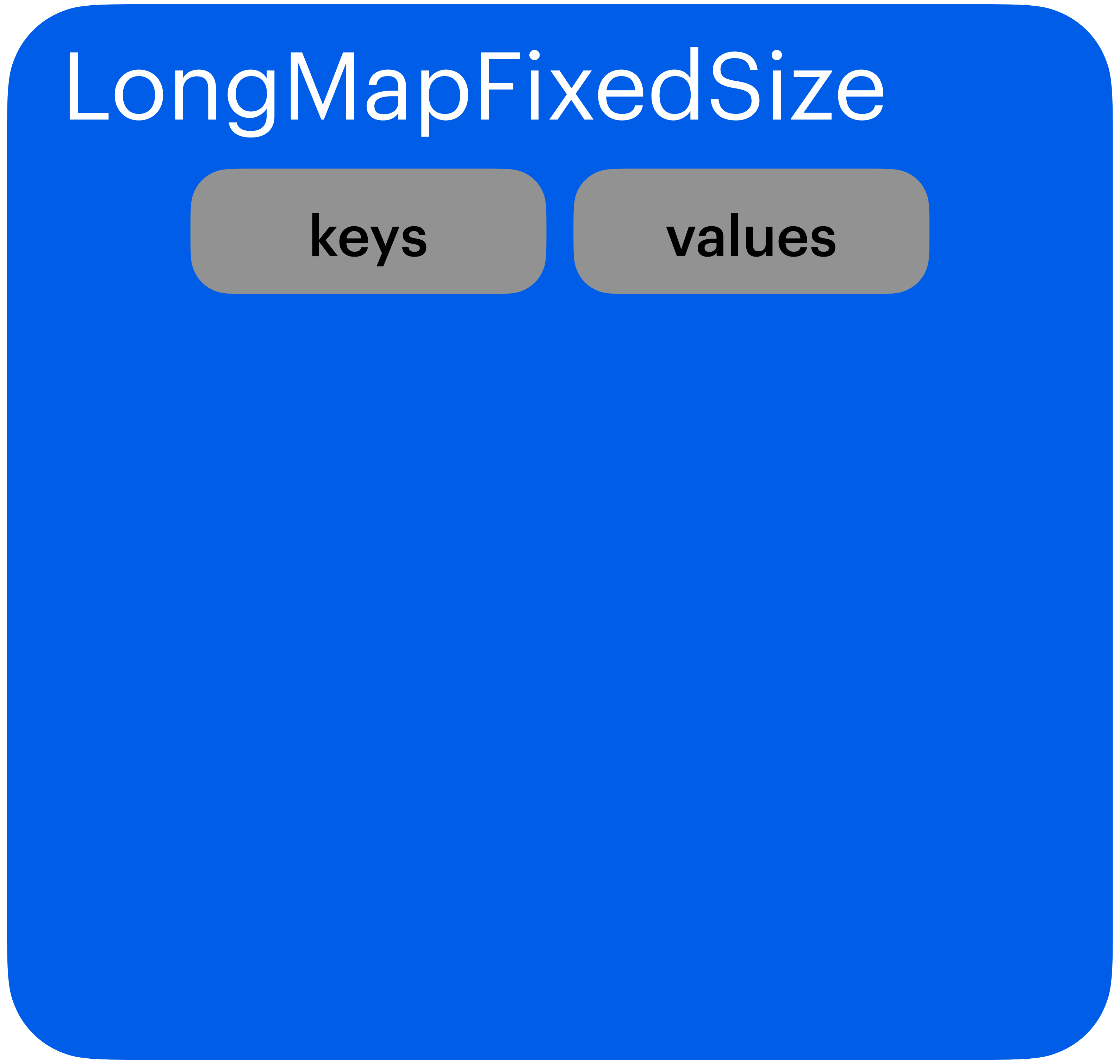
LongMap

Decorator Pattern

LongMapFixedSize

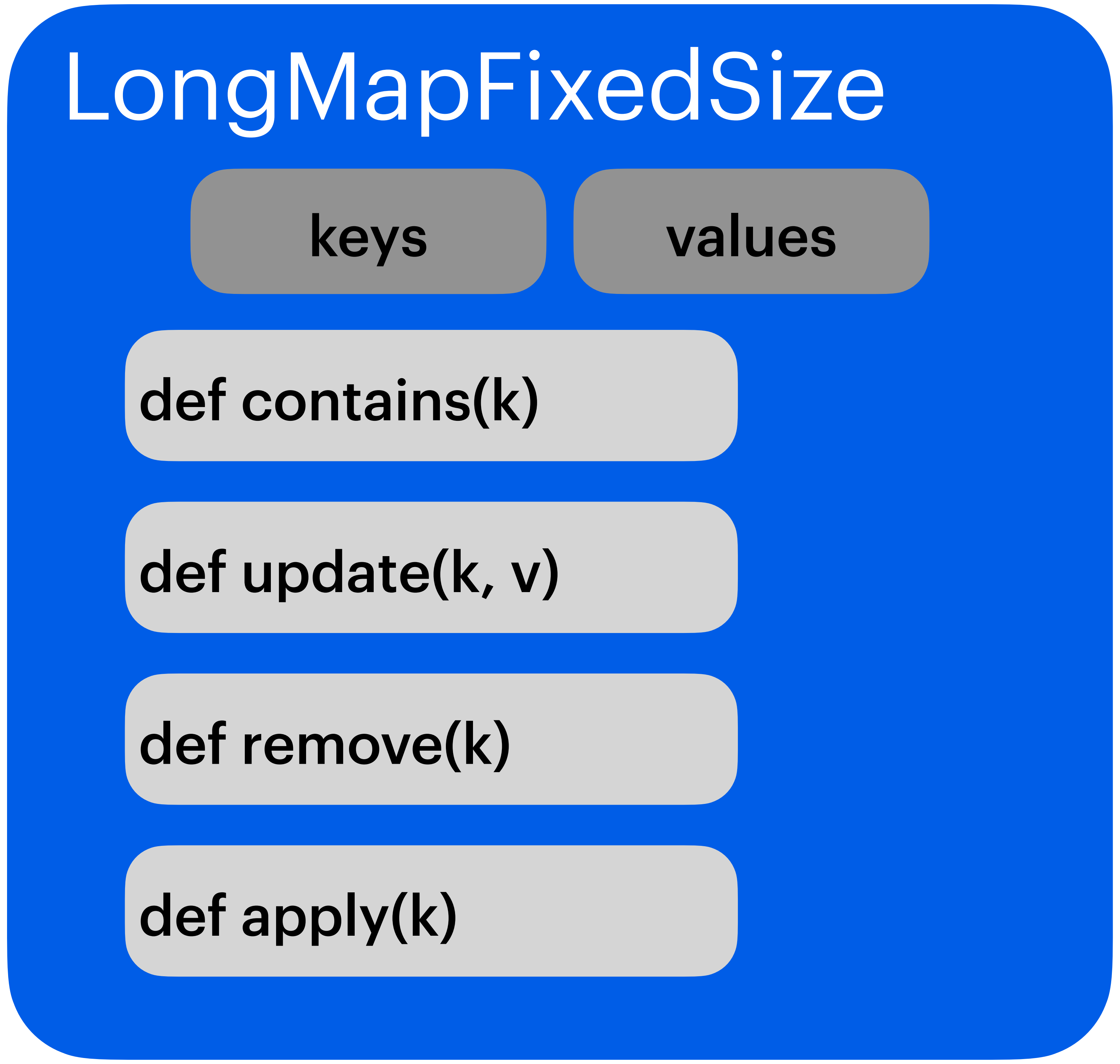
LongMap

Decorator Pattern



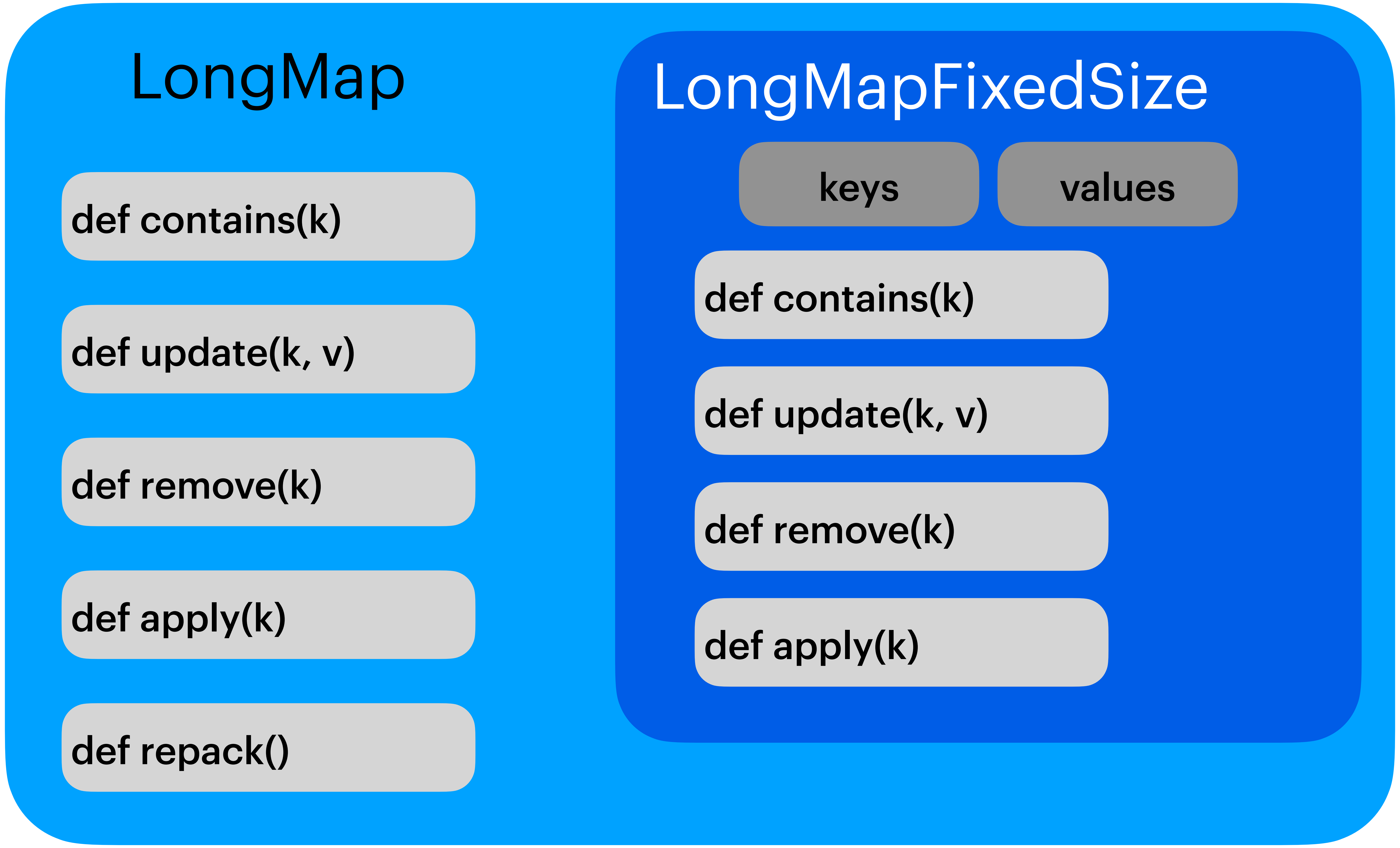
LongMap

Decorator Pattern



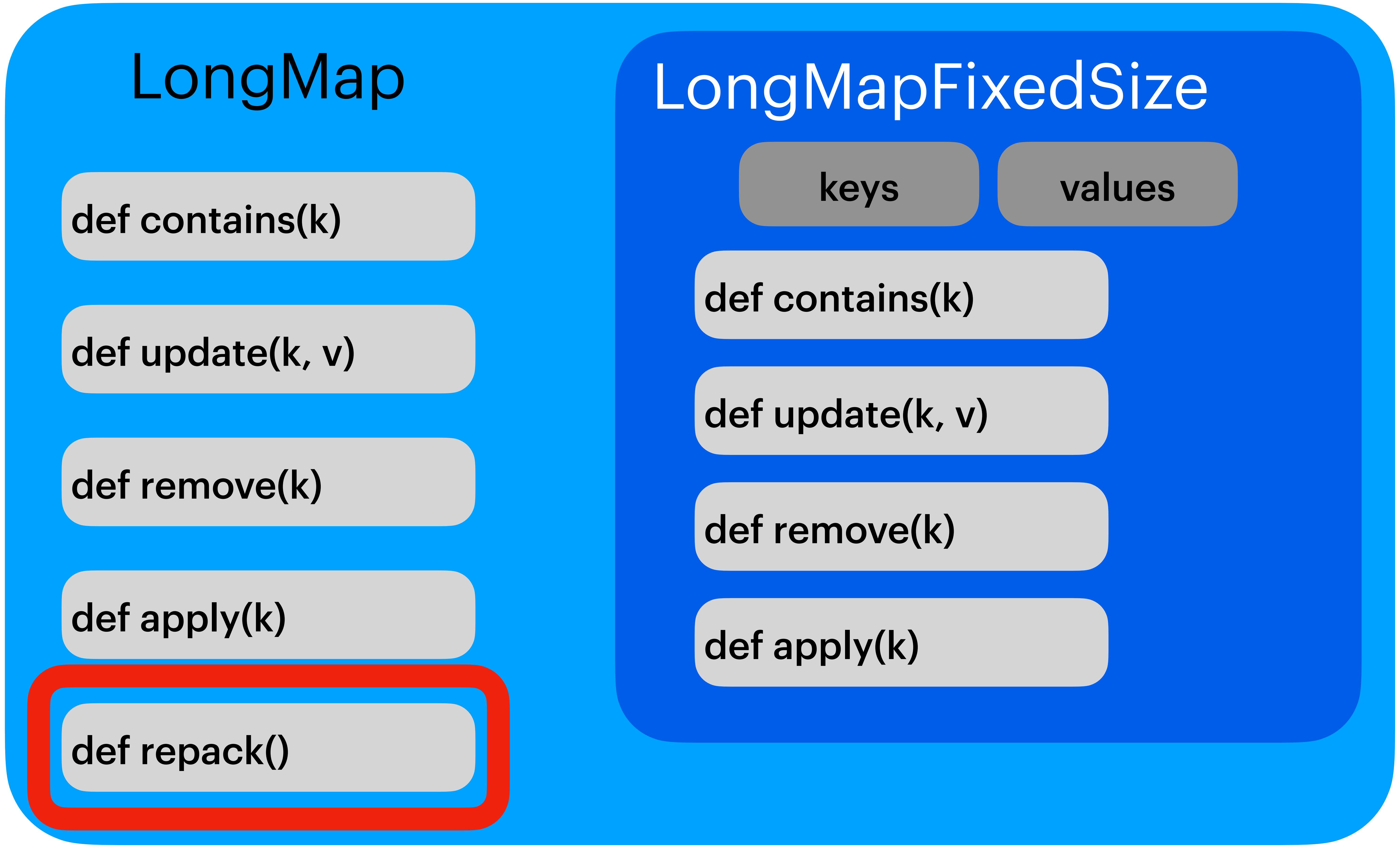
LongMap

Decorator Pattern



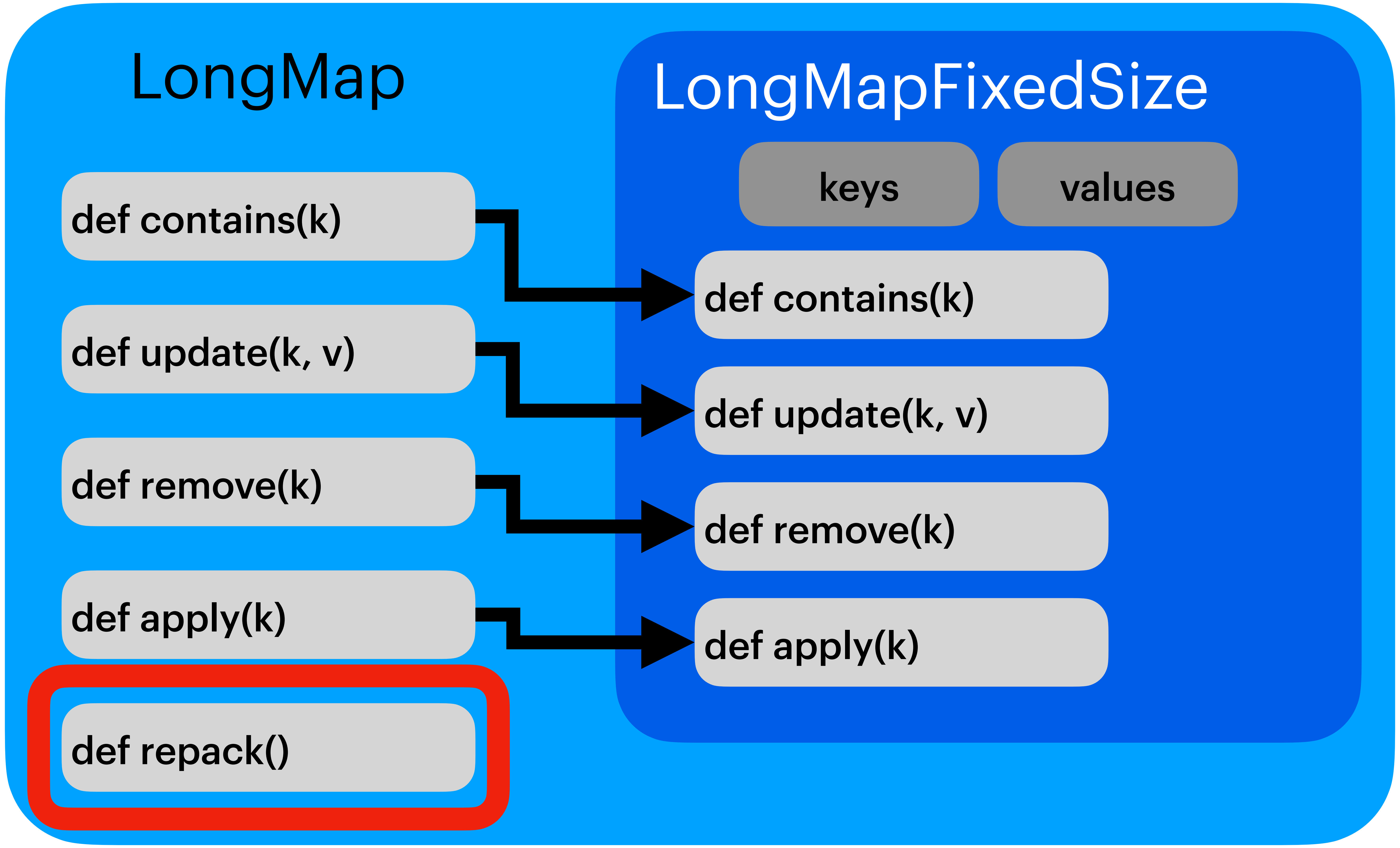
LongMap

Decorator Pattern



LongMap

Decorator Pattern



Repack

Algorithm Pseudo Code

```
// Resize arrays and rebalance keys (pseudocode)
def repack() =
    val size = this.computeArraySize()
    val newMap = new LongMapFixedSize(size)
    for k, v <- this do
        newMap.update(k, v)
    this.underlying = newMap
```

Repack

Algorithm Pseudo Code

```
// Resize arrays and rebalance keys (pseudocode)
def repack() =
    val size = this.computeArraySize()
    val newMap = new LongMapFixedSize(size)
    for k, v <- this do
        newMap.update(k, v)
    this.underlying = newMap
```

Aliasing!

Repack

Algorithm Pseudo Code

```
// Resize arrays and rebalance keys (pseudocode)
def repack() =
    val size = this.computeArraySize()
    val newMap = new LongMapFixedSize(size)
    for k, v <- this do
        newMap.update(k, v)
    this.underlying = newMap
```

Aliasing!

Stainless disallows this kind of aliasing!

Repack

Algorithm Pseudo Code

```
// Resize arrays and rebalance keys (pseudocode)
def repack() =
    val size = this.computeArraySize()
    val newMap = new LongMapFixedSize(size)
    for k, v <- this do
        newMap.update(k, v)
    this.underlying = newMap
```

Aliasing!

Stainless disallows this kind of aliasing!

We introduce a new structure: `Cell`

`Cell` & Swap Operation

Aliasing in repack

`Cell` & Swap Operation

Aliasing in repack

```
class Cell[@mutable T](var v: T):
  def swap(other: Cell[T])
  def v(): T
```

`Cell` & Swap Operation

Aliasing in repack

```
class Cell[@mutable T](var v: T):
  def swap(other: Cell[T])
  def v(): T
```

```
// Resize arrays and rebalance keys (pseudocode)
def repack() =
  val size = this.computeArraySize()
  val newMap = Cell(new LongMapFixedSize(size))
  for k, v <- this do
    newMap.v().update(k, v)
  this.underlying.swap(newMap)
```

`Cell` & Swap Operation

Aliasing in repack

```
class Cell[@mutable T](var v: T):
  def swap(other: Cell[T])
  def v(): T
```

```
// Resize arrays and rebalance keys (pseudocode)
def repack() =
  val size = this.computeArraySize()
  val newMap = Cell(new LongMapFixedSize(size))
  for k, v <- this do
    newMap.v().update(k, v)
  this.underlying.swap(newMap)
```

⇒ Greater expressiveness without introducing aliasing

Cell and Swap

Stainless Internal

```scala
case class MutA(var x: BigInt)

def exSwap(): Unit =
  val m1 = MutA(0)
  val m2 = MutA(1)
  val c1 = Cell(m1)
  val c2 = Cell(m2)
  assert(c1.v.x == 0)
  swap(c1, c2)
  assert(c1.v.x == 1)
```

Cell and Swap

Stainless Internal

```scala
case class MutA(var x: BigInt)

def exSwap(): Unit =
  val m1 = MutA(0)
  val m2 = MutA(1)
  val c1 = Cell(m1)
  val c2 = Cell(m2)
  assert(c1.v.x == 0)
  swap(c1, c2)
  assert(c1.v.x == 1)
```

Transformation

```scala
def exSwap: Unit =
  val m1: MutA = MutA(0)
  val m2: MutA = MutA(1)
  var c1: Cell[MutA] = Cell[MutA](m1)
  var c2: Cell[MutA] = Cell[MutA](m2)
  assert(c1.v.x == 0)
  val temp: MutA = c2.v
  c2 = Cell[MutA](c1.v)
  c1 = Cell[MutA](temp)
  assert(c1.v.x == 1)
  val res: Unit = ()
  val res: Unit = res
  res
```

Cell and Swap

Stainless Internal

```scala
case class MutA(var x: BigInt)

def exSwap(): Unit =
  val m1 = MutA(0)
  val m2 = MutA(1)
  val c1 = Cell(m1)
  val c2 = Cell(m2)
  assert(c1.v.x == 0)
  swap(c1, c2)
  assert(c1.v.x == 1)
```

Transformation

```scala
def exSwap: Unit =
  val m1: MutA = MutA(0)
  val m2: MutA = MutA(1)
  var c1: Cell[MutA] = Cell[MutA](m1)
  var c2: Cell[MutA] = Cell[MutA](m2)
  assert(c1.v.x == 0)
  val temp: MutA = c2.v
  c2 = Cell[MutA](c1.v)
  c1 = Cell[MutA](temp)
  assert(c1.v.x == 1)
  val res: Unit = ()
  val res: Unit = res
  res
```

Cell and Swap

Stainless Internal

```scala
case class MutA(var x: BigInt)

def exSwap(): Unit =
  val m1 = MutA(0)
  val m2 = MutA(1)
  val c1 = Cell(m1)
  val c2 = Cell(m2)
  assert(c1.v.x == 0)
  swap(c1, c2)
  assert(c1.v.x == 1)
```

Transformation

```scala
def exSwap: Unit =
  val m1: MutA = MutA(0)
  val m2: MutA = MutA(1)
  var c1: Cell[MutA] = Cell[MutA](m1)
  var c2: Cell[MutA] = Cell[MutA](m2)
  assert(c1.v.x == 0)
  val temp: MutA = c2.v
  c2 = Cell[MutA](c1.v)
  c1 = Cell[MutA](temp)
  assert(c1.v.x == 1)
  val res: Unit = ()
  val res: Unit = res
  res
```

This transformation is performed **before eliminating imperative code**

HashMap Interface

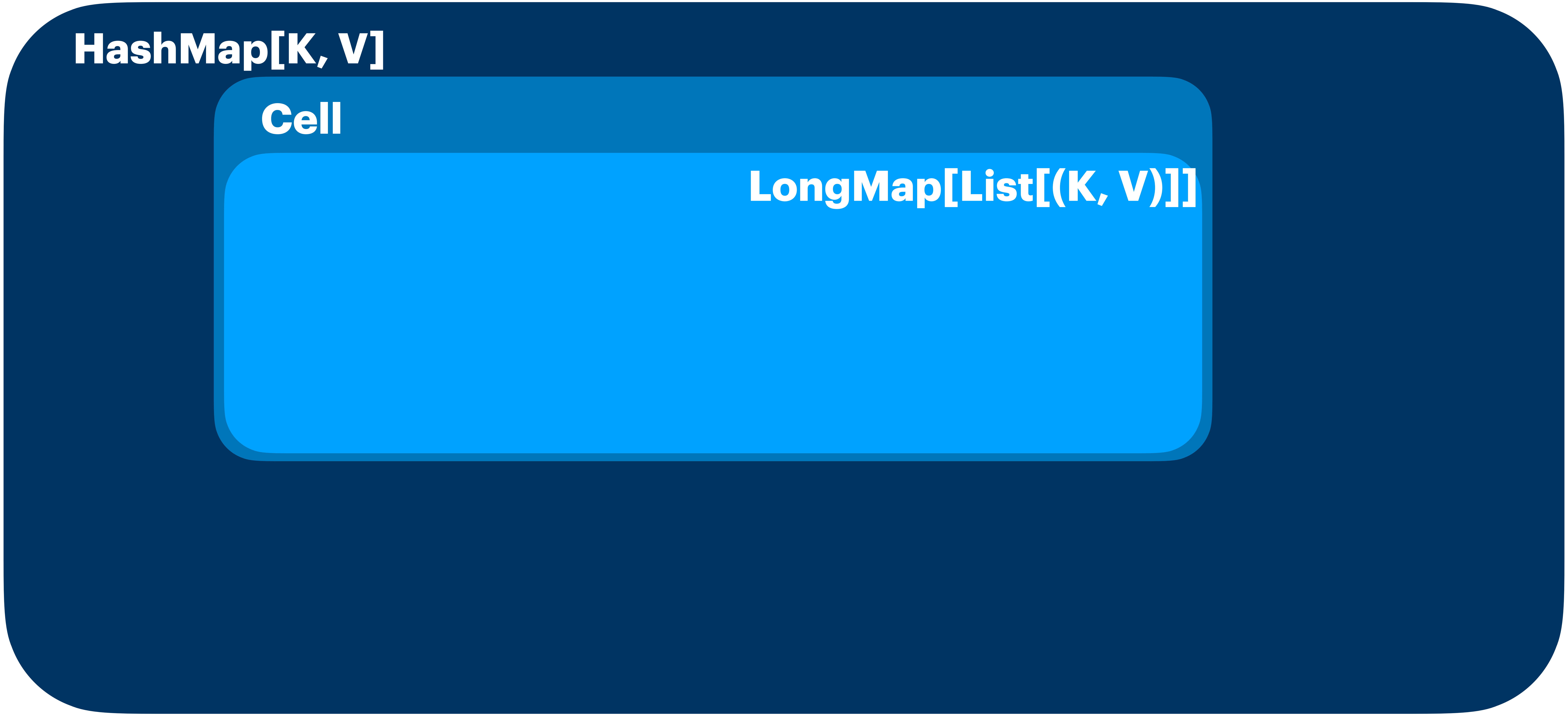
Hash Table, Generically Typed Keys

```
trait HashMap[K, V]:
  def contains(key: K): Boolean
  def apply(key: K): V                    // Lookup
  def update(key: K, v: V): Boolean
  def remove(key: K): Boolean
  def repack(): Boolean

trait Hashable[K]: // “Type class”
  @pure
  def hash(k: K): Long
```

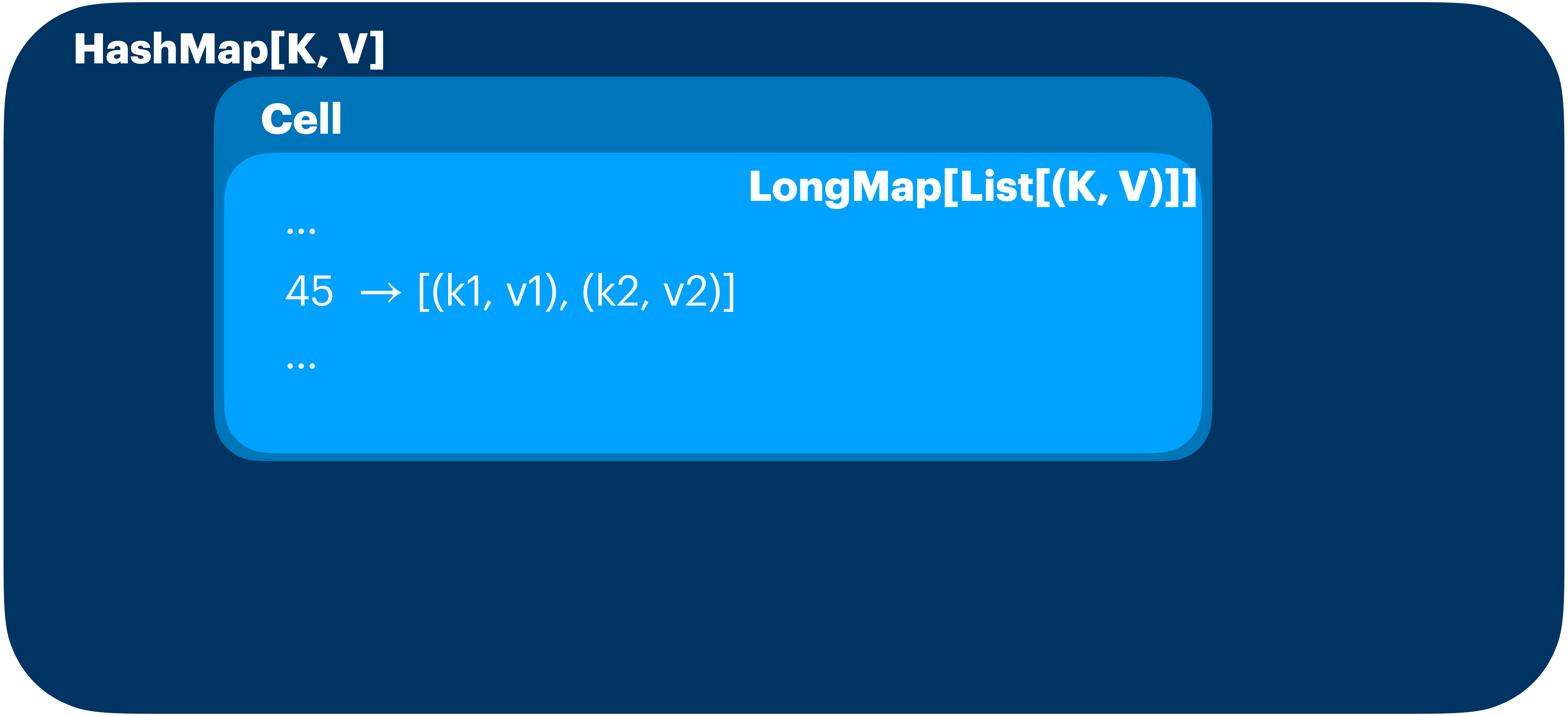
Implementation

Insert (k3: K, v3: V)



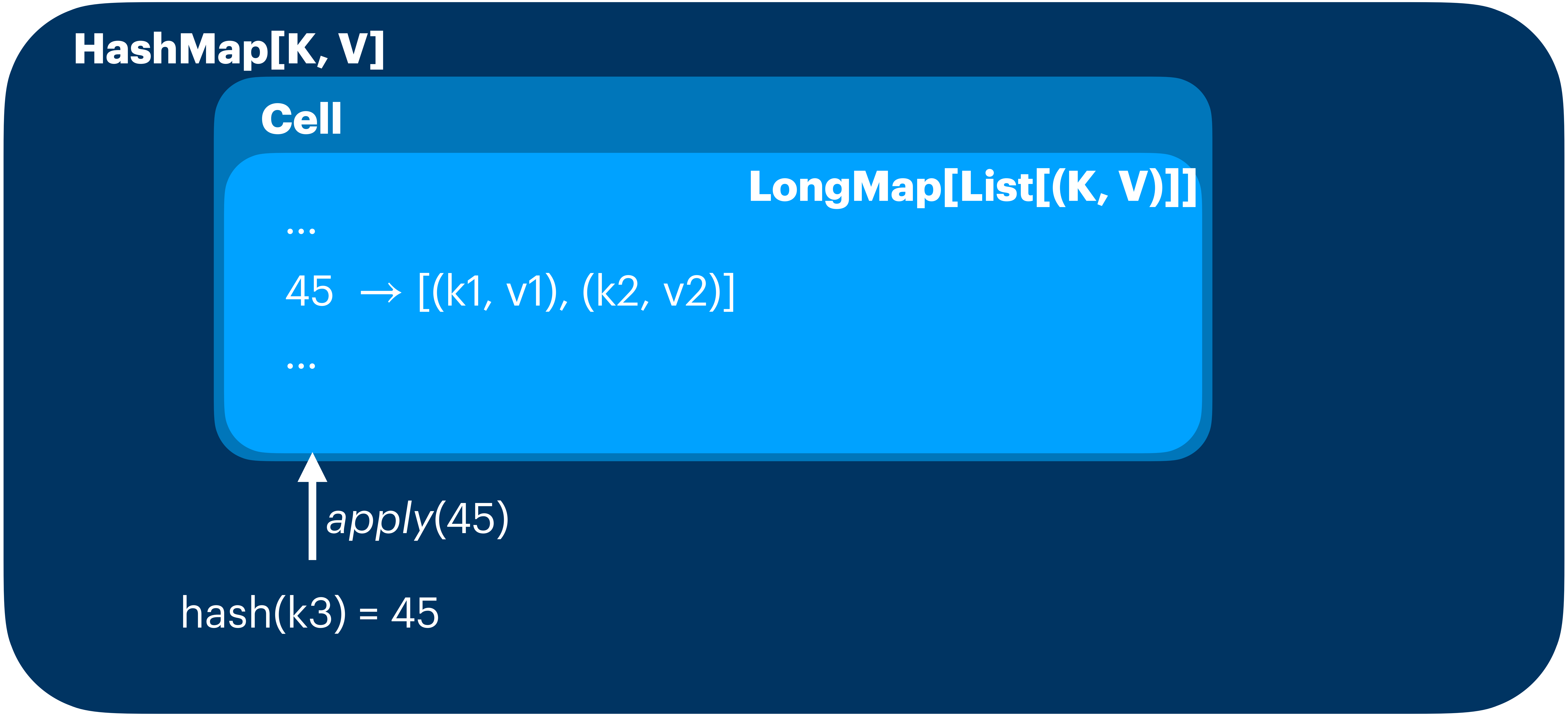
Implementation

Insert (k3: K, v3: V)



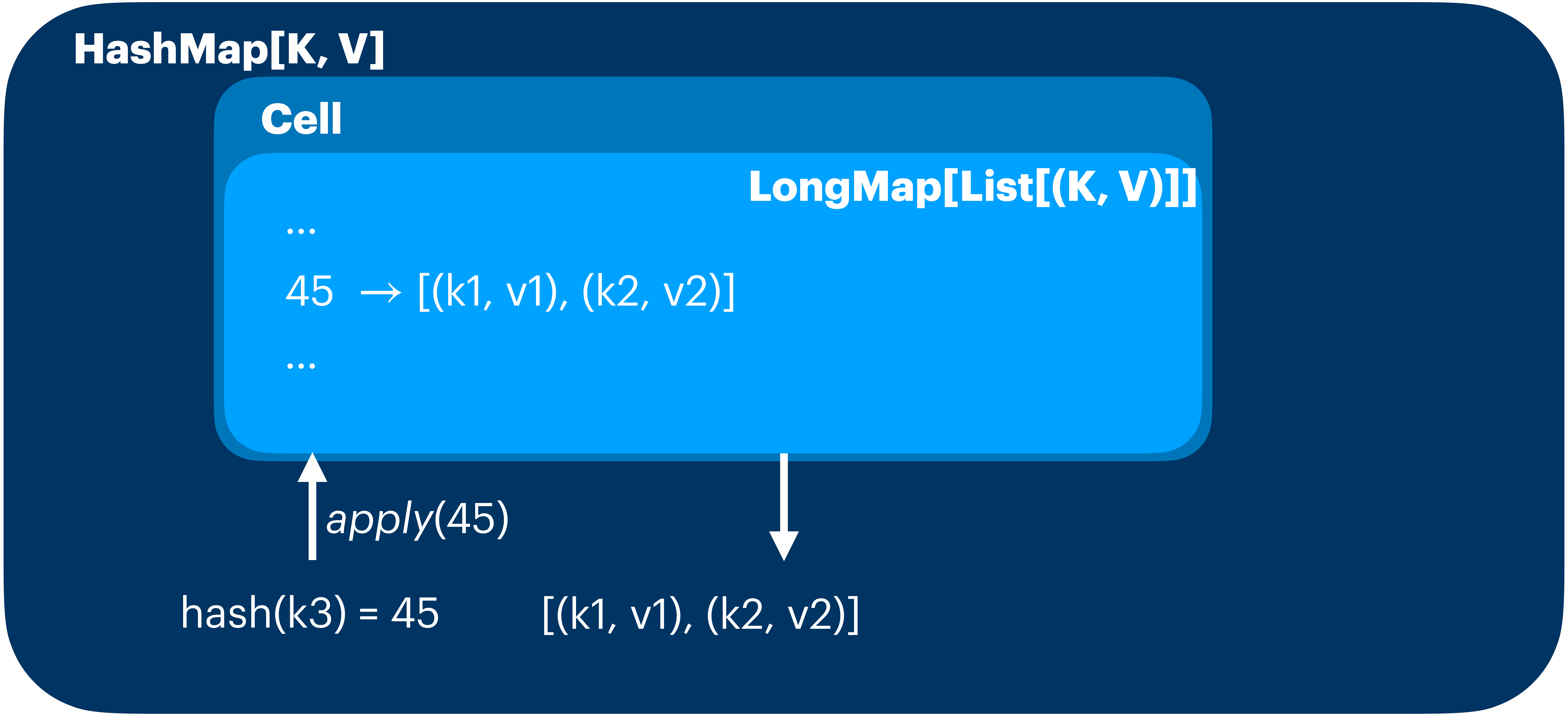
Implementation

Insert (k3: K, v3: V)



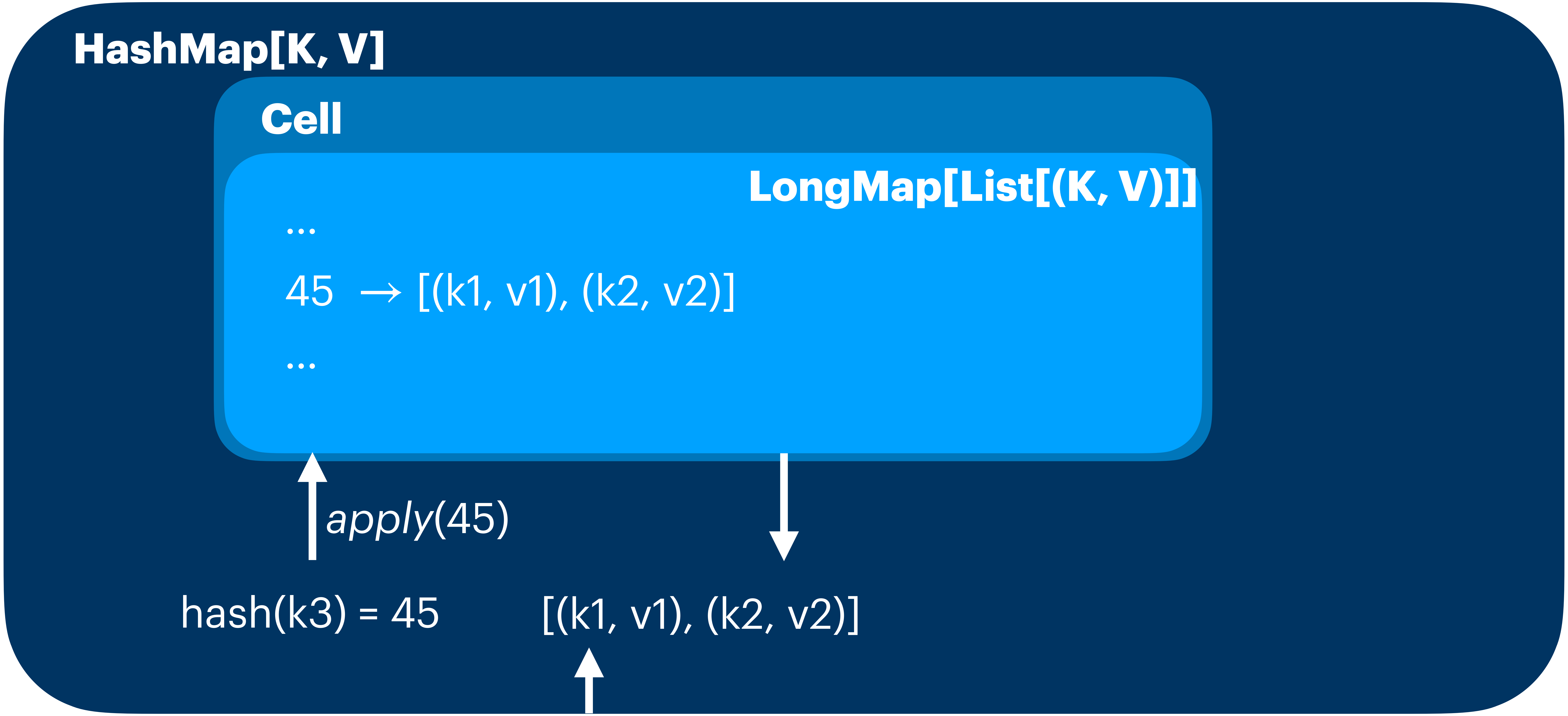
Implementation

Insert (k3: K, v3: V)



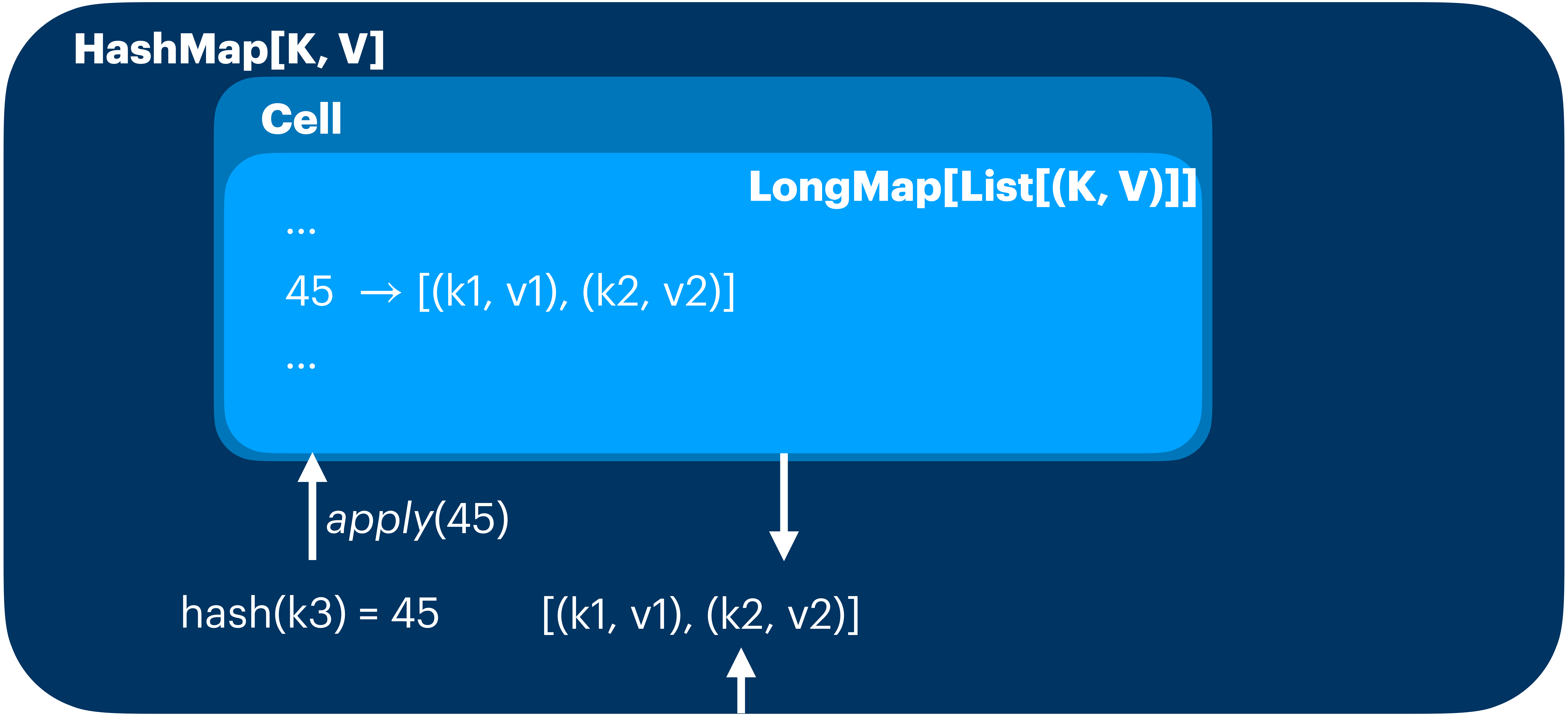
Implementation

Insert (k3: K, v3: V)



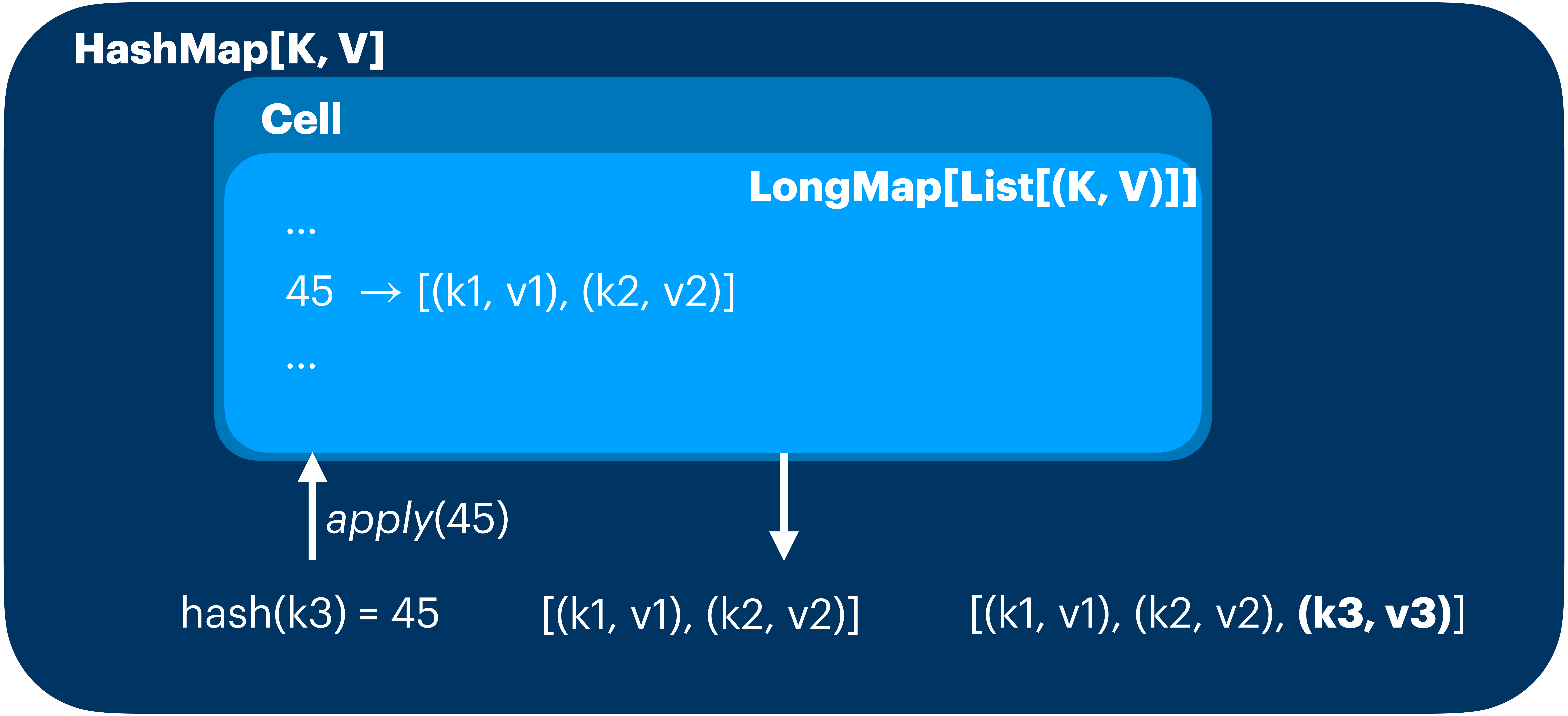
Implementation

Insert (k3: K, v3: V)



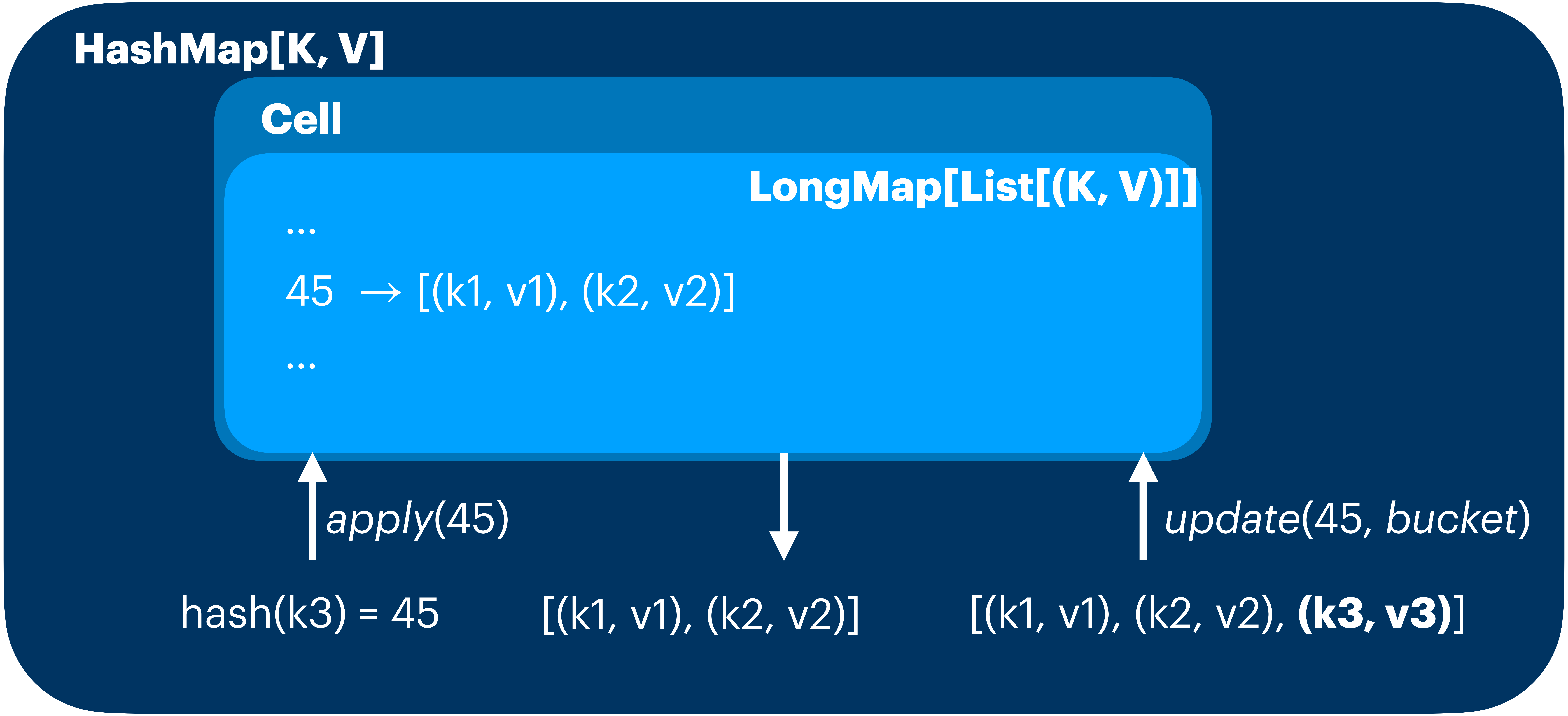
Implementation

Insert (k3: K, v3: V)



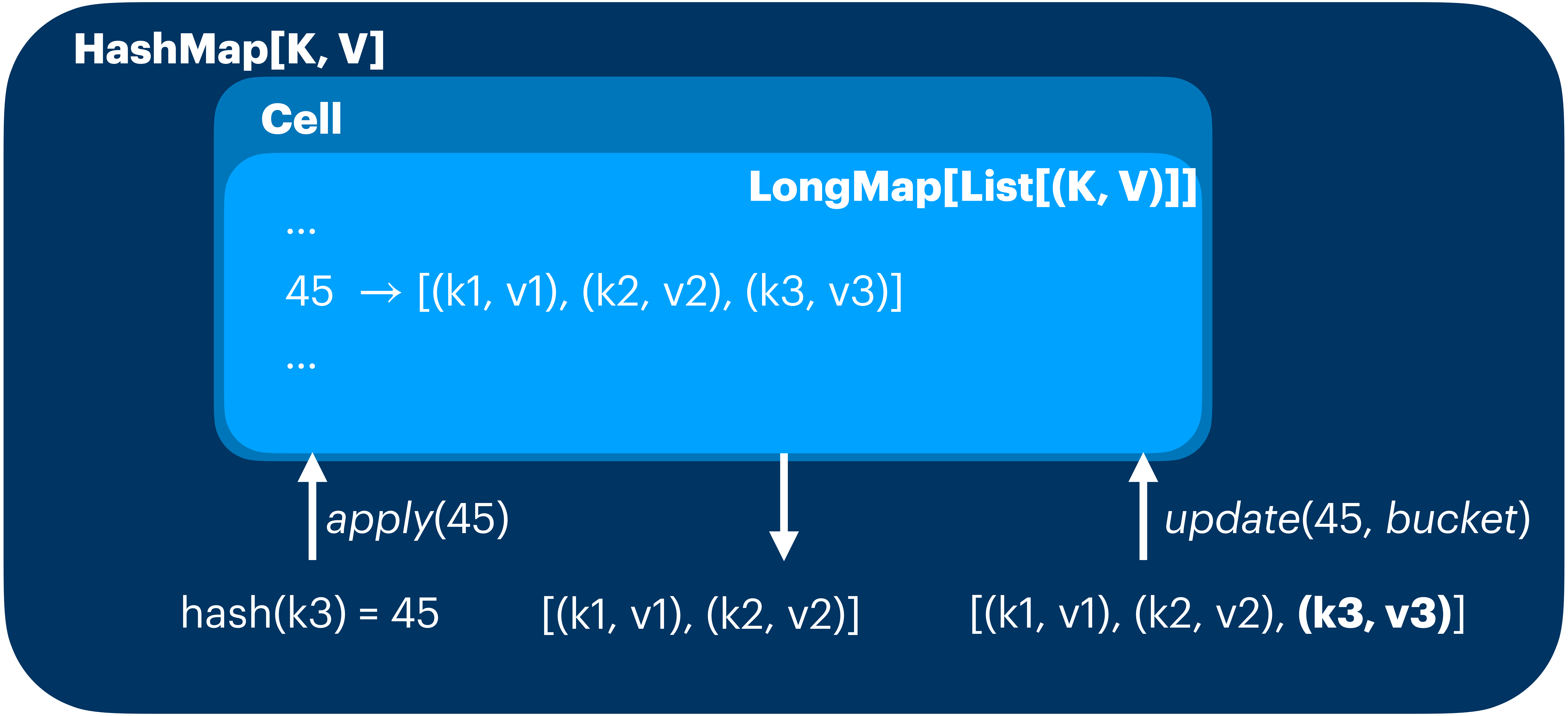
Implementation

Insert (k3: K, v3: V)



Implementation

Insert (k3: K, v3: V)



Memoization

Tailored Lemmas

$$valid(cache) = \forall a : A \;\; cache(a) = some(b) \implies b = f(a)$$

Memoization

Tailored Lemmas

$$valid(cache) = \forall a : A \;\; cache(a) = some(b) \implies b = f(a)$$

```
def lemmaForallPairsThenForLookup[K, V](
  hm: HashMap[K, V],
  k: K,
  p: ((K, V)) => Boolean
): Unit = {
  require(hm.valid)
  require(hm.map.forall(p))
  require(hm.contains(k))
  // ...
}.ensuring(_ => p((k, hm.apply(k))))
```

Memoization

Tailored Lemmas

$$valid(cache) = \forall a : A \ \ cache(a) = some(b) \implies b = f(a)$$

```scala
def lemmaUpdatePreservesForallPairs[K, V](
    hm: HashMap[K, V],
    k: K,
    v: V,
    p: ((K, V)) => Boolean
  ): Unit = {
    require(hm.valid)
    require(hm.map.forall(p))
    require(p((k, v)))
    // ...
  }.ensuring(_ => {
    hm.update(k, v)
    hm.map.forall(p)
  })
```

```scala
def lemmaForallPairsThenForLookup[K, V](
  hm: HashMap[K, V],
  k: K,
  p: ((K, V)) => Boolean
): Unit = {
  require(hm.valid)
  require(hm.map.forall(p))
  require(hm.contains(k))
  // ...
}.ensuring(_ => p((k, hm.apply(k))))
```

Memoization

Tailored Lemmas

$$valid(cache) = \forall a : A \ \ cache(a) = some(b) \implies b = f(a)$$

```
def lemmaUpdatePreservesForallPairs[K, V](
    hm: HashMap[K, V],
    k: K,
    v: V,
    p: ((K, V)) => Boolean
  ): Unit = {
    require(hm.valid)
    require(hm.map.forall(p))
    require(p((k, v)))
    // ...
  }.ensuring(_ => {
    hm.update(k, v)
    hm.map.forall(p)
  })
```

```
def lemmaForallPairsThenForLookup[K, V](
  hm: HashMap[K, V],
  k: K,
  p: ((K, V)) => Boolean
): Unit = {
  require(hm.valid)
  require(hm.map.forall(p))
  require(hm.contains(k))
  // ...
}.ensuring(_ => p((k, hm.apply(k))))
```

```
def lemmaRemovePreservesForallPairs[K, V](
    hm: HashMap[K, V],
    k: K,
    p: ((K, V)) => Boolean
  ): Unit = {
    require(hm.valid)
    require(hm.map.forall(p))
    // ...
  }.ensuring(_ => {
    hm.remove(k)
    hm.map.forall(p)
  })
```

Decorator and Composition

Powerful Pattern for Verification

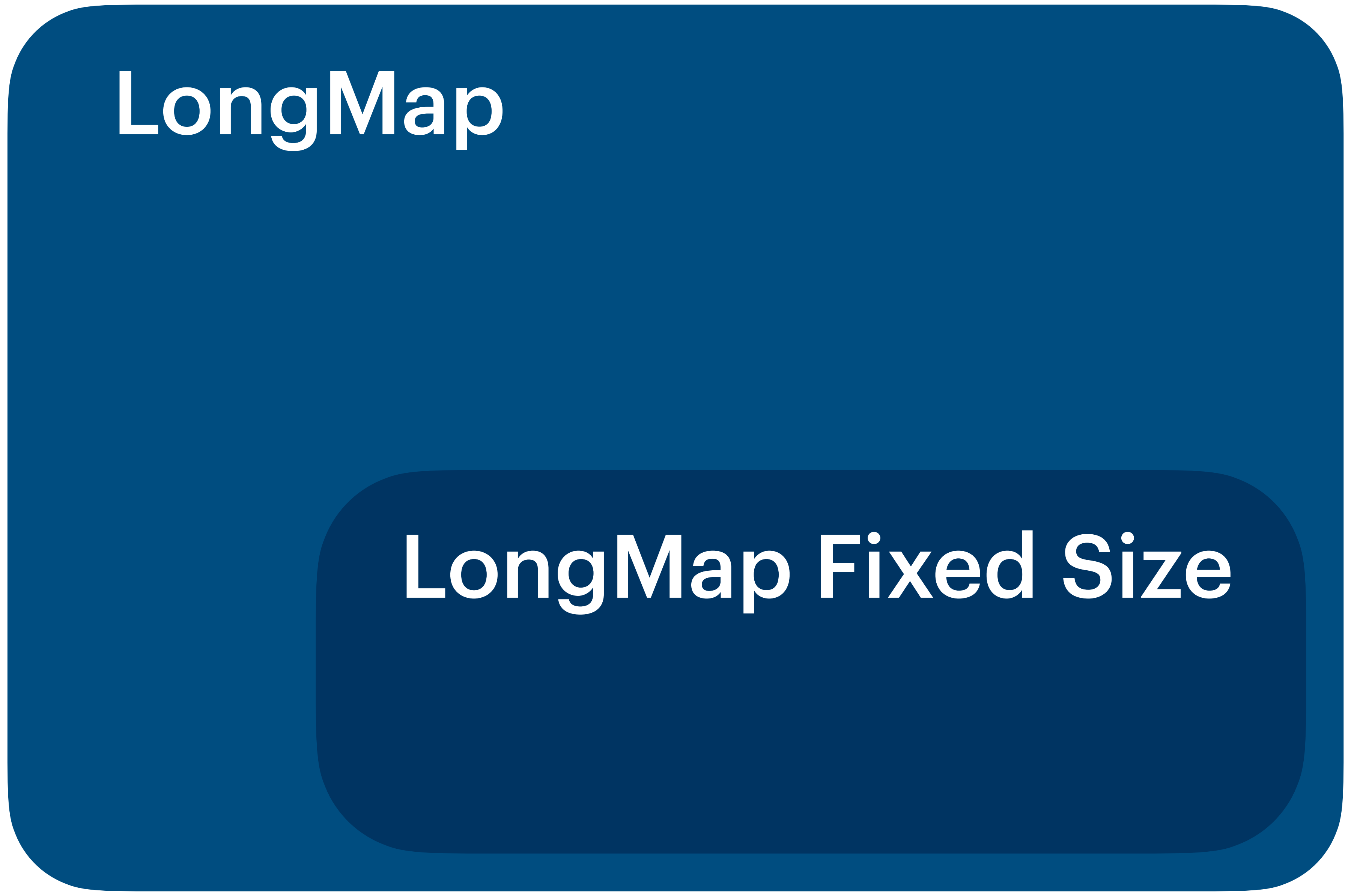
Decorator and Composition

Powerful Pattern for Verification

LongMap Fixed Size

Decorator and Composition

Powerful Pattern for Verification

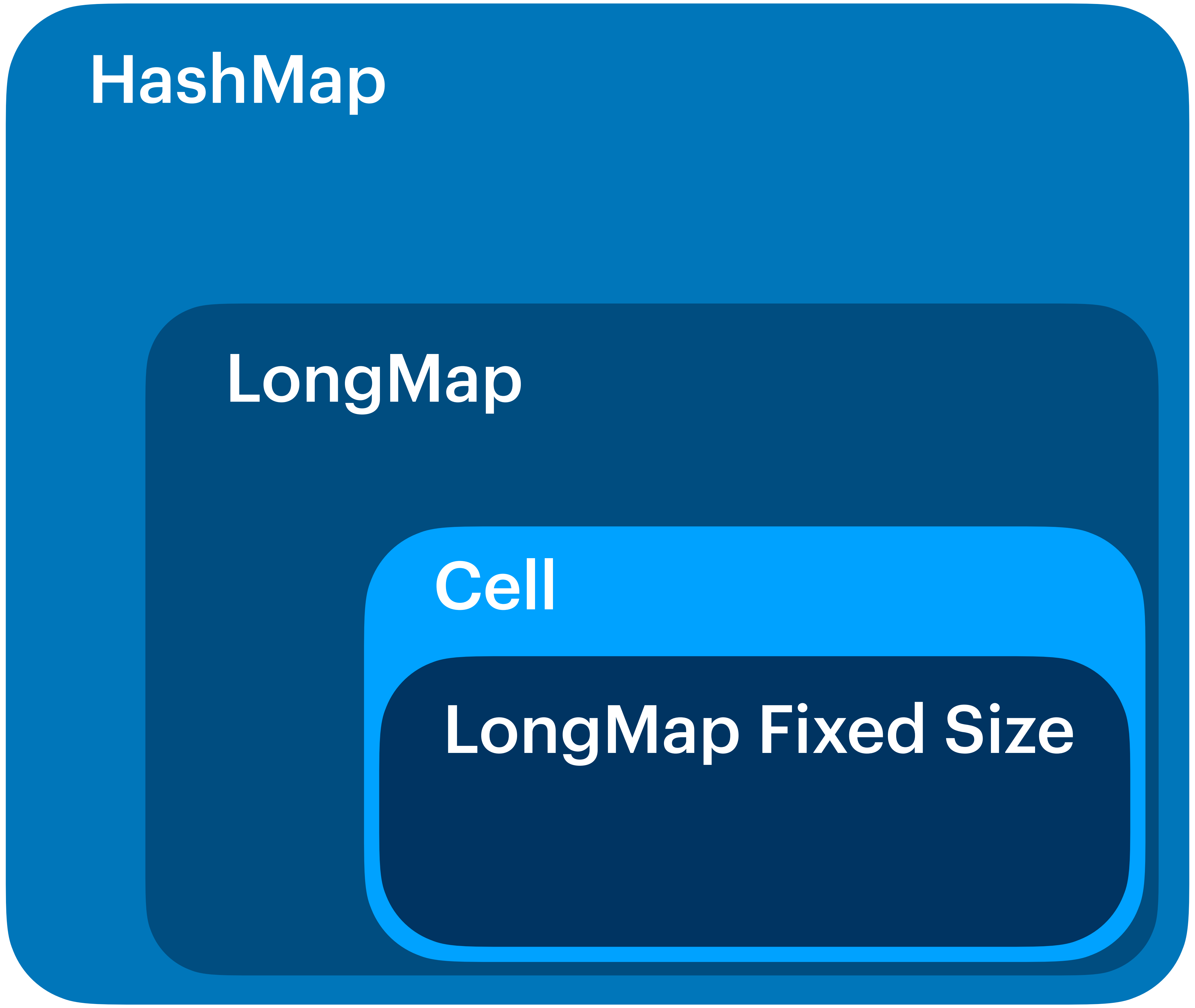


Decorator and Composition

Powerful Pattern for Verification

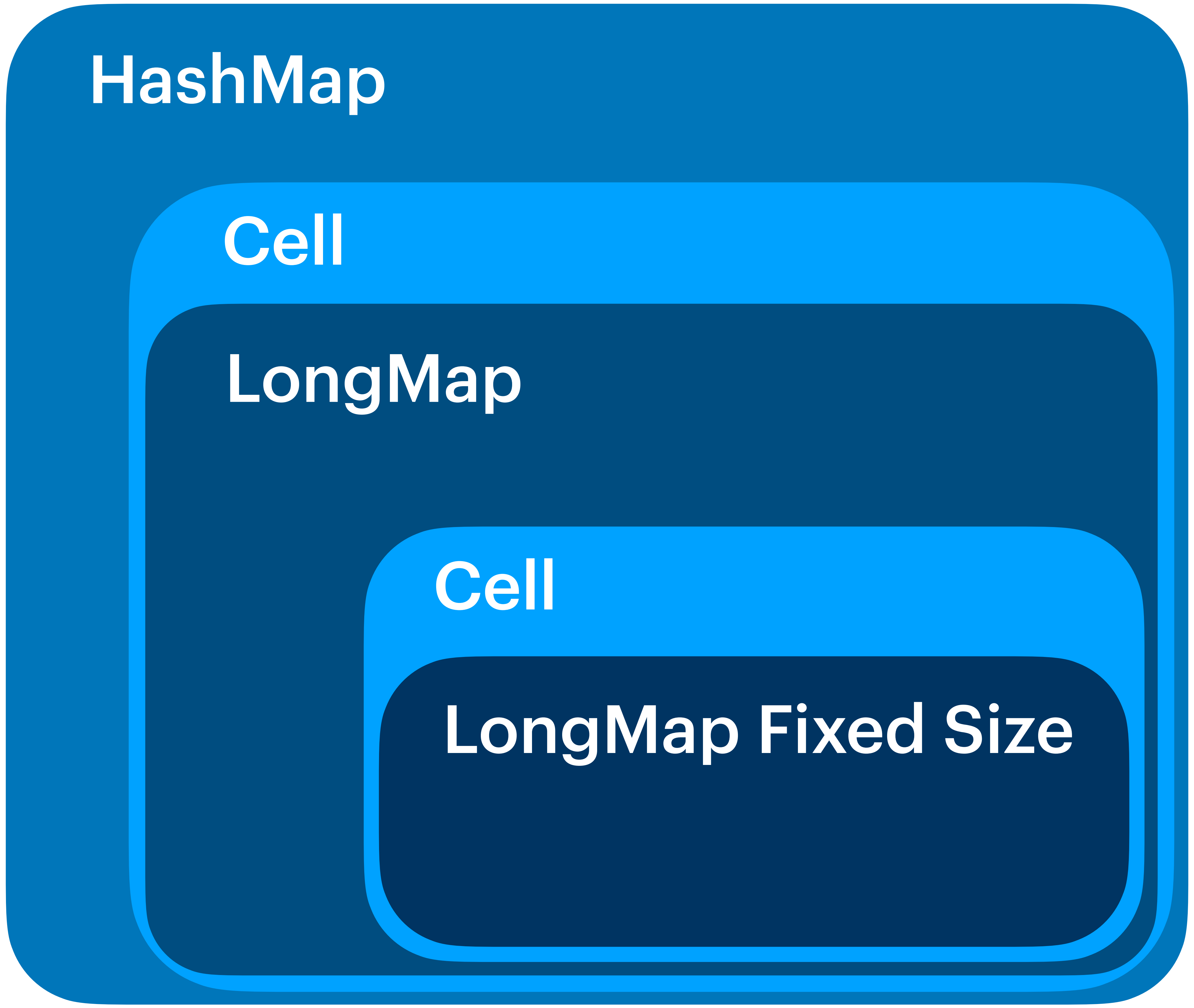
Decorator and Composition

Powerful Pattern for Verification



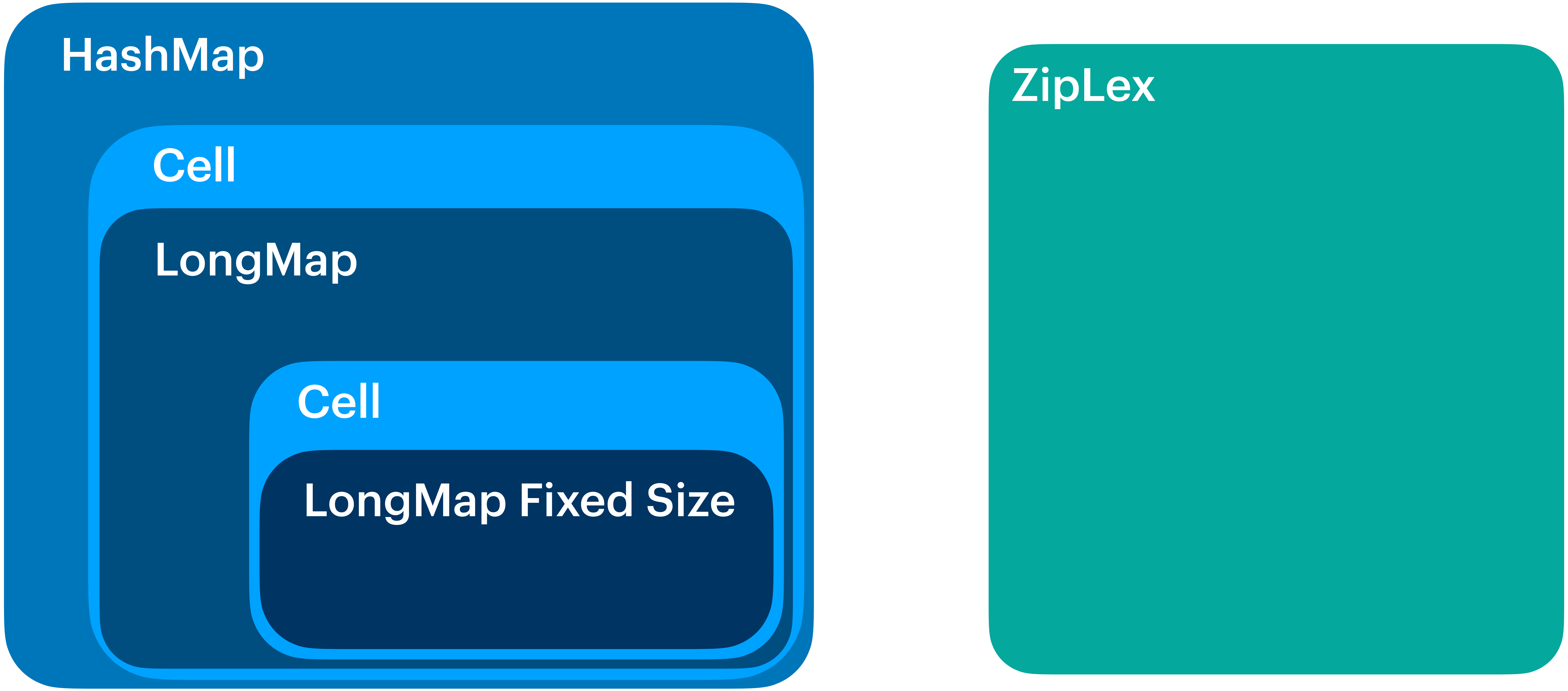
Decorator and Composition

Powerful Pattern for Verification



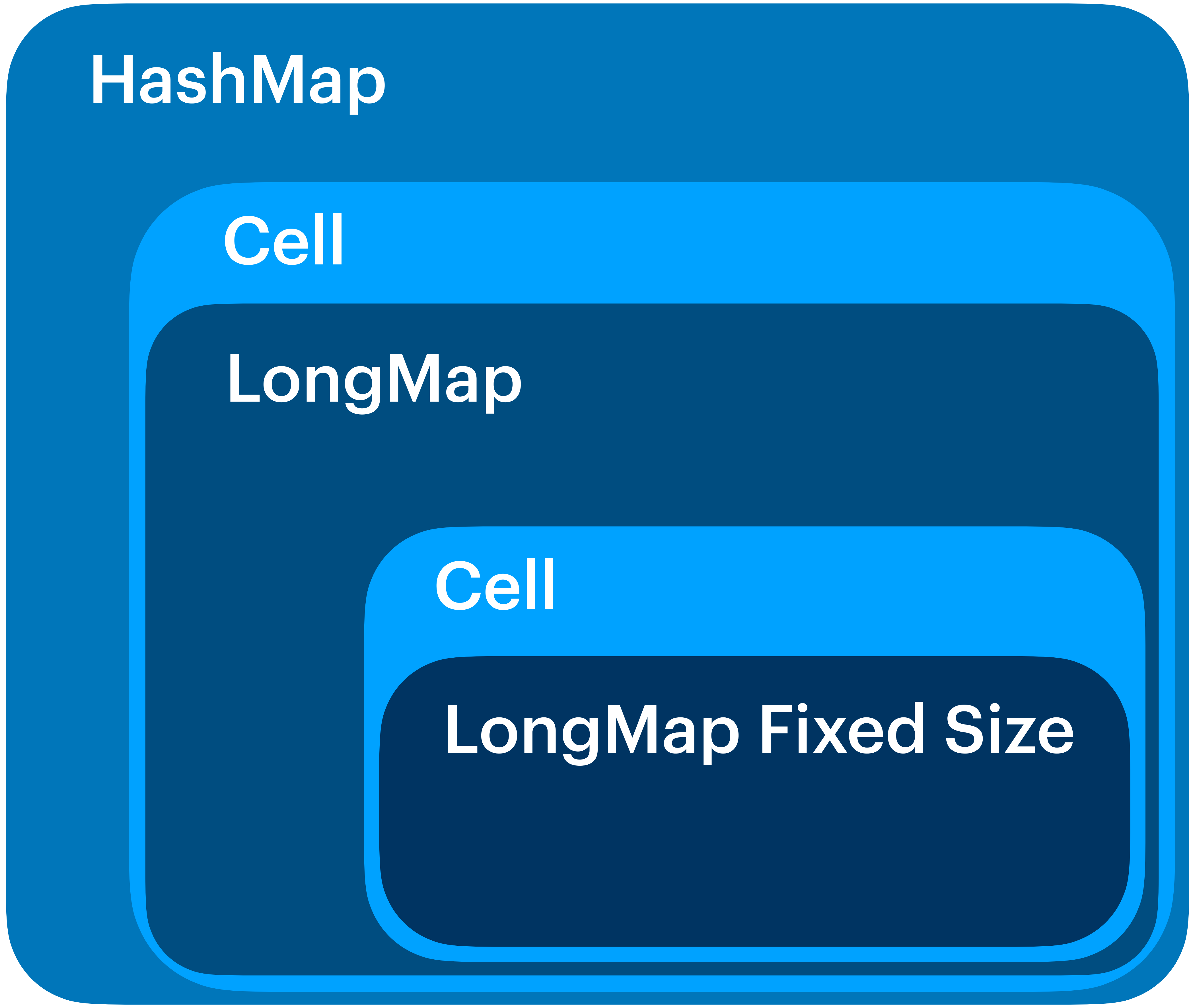
Decorator and Composition

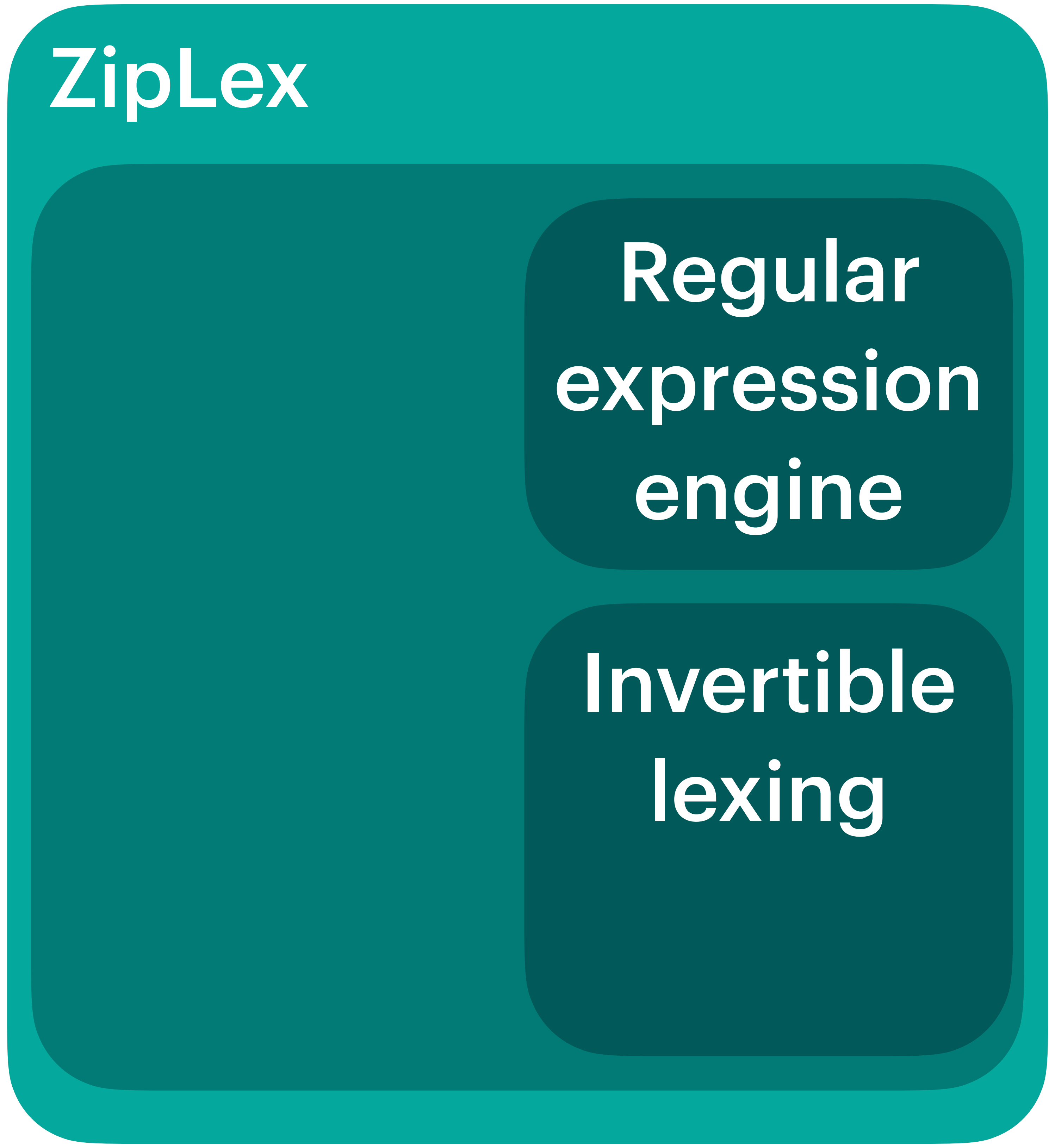
Powerful Pattern for Verification



Decorator and Composition

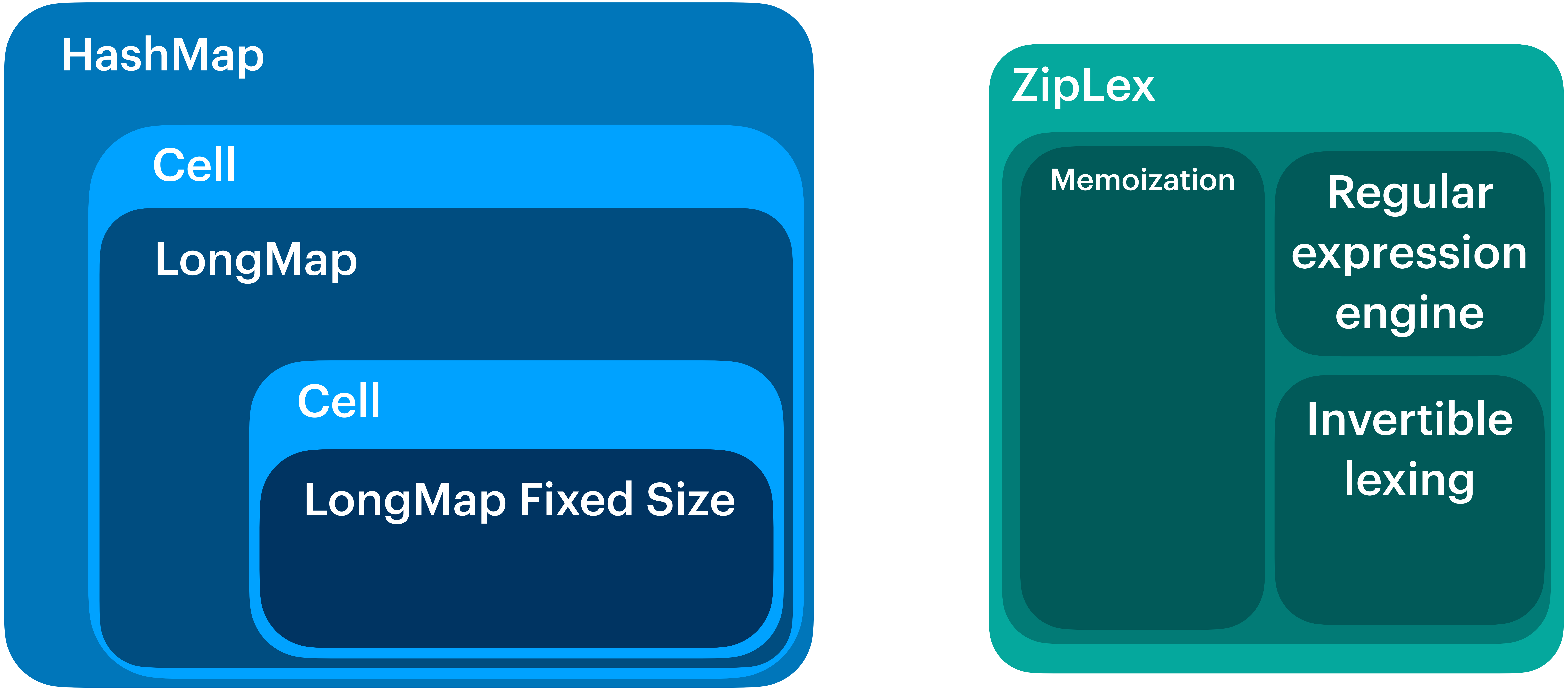
Powerful Pattern for Verification





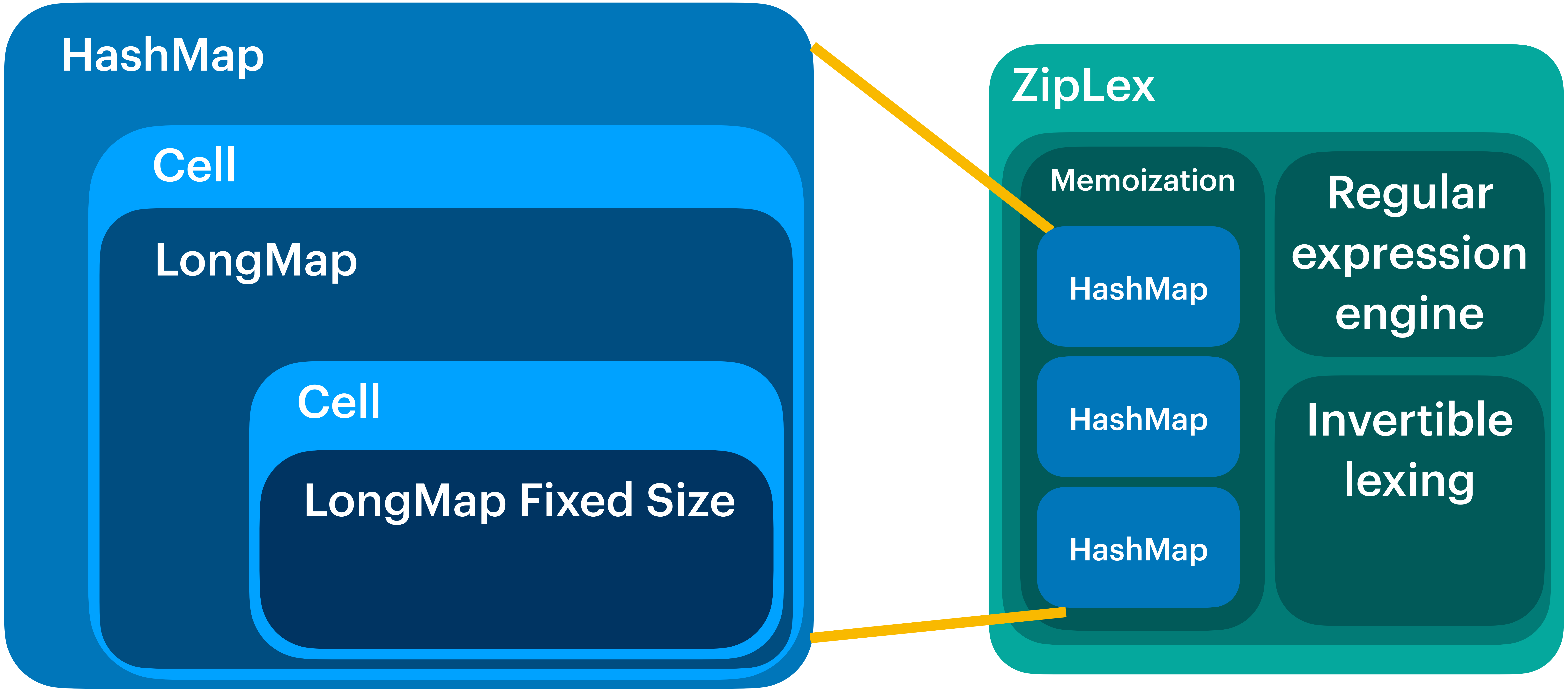
Decorator and Composition

Powerful Pattern for Verification



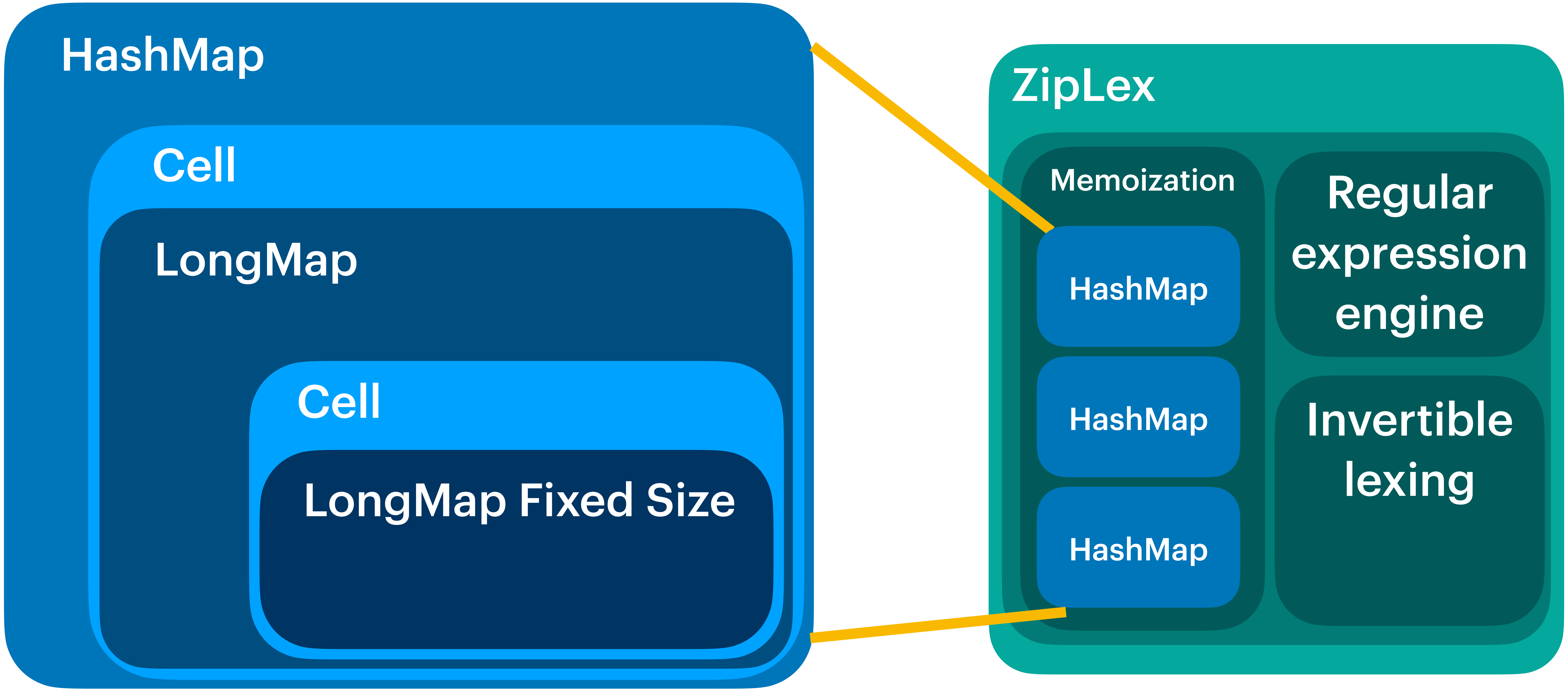
Decorator and Composition

Powerful Pattern for Verification



Decorator and Composition

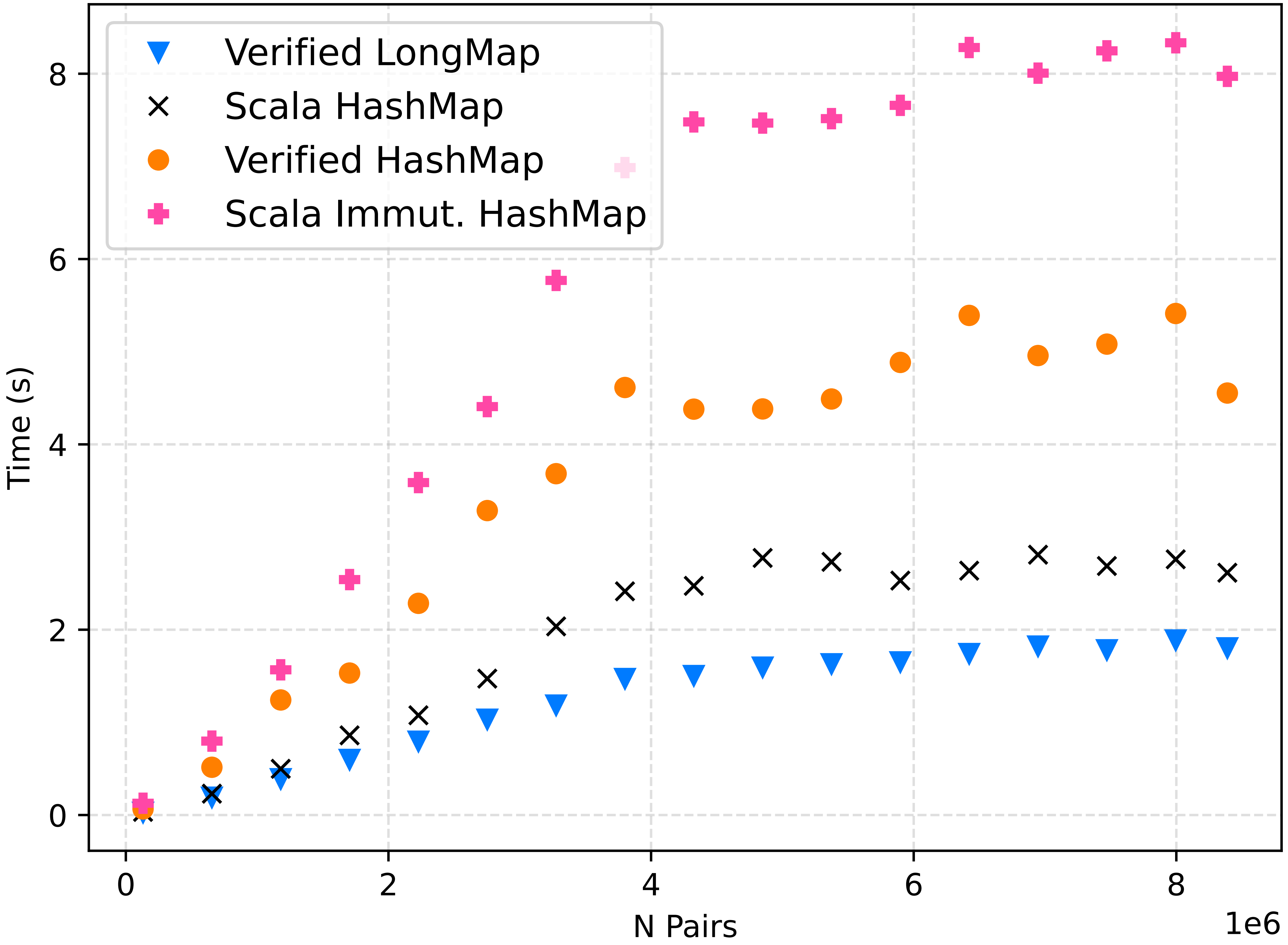
Powerful Pattern for Verification



Enabled by the Cell construct without introducing aliasing

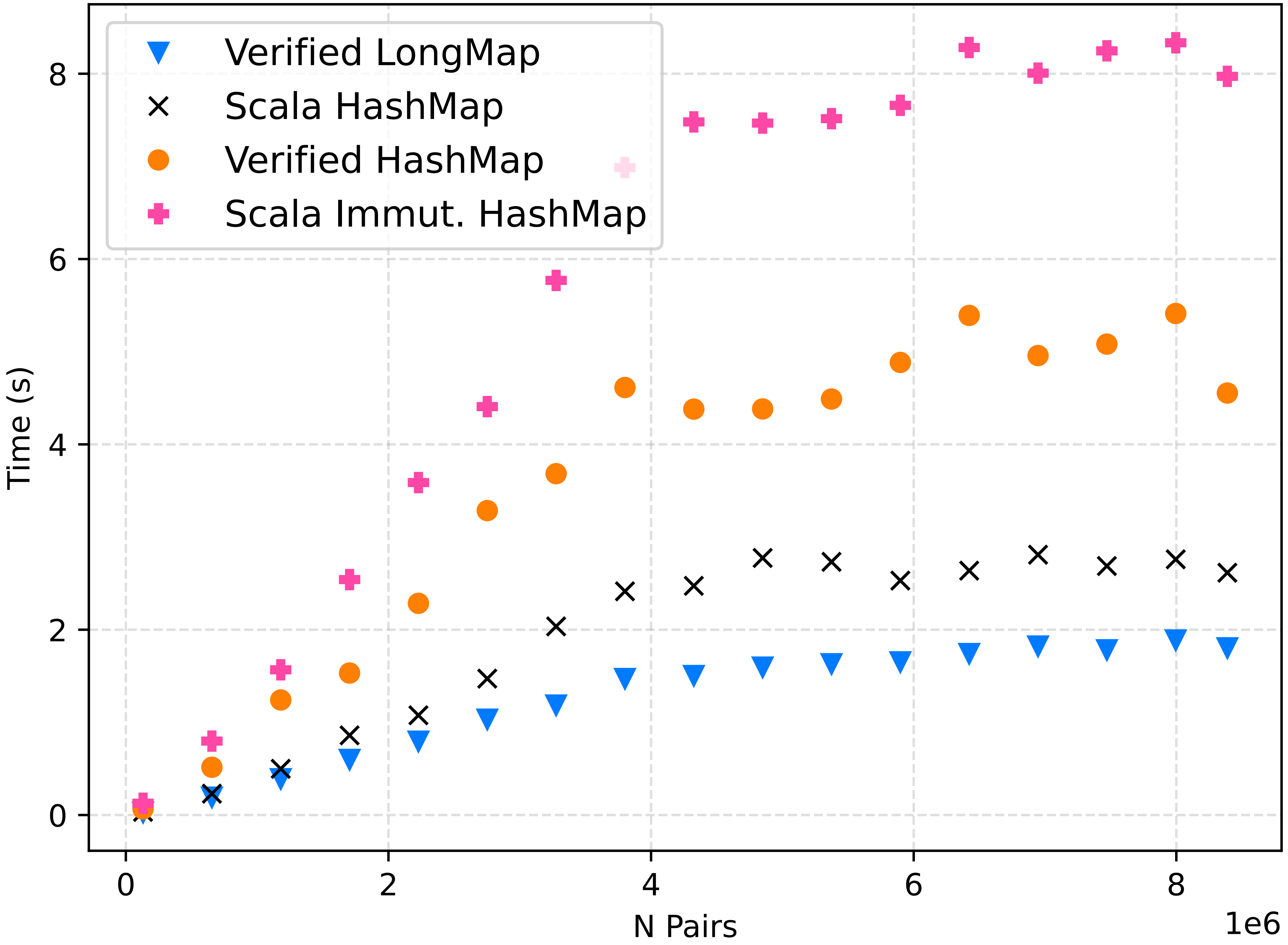
Maps Performance

Maps Performance



Insert N pairs, remove N/2, insert N

Maps Performance



Insert N pairs, remove N/2, insert N

⇒ Our **verified mutable** hash table outperforms the Scala library **unverified immutable** map

Verified Lexing with Memoization

ZipLex[2]

Application Relying on Memoization

[2] **Samuel Chassot**, & Viktor Kunčak. (2026). Formally Verified Linear-Time Invertible Lexing.

ZipLex[2]

Application Relying on Memoization

Fully **verified lexical analysis** framework

[2] **Samuel Chassot**, & Viktor Kunčak. (2026). Formally Verified Linear-Time Invertible Lexing.

ZipLex[2]

Application Relying on Memoization

Fully **verified lexical analysis** framework

Linear-time, invertible lexing

[2]**Samuel Chassot**, & Viktor Kunčak. (2026). Formally Verified Linear-Time Invertible Lexing.

ZipLex[2]

Application Relying on Memoization

Fully **verified lexical analysis** framework

Linear-time, invertible lexing

Regex matching based on **Brzozowski derivatives**

[2]**Samuel Chassot**, & Viktor Kunčak. (2026). Formally Verified Linear-Time Invertible Lexing.

ZipLex[2]

Application Relying on Memoization

Fully **verified lexical analysis** framework

Linear-time, invertible lexing

Regex matching based on **Brzozowski derivatives**

Verified **performance** optimizations including **memoization**

[2]**Samuel Chassot**, & Viktor Kunčak. (2026). Formally Verified Linear-Time Invertible Lexing.

Lexical Analysis

Introduction

Lexical Analysis

Introduction

Stream of **characters** → stream of **tokens**

Lexical Analysis

Introduction

Stream of **characters** → stream of **tokens**

Rules with **regular expressions**

Lexical Analysis

Introduction

Stream of **characters** → stream of **tokens**

Rules with **regular expressions**

Spec: **Longest match** and **invertibility**

Lexical Analysis

Introduction

Stream of **characters** → stream of **tokens**

Rules with **regular expressions**

Spec: **Longest match** and **invertibility**

Interface:

Lexical Analysis

Introduction

Stream of **characters** → stream of **tokens**

Rules with **regular expressions**

Spec: **Longest match** and **invertibility**

Interface:

```
def lex[C](rules: List[Rule[C]], input: List[C]):
    (List[Token[C]], List[C])
```

Lexical Analysis

Introduction

Stream of **characters** → stream of **tokens**

Rules with **regular expressions**

Spec: **Longest match** and **invertibility**

Interface:

```
def lex[C](rules: List[Rule[C]], input: List[C]):
    (List[Token[C]], List[C])

def print[C](v: List[Token[C]]): List[C]
```

Lexer Specification

Longest Match and Invertibility

Lexer Specification

Longest Match and Invertibility

Longest Match

Lexer Specification

Longest Match and Invertibility

Longest Match

When creating a token, consume the longest prefix accepted by any rule

Lexer Specification

Longest Match and Invertibility

Longest Match

When creating a token, consume the longest prefix accepted by any rule

Invertibility

Lexer Specification

Longest Match and Invertibility

Longest Match

When creating a token, consume the longest prefix accepted by any rule

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Lexer Specification

Longest Match and Invertibility

Longest Match

When creating a token, consume the longest prefix accepted by any rule

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

Lexer Specification

Longest Match and Invertibility

Longest Match

When creating a token, consume the longest prefix accepted by any rule

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$\mathit{print}(\mathit{lex}(s)) = s$$

Lexer Specification

Longest Match and Invertibility

Longest Match

When creating a token, consume the longest prefix accepted by any rule

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$\mathit{print}(\mathit{lex}(s)) = s \qquad \text{and}$$

Lexer Specification

Longest Match and Invertibility

Longest Match

When creating a token, consume the longest prefix accepted by any rule

Invertibility

For a token sequence $ts = t_1, t_2, \ldots, t_n$ and string s

Invertible lexer if both

$$\mathit{print}(\mathit{lex}(s)) = s \qquad \text{and} \qquad \mathit{lex}(\mathit{print}(ts)) = ts$$

Sorting Functions

Refactor Tool Example

```
def abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
def mul(x: Int, y: Int): Int =
  x * y
end mul
```

Sorting Functions

Refactor Tool Example

```
def abs(x: Int): Int =⏎
  if (x < 0) then⏎
    -x⏎
  else⏎
    x⏎
end abs⏎
def mul(x: Int, y: Int): Int =⏎
  x * y⏎
end mul
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, y: Int): Int =⏎
  x * y⏎
end mul
def abs(x: Int): Int =⏎
  if (x < 0) then⏎
    -x⏎
  else⏎
    x⏎
end abs⏎
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, y: Int): Int =
  x * y
end mul
def abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, y: Int): Int =
  x * y
end muldef abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, y: Int): Int =
  x * y
end muldef abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, y: Int): Int =
  x * y
end muldef abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
```

Sorting Functions

Refactor Tool Example

```
def mul(x: Int, y: Int): Int =
  x * y
end muldef abs(x: Int): Int =
  if (x < 0) then
    -x
  else
    x
end abs
```

ZipLex Memoization

Lexer and Regex engine

ZipLex Memoization

Lexer and Regex engine

Memoized zipper-**derivatives computation**

ZipLex Memoization

Lexer and Regex engine

Memoized zipper-**derivatives computation**

yielding DFAs

ZipLex Memoization

Lexer and Regex engine

Memoized zipper-**derivatives computation**

yielding DFAs

Memoization enables linear **complexity lexing** (REPS 1998)

ZipLex Memoization

Lexer and Regex engine

Memoized zipper-**derivatives computation**

  yielding DFAs

Memoization enables linear **complexity lexing** (REPS 1998)

  Longest prefix computation

ZipLex Memoization

Lexer and Regex engine

Memoized zipper-**derivatives computation**

  yielding DFAs

Memoization enables linear **complexity lexing** (REPS 1998)

  Longest prefix computation

  Tail-recursive $\Rightarrow$ the technique does not apply

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCause(r)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCause(r)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCause(r)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]

def prefixFromMem[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
  (using c: MutableCache[C]): BigInt =
  c.get(r, from) match
    case Some(v) => v
    case None() =>
      val result =
        if (from == input.size || lostCauseZipper(z)) then acc
        else
          val newAcc = if (nullable(r.deriv(input(from)))) from else acc
          prfxSFromMem(r, from + 1, input, newAcc)
      c.update(r, from, result)
      result
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCause(r)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
def prefixFromMem[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
  (using c: MutableCache[C]): BigInt =
  c.get(r, from) match
    case Some(v) => v
    case None() =>
      val result =
        if (from == input.size || lostCauseZipper(z)) then acc
        else
          val newAcc = if (nullable(r.deriv(input(from)))) from else acc
          prfxSFromMem(r, from + 1, input, newAcc)
      c.update(r, from, result)
      result
```

Not tail recursive anymore!

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])
@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt =
  c.get(r, from, acc) match
    case Some(v) => v
    case None() =>
      if (from == input.size || lostCauseZipper(z)) then acc
      else
        val newAcc = if (nullable(r.deriv(input(from)))) from else acc
        stack.frames = (r, from, newAcc) :: stack.frames
        prefixFromMemI(r, from + 1, input, newAcc, stack)
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])
@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt =
  c.get(r, from, acc) match
    case Some(v) => v
    case None() =>
      if (from == input.size || lostCauseZipper(z)) then acc
      else
        val newAcc = if (nullable(r.deriv(input(from)))) from else acc
        stack.frames = (r, from, newAcc) :: stack.frames
        prefixFromMemI(r, from + 1, input, newAcc, stack)
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])
@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt =
  c.get(r, from, acc) match
    case Some(v) => v
    case None() =>
      if (from == input.size || lostCauseZipper(z)) then acc
      else
        val newAcc = if (nullable(r.deriv(input(from)))) from else acc
        stack.frames = (r, from, newAcc) :: stack.frames
        prefixFromMemI(r, from + 1, input, newAcc, stack)
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])
@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt =
  c.get(r, from, acc) match
    case Some(v) => v
    case None() =>
      if (from == input.size || lostCauseZipper(z)) then acc
      else
        val newAcc = if (nullable(r.deriv(input(from)))) from else acc
        stack.frames = (r, from, newAcc) :: stack.frames
        prefixFromMemI(r, from + 1, input, newAcc, stack)
```

Memoization

Tail recursive functions

```scala
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```scala
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])
@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt =
  c.get(r, from, acc) match
    case Some(v) => v
    case None() =>
      if (from == input.size || lostCauseZipper(z)) then acc
      else
        val newAcc = if (nullable(r.deriv(input(from)))) from else acc
        stack.frames = (r, from, newAcc) :: stack.frames
        prefixFromMemI(r, from + 1, input, newAcc, stack)
```

Memoization

Tail recursive functions

```scala
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```scala
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])
@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt =
  c.get(r, from, acc) match
    case Some(v) => v
    case None() =>
      if (from == input.size || lostCauseZipper(z)) then acc
      else
        val newAcc = if (nullable(r.deriv(input(from)))) from else acc
        stack.frames = (r, from, newAcc) :: stack.frames
        prefixFromMemI(r, from + 1, input, newAcc, stack)
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])
@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt =
  c.get(r, from, acc) match
    case Some(v) => v
    case None() =>
      if (from == input.size || lostCauseZipper(z)) then acc
      else
        val newAcc = if (nullable(r.deriv(input(from)))) from else acc
        stack.frames = (r, from, newAcc) :: stack.frames
        prefixFromMemI(r, from + 1, input, newAcc, stack)
```

Memoization

Tail recursive functions

```
@tailrec
def prefixFrom[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt): BigInt =
  if (from == input.size || lostCauseZipper(z)) then acc
  else
    val newAcc = if (nullable(r.deriv(input(from)))) from else acc
    prefixFrom(r, from + 1, input, newAcc)
```

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])
@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt =
  c.get(r, from, acc) match
    case Some(v) => v
    case None() =>
      if (from == input.size || lostCauseZipper(z)) then acc
      else
        val newAcc = if (nullable(r.deriv(input(from)))) from else acc
        stack.frames = (r, from, newAcc) :: stack.frames
        prefixFromMemI(r, from + 1, input, newAcc, stack)
```

All recursive calls return the same value

Memoization

Tail recursive functions

```
def prfxSFromMem[C](r: Regex[C], from: BigInt, input: List[C])(using cache: Cache[C]):
(List[C], List[C]) =
  val stack = new MutStack(Nil)
  val res = prfxSFromMemI(r, from, input, input.size, stack)
  stack.frames.foreach {
    case (r, from, lastNullablePos) => cache.update(r, from, lastNullablePos)
  }
```

Memoization

Tail recursive functions

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])

@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt = //...

def prfxSFromMem[C](r: Regex[C], from: BigInt, input: List[C])(using cache: Cache[C]):
(List[C], List[C]) =
  val stack = new MutStack(Nil)
  val res = prfxSFromMemI(r, from, input, input.size, stack)
  stack.frames.foreach {
    case (r, from, lastNullablePos) => cache.update(r, from, lastNullablePos)
  }
```

Memoization

Tail recursive functions

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])

@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt = //...

def prfxSFromMem[C](r: Regex[C], from: BigInt, input: List[C])(using cache: Cache[C]):
(List[C], List[C]) =
  val stack = new MutStack(Nil)
  val res = prfxSFromMemI(r, from, input, input.size, stack)
  stack.frames.foreach {
    case (r, from, lastNullablePos) => cache.update(r, from, lastNullablePos
  }
```

Memoization

Tail recursive functions

```
type MutableCache[C] = Map[(Regex[C], BigInt), BigInt]
case class MutStack[C](var frames: List[(Regex[C], BigInt, BigInt)])

@tailrec
def prefixFromMemI[C](r: Regex[C], from: BigInt, input: List[C], acc: BigInt)
    (using c: MutableCache[C], stack: MutStack[C]): BigInt = //...

def prfxSFromMem[C](r: Regex[C], from: BigInt, input: List[C])(using cache: Cache[C]):
(List[C], List[C]) =
  val stack = new MutStack(Nil)
  val res = prfxSFromMemI(r, from, input, input.size, stack)
  stack.frames.foreach {
    case (r, from, lastNullablePos) => cache.update(r, from, lastNullablePos
  }
```

Explicit simulation of JVM stack without stack overflows enabling memoization without breaking tail recursion

ZipLex Performance

Performance

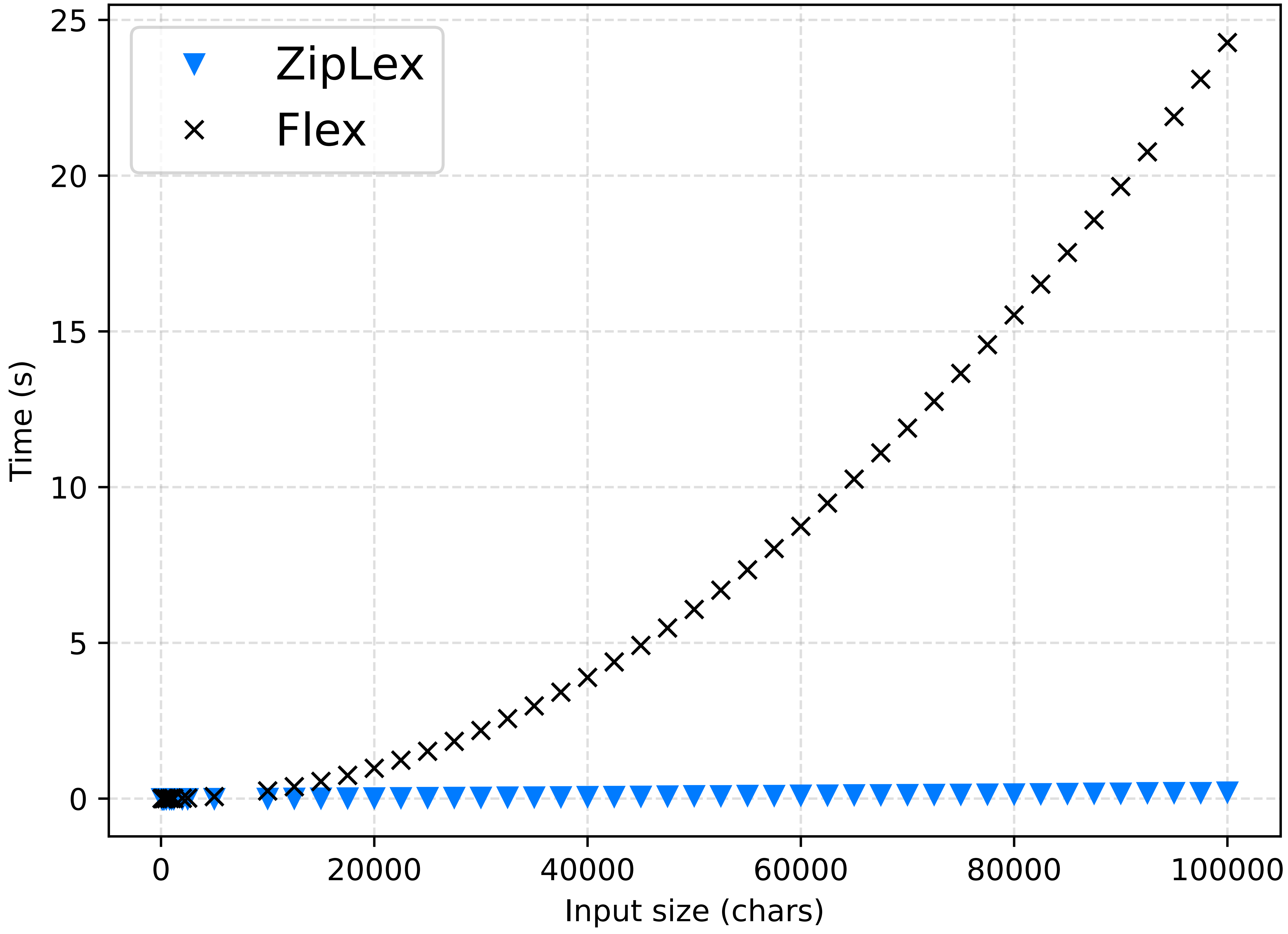
a and $a{*}b$ Benchmark

Input: “$aaa\ldots a$”

Performance

a and a^*b Benchmark

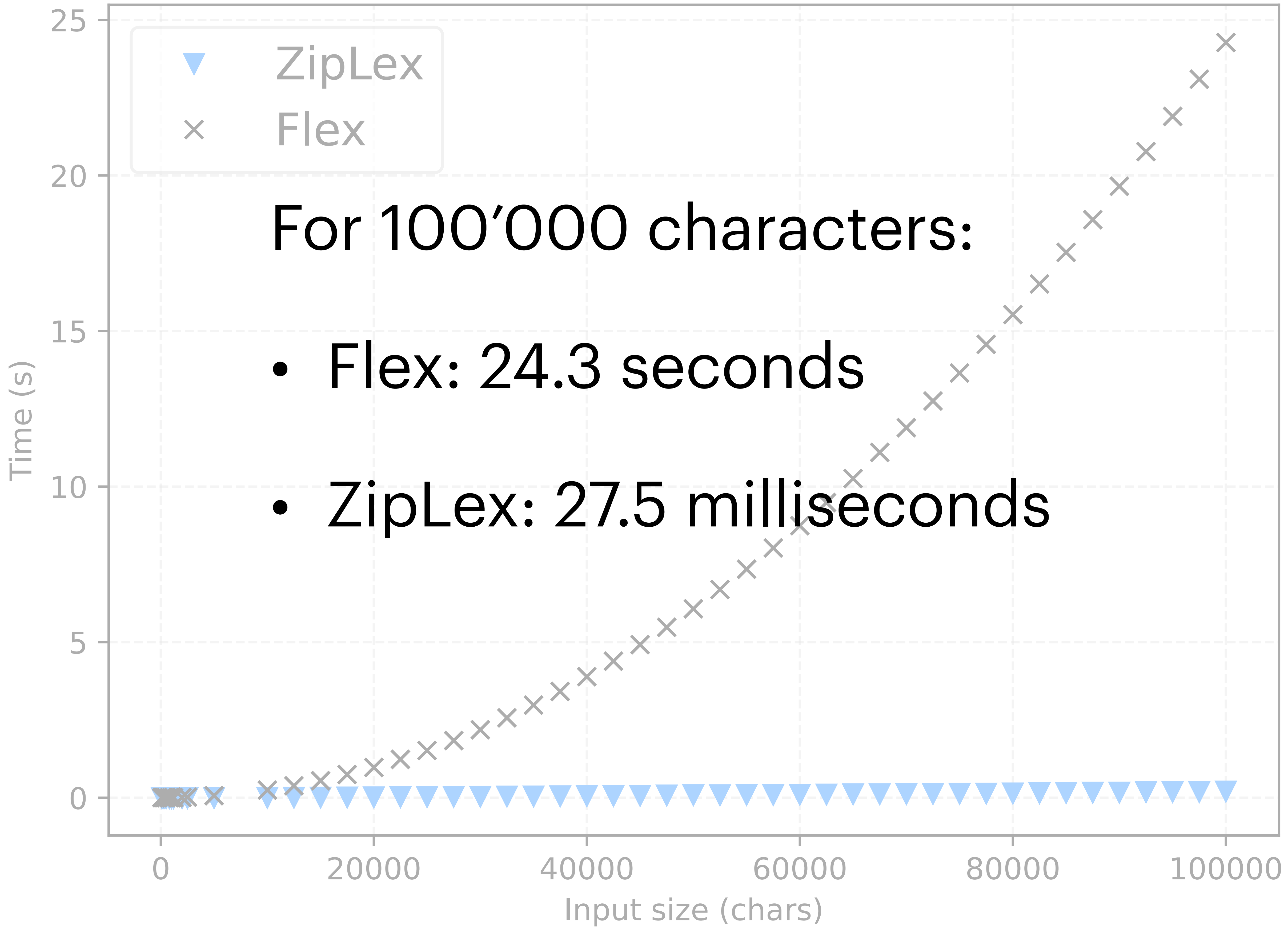
Input: "$aaa\ldots a$"



Performance

a and $a{*}b$ Benchmark

Input: "$aaa \ldots a$"



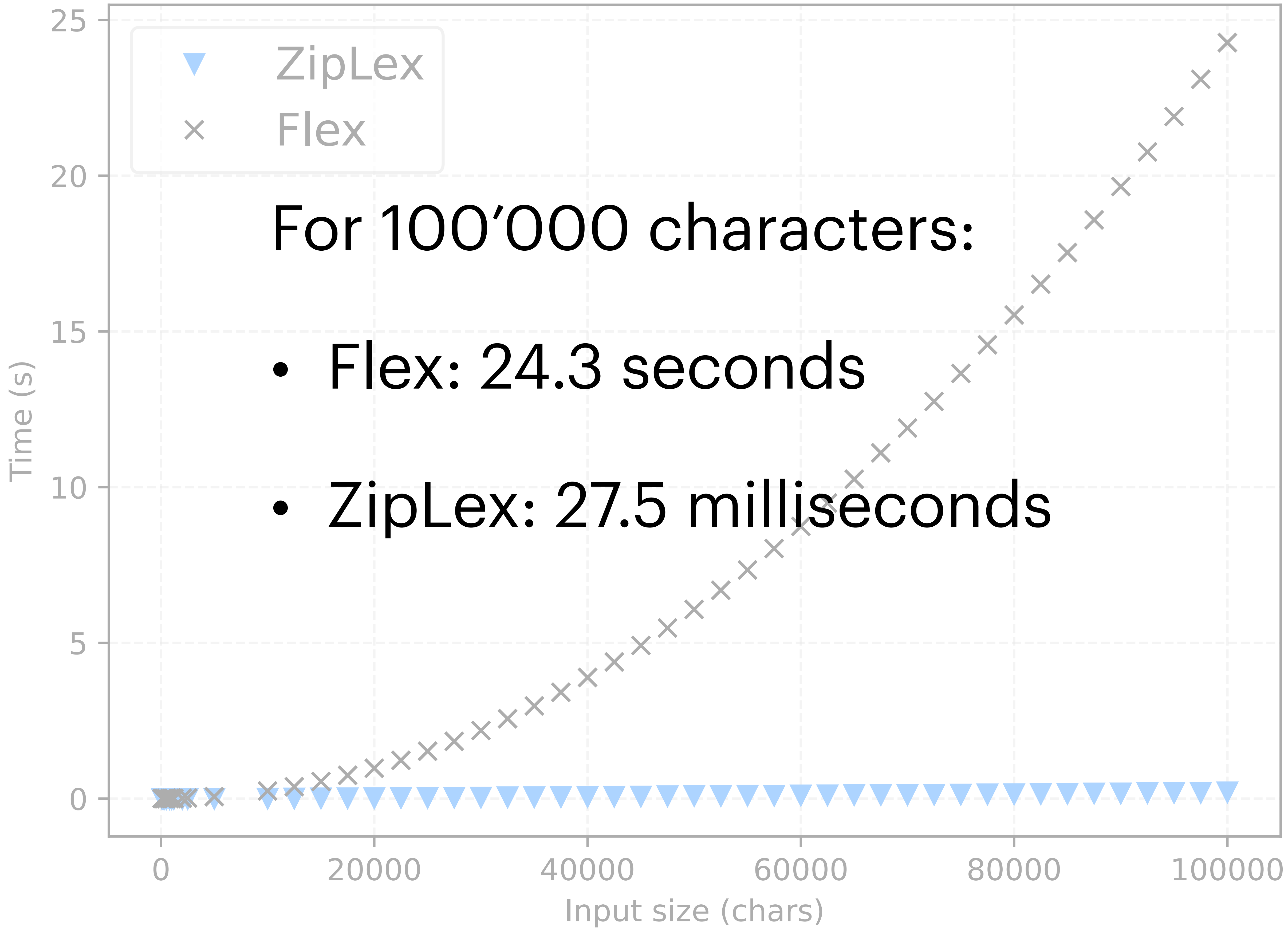
For 100’000 characters:

- Flex: 24.3 seconds
- ZipLex: 27.5 milliseconds

Performance

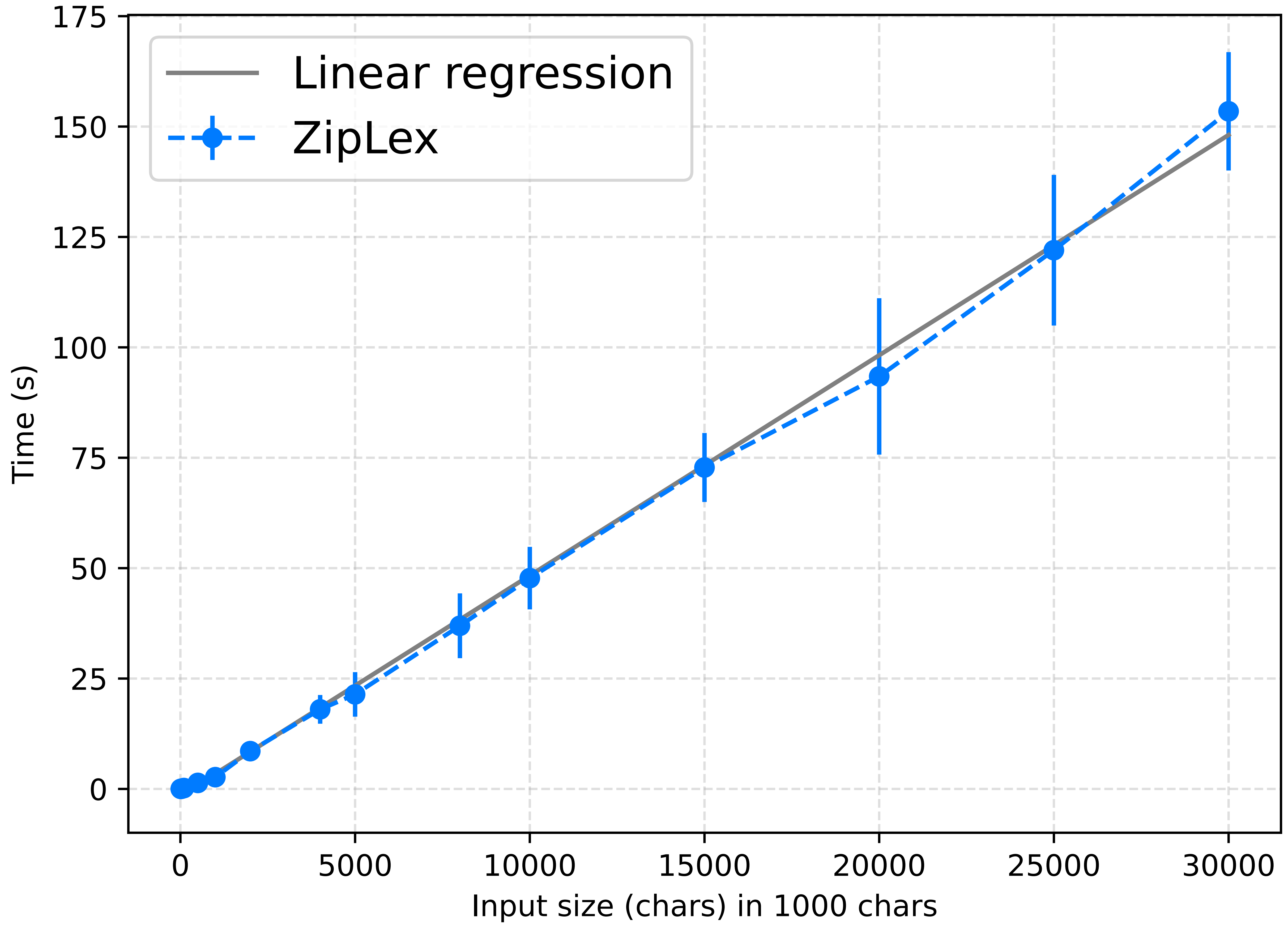
a and a^*b Benchmark

Input: “$aaa\ldots a$”



For 100’000 characters:

- Flex: 24.3 seconds
- ZipLex: 27.5 milliseconds



Conclusion

Conclusion

Mutability important for **efficient programs**

Conclusion

Mutability important for **efficient programs**

e.g., **memoization**

Conclusion

Mutability important for **efficient programs**

e.g., **memoization**

Expensive reasoning due to **aliasing**

Conclusion

Mutability important for **efficient programs**

e.g., **memoization**

Expensive reasoning due to **aliasing**

Introduced **cells** with atomic swaps **without introducing aliasing**

Conclusion

Mutability important for **efficient programs**

e.g., **memoization**

Expensive reasoning due to **aliasing**

Introduced **cells** with atomic swaps **without introducing aliasing**

Decorator DP for **modular** verification

Conclusion

Mutability important for **efficient programs**

e.g., **memoization**

Expensive reasoning due to **aliasing**

Introduced **cells** with atomic swaps **without introducing aliasing**

Decorator DP for **modular** verification

Mutable Hash Table for **verified memoization**

Conclusion

Mutability important for **efficient programs**

  e.g., **memoization**

Expensive reasoning due to **aliasing**

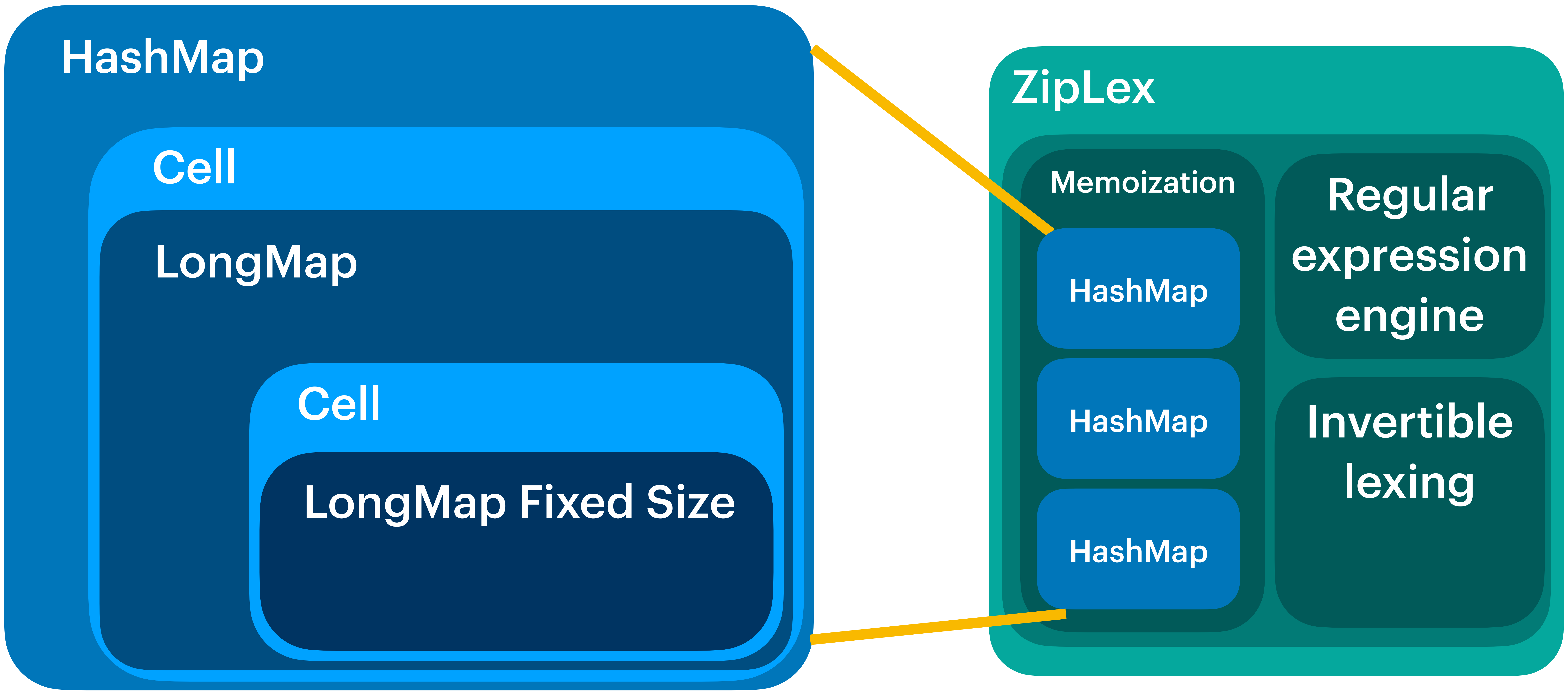
Introduced **cells** with atomic swaps **without introducing aliasing**

  Decorator DP for **modular** verification

Mutable Hash Table for **verified memoization**

Linear-time verified invertible lexing with ZipLex

Conclusion



Enabled by the Cell construct without introducing aliasing

samuel.chassot@epfl.ch