Algebraic Data Type

an essential concept
for safe and compositional come

Kim Sol

Interests

- **Type System**
- **Theoretical Backgrounds of Programming Languages**
- **Program Reasoning (Static Analysis, Software Verification, …)**

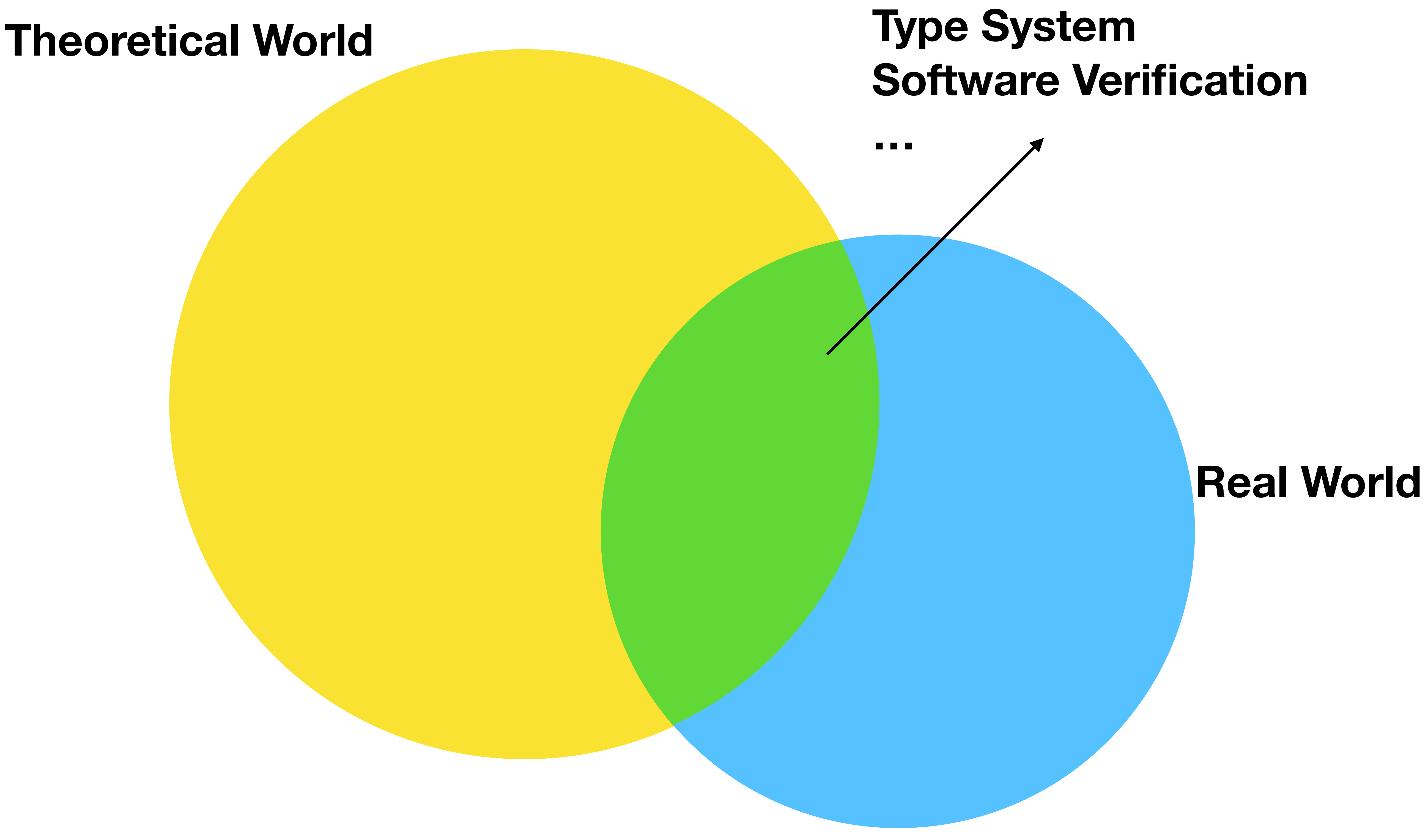
Interests

- **Type System**
- **Theoretical Backgrounds of Programming Languages**
- **Program Reasoning (Static Analysis, Software Verification, …)**

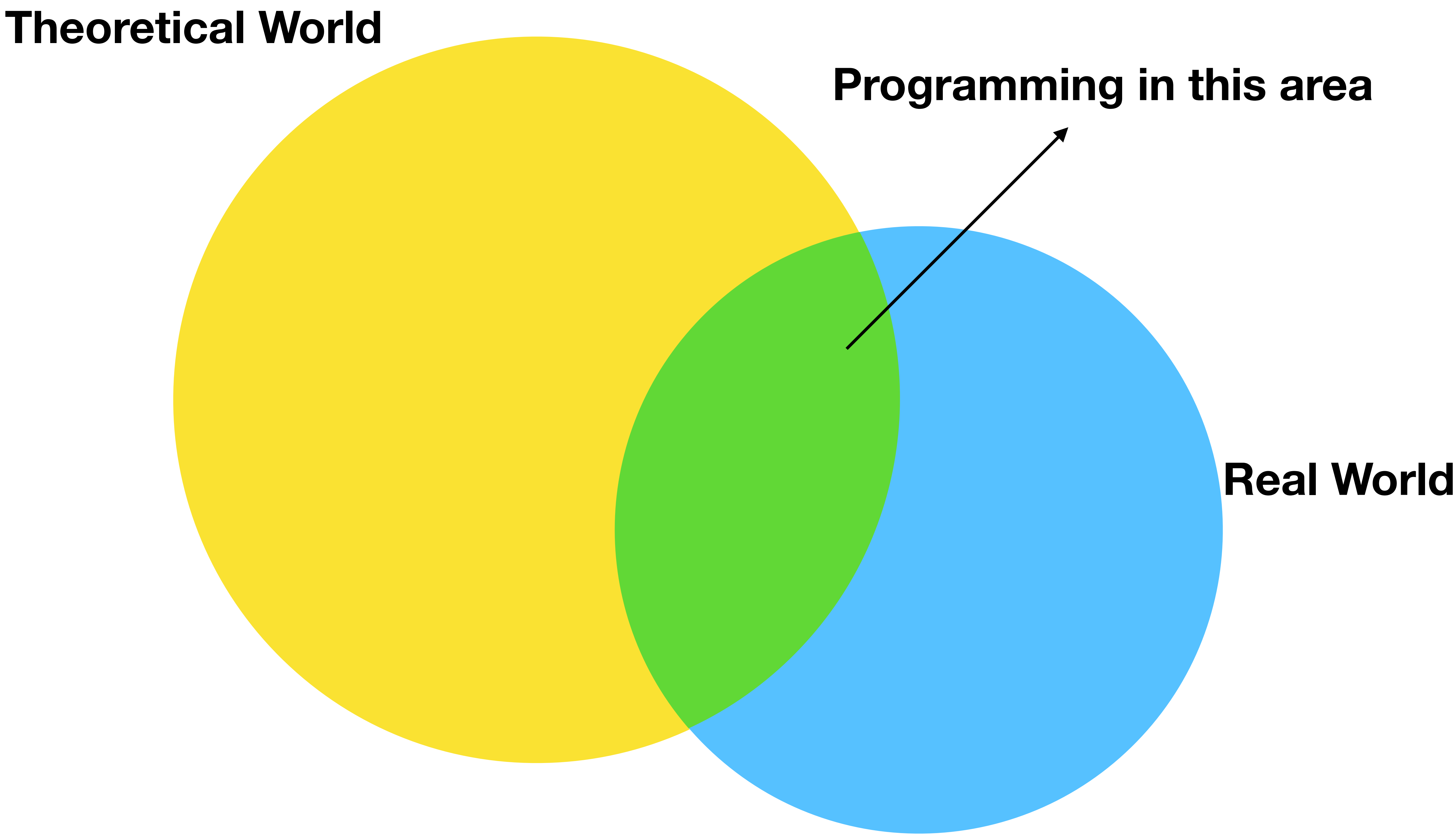
Design of ↓ ↑ **Implementation of**

Functional Programming

Interests


Theoretical World
Type System
Software Verification
...
Real World

Interests



Index

- Algebraic Data Type
- Functional Languages' Algebraic Data Type
- A Background of Algebraic Data Type
- Pros of Algebraic Data Type
- Advanced Data Types based on Algebraic Data Type
- Non-Functional Languages' Workarounds
- Modern Programming Languages' Algebraic Data Type

Algebraic Data Type

Hopefully, what you can get through this talk

If you don't know :	**ability to use a new wave,**
If you already know :	**advanced uses,**
	fundamental materials of the type,
	etc.

Algebraic Data Type

Product Type **&** **Sum Type**

Algebraic Data Type

Product Type **&** **Sum Type**

= cartesian product ***= separated sum***

Algebraic Data Type

Product Type **&** **Sum Type**
= *cartesian product* **= *separated sum***

i.e. product type $A \times B$

$$A \times B = \{(a, b) |\ a \in A, b \in B\}$$

Algebraic Data Type

Product Type **&** **Sum Type**

= *cartesian product* **= *separated sum***

i.e. sum type $A{+}B$

$$A{+}B = \{a|\ a \in A\} \bigcup \{b|\ b \in B\}$$

Algebraic Data Type

Product Type **&** **Sum Type**

= *cartesian product* **= *separated sum***

i.e. sum type $A+B$

$$A+B = \{a|\ a \in A\} \bigcup \{b|\ b \in B\}$$

A way to remember an origin of elements

$$A+B = \{(a,\underline{1})|\ a \in A\} \bigcup \{(b,\underline{2})|\ b \in B\}$$

ENCODING

Algebraic Data Type

Product Type **&** **Sum Type**

= cartesian product ***= separated sum***

Safe & Compositional

Functional Languages' Algebraic Data Type

option type
nat type
list type

Functional Languages' Algebraic Data Type

option type

```
type 'a option = None | Some of 'a

let map : ('a -> 'b) -> 'a option -> 'b option
= fun f o ->
  match o with
  | Some(a) -> Some(f a)
  | None -> None
```

Functional Languages' Algebraic Data Type

nat type

```
type nat = O | S of nat

let rec plus : nat -> nat -> nat
= fun n m ->
  match n with
  | S n' -> plus n' (S m)
  | O -> m
```

Functional Languages' Algebraic Data Type

list type

```
type 'a list = Nil | Cons of 'a * 'a list

let rec zip : 'a list -> 'b list -> ('a * 'b) list
= fun l1 l2 ->
  match l1, l2 with
  | Cons(a, l1'), Cons(b, l2') -> Cons (a, b) (zip l1' l2')
  | _, _ -> Nil
```

A Background of Algebraic Data Type

Correspondence between mathematical logic and types

Propositions as types

Proofs as programs

Simplification of proofs as evaluation of programs

A Background of Algebraic Data Type

Correspondence between mathematical logic and types

propositions	program types
$A \vee B$	$A + B$
$A \wedge B$	$A \times B$
$A \supset B$	$A \to B$
$\forall$	Π
$\exists$	Σ
...	...

A Background of Algebraic Data Type

Correspondence between mathematical logic and types

Simplifying a proof (Natural Deduction)

$$\dfrac{\dfrac{\dfrac{\dfrac{[B \,\&\, A]^z}{A}\,\&\text{-E}_2 \qquad \dfrac{[B \,\&\, A]^z}{B}\,\&\text{-E}_1}{A \,\&\, B}\,\&\text{-I}}{(B \,\&\, A) \supset (A \,\&\, B)}\,\supset\text{-I}^z \qquad \dfrac{B \qquad A}{B \,\&\, A}\,\&\text{-I}}{A \,\&\, B}\,\supset\text{-E}$$

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{B \qquad A}{B \,\&\, A}\,\&\text{-I}}{A}\,\&\text{-E}_2 \qquad \dfrac{\dfrac{B \qquad A}{B \,\&\, A}\,\&\text{-I}}{B}\,\&\text{-E}_1}{A \,\&\, B}\,\&\text{-I}$$

$$\Downarrow$$

$$\dfrac{A \qquad B}{A \,\&\, B}\,\&\text{-I}$$

Evaluating a program (Lambda Calculus)

$$\dfrac{\dfrac{\dfrac{\dfrac{[z : B \times A]^z}{\pi_2\, z : A}\,\times\text{-E}_2 \qquad \dfrac{[z : B \times A]^z}{\pi_1\, z : B}\,\times\text{-E}_1}{\langle \pi_2\, z, \pi_1\, z\rangle : A \times B}\,\times\text{-I}}{\lambda z.\, \langle \pi_2\, z, \pi_1\, z\rangle : (B \times A) \to (A \times B)}\,\to\text{-I}^z \qquad \dfrac{y : B \qquad x : A}{\langle y, x\rangle : B \times A}\,\times\text{-I}}{(\lambda z.\, \langle \pi_2\, z, \pi_1\, z\rangle)\, \langle y, x\rangle : A \times B}\,\to\text{-E}$$

$$\Downarrow$$

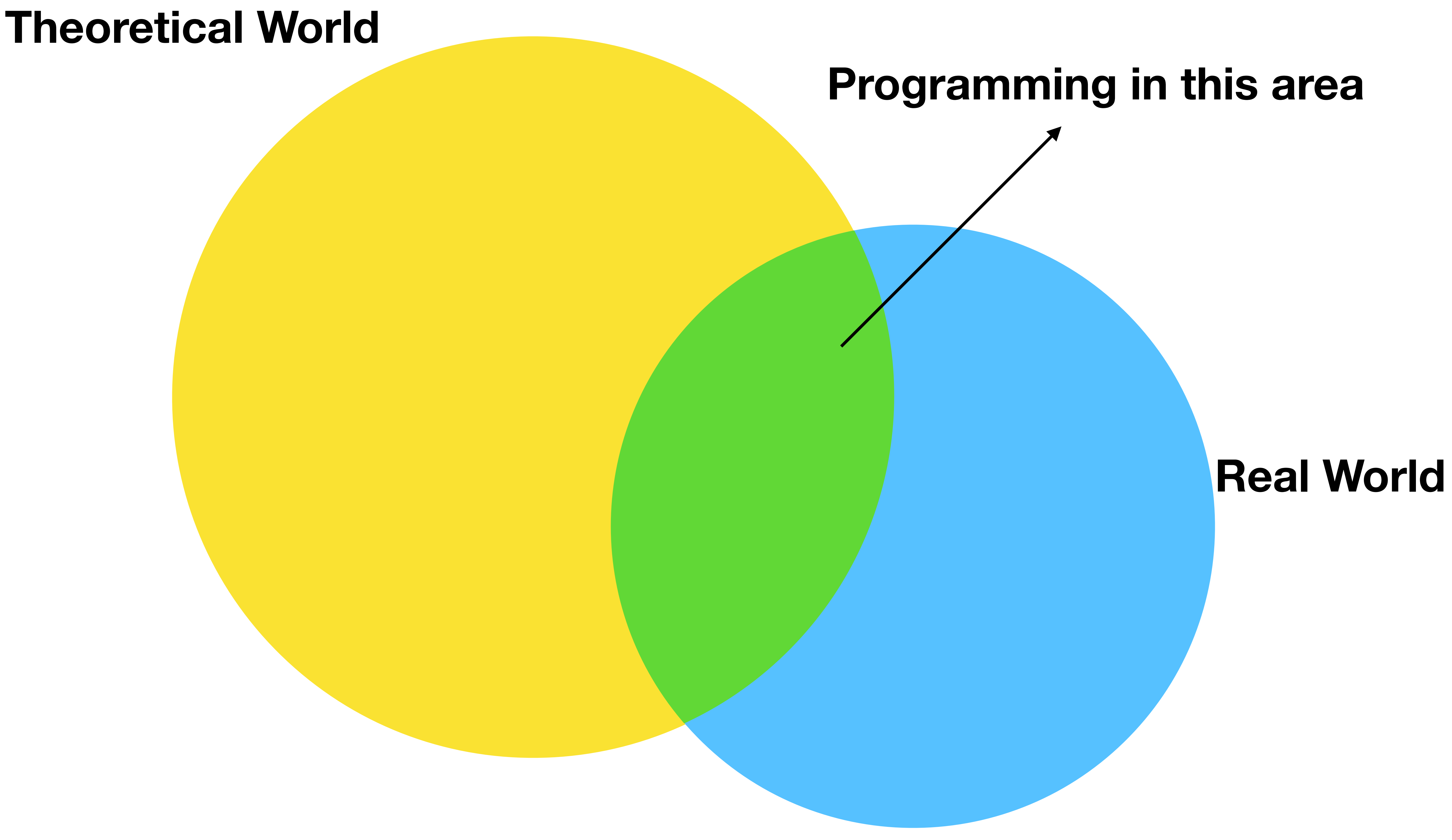
$$\dfrac{\dfrac{\dfrac{y : B \qquad x : A}{\langle y, x\rangle : B \times A}\,\times\text{-I}}{\pi_2\, \langle y, x\rangle : A}\,\times\text{-E}_2 \qquad \dfrac{\dfrac{y : B \qquad x : A}{\langle y, x\rangle : B \times A}\,\times\text{-I}}{\pi_1\, \langle y, x\rangle : B}\,\times\text{-E}_1}{\langle \pi_2\, \langle y, x\rangle, \pi_1\, \langle y, x\rangle\rangle : A \times B}\,\times\text{-I}$$

$$\Downarrow$$

$$\dfrac{x : A \qquad y : B}{\langle x, y\rangle : A \times B}\,\times\text{-I}$$

(2) Proposition as Types Philip Wadler. 2015.

Pros of Algebraic Data Type



Pros of Algebraic Data Type

Safe & Compositional

Pros of Algebraic Data Type

Safe (Exhaustiveness)

```
type other1 = E | F | G
type other2 = K | I
type either = Other1 of other1 | Other2 of other2
type complex_data_type = A | B | C
                         | Either of either
                         | Both of other1 * other2

let f : complex_data_type -> int
= fun data ->
  match data with
  | A -> 1
  | B -> 2
  | C -> 3
  | Both(E, K) -> 4
  | Both(F, I) -> 5
  | Either(Other1(E)) -> 6
```

Pros of Algebraic Data Type

Safe (Exhaustiveness)

```
skim:tmp/ $ ocaml exhaustiveness.ml
File "./exhaustiveness.ml", line 8, characters 2-117:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(Both (F, K)|Both (E, I)|Both (G, _)|Either (Other1 (F|
G))|Either (Other2 _))
```

Type system guarantees exhaustiveness of code

Pros of Algebraic Data Type

Compositional

```
type stmt = Skip
          | Seq of stmt * stmt
          | Assign of var * expr
          | If of expr * stmt * stmt
          | While of expr * stmt
and expr = I of int
         | B of bool
         | V of string
         | Add of expr * expr
         | Lt of expr * expr
and var = Var of string

let program = Seq(Assign(Var("i"), I(0)),
                  While(Lt(V("i"), I(10)),
                        Assign(Var("i"), Add(V("i"), I(1)))
                       )
                 )
```

Pros of Algebraic Data Type

Safe & Compositional (Design & Implementation)

```
type expr = I of int
          | B of bool
          | V of string
          | Add of expr * expr
          | Lt of expr * expr

let rec eval : expr -> mem -> 'a
= fun e m ->
  match e with
  | I(i) -> i
  | B(b) -> b
  | V(v) -> Memory.get v m
  | Add(e1, e2) -> eval e1 m + eval e2 m
  | Lt(e1, e2) -> eval e1 m < eval e2 m
```

Pros of Algebraic Data Type

Safe & Compositional (Design & Implementation)

```
type stmt = Skip
          | Seq of stmt * stmt
          | Assign of var * expr
          | If of expr * stmt * stmt
          | While of expr * stmt

let rec interpreter : stmt -> mem -> mem
= fun s m ->
  match s with
   | Skip -> m
   | Seq(s1, s2) -> interpreter s2 (interpreter s1 m)
   | Assign(Var(v), e) -> Memory.set v (eval e m) m
   | If(e, s1, s2) -> if (eval e m)
                      then interpreter s1 m
                      else interpreter s2 m
   | While(e, s) -> if (eval e m)
                    then interpreter (While(e, s, m)) m
                    else m
```

Pros of Algebraic Data Type

Encoding program executions to types

Pros of Algebraic Data Type

Lifting program executions to types

Pros of Algebraic Data Type

Lifting semantics to types

Pros of Algebraic Data Type

Lifting semantics to types

```
type 'a option = None | Some of ‘a
```

empty & non-empty

```
type nat = O | S of nat
```

zero & non zero

```
type 'a list = Nil | Cons of 'a * 'a list
```

empty list & non-empty list

Pros of Algebraic Data Type

Lifting semantics to types

```
type natural_number = E of even | O of odd
and even = Zero | Even of even
and odd = One | Odd of odd
```

even & odd

Pros of Algebraic Data Type

```
type 'a list = Nil | Cons of 'a * 'a list

let rec zip : 'a list -> 'b list -> ('a * 'b) list
= fun l1 l2 ->
  match l1, l2 with
  | Cons(a, l1'), Cons(b, l2') -> Cons (a, b) (zip l1' l2')
  | _, _ -> Nil
```

The specification of the function can be simply expressed without if-else statements

Advanced Data Types based on Algebraic Data Type

Generalized Algebraic Data Type (GADT, First-class Phantom Type)

Types that can depend on indexed types of ADT

Dependent Types

Types that can depend on terms

Advanced Data Types based on Algebraic Data Type

Generalized Algebraic Data Type (GADT, First-class Phantom Type)

Types that can depend on indexed types of ADT

Advanced Data Types based on Algebraic Data Type

Generalized Algebraic Data Type (GADT, First-class Phantom Type)

```
type stmt = Skip
          | Seq of stmt * stmt
          | Assign of var * expr
          | If of expr * stmt * stmt
          | While of expr * stmt
and expr = I of int
         | B of bool
         | V of string
         | Add of expr * expr
         | Lt of expr * expr
and var = Var of string
```

Advanced Data Types based on Algebraic Data Type

Generalized Algebraic Data Type (GADT, First-class Phantom Type)

```
type _ stmt = Skip
            | Seq : 'a stmt * 'b stmt -> 'b stmt
            | Assign : string var * 'a expr -> 'b stmt
            | If : bool expr * 'a expr * 'a expr -> 'a stmt
            | While : bool expr * 'a stmt -> 'b stmt
and _ expr = I : int -> int expr
           | B : bool -> bool expr
           | V : string -> 'a expr
           | Add : int expr * int expr -> int expr
           | Lt : int expr * int expr -> bool expr
and _ var = Var : string -> string var
```

Advanced Data Types based on Algebraic Data Type

Generalized Algebraic Data Type (GADT, First-class Phantom Type)

```
let program = Seq(Assign(Var("i"), I(0)),
                  While(Lt(B(true), I(10)),
                        Assign(Var("i"), Add(V("i"), I(1)))
                       )
                 )
```

Advanced Data Types based on Algebraic Data Type

Generalized Algebraic Data Type (GADT, First-class Phantom Type)

```
let program = Seq(Assign(Var("i"), I(0)),
                  While(Lt(B(true), I(10)),
                        Assign(Var("i"), Add(V("i"), I(1)))
                       )
                 )

type expr = I of int
          | B of bool
          | V of string
          | Add of expr * expr
          | Lt of expr * expr
```

can compile

Advanced Data Types based on Algebraic Data Type

Generalized Algebraic Data Type (GADT, First-class Phantom Type)

```
let program = Seq(Assign(Var("i"), I(0)),
                  While(Lt(B(true), I(10)),
                        Assign(Var("i"), Add(V("i"), I(1)))
                       )
                 )

type _ expr = I : int -> int expr
            | B : bool -> bool expr
            | V : string -> 'a expr
            | Add : int expr * int expr -> int expr
            | Lt : int expr * int expr -> bool expr
```

cannot compile

```
skim:tmp/ $ ocaml test.ml
File "./test.ml", line 20, characters 27-34:
Error: This expression has type bool expr
       but an expression was expected of type int expr
       Type bool is not compatible with type int
```

Advanced Data Types based on Algebraic Data Type

Generalized Algebraic Data Type (GADT, First-class Phantom Type)

Can express more general specifications using a type system

Advanced Data Types based on Algebraic Data Type

Dependent Types

Types that can depend on terms

Advanced Data Types based on Algebraic Data Type

Dependent Types

What if?

```
type _ integer = Zero : {n:nat | n = 0} integer
               | Postive : {n:nat | n > 0} integer
               | Negative : {n:nat | n < 0} integer
```

Advanced Data Types based on Algebraic Data Type

Dependent Types, (Types depend on terms)

```
Inductive nat :=
| O : nat
| S : nat -> nat.

Inductive list (A : Type) : Type :=
| Nil : list A
| Cons : A -> list A -> list A.

Inductive ilist (A : Type) : nat -> Type :=
| INil : ilist A O
| ICons : forall n, A -> ilist A n -> ilist A (S n).

Definition hd n (A : Type) (ls : ilist A (S n)) := …

Fixpoint concat (A : Type)
                (n : nat) (ls1 : ilist A n)
                (m : nat) (ls2 : ilist A m)
                : ilist A (m + m) := …
```

Non-Functional Languages’ Workarounds

Can mimic Algebraic Data Type

but

1. Does not type safe
2. Depends on developer’s level of understanding

Non-Functional Languages' Workarounds

interface & implementation pattern in Java

```
import java.util.Optional;
interface SumTypeOfAB {}
class A implements SumTypeOfAB {}
class B implements SumTypeOfAB {}

…
    public Optional<Integer> func(SumTypeOfAB data) {
        if (data instanceof A) {
            return Optional.of(1);
        } else if (data instanceof B) {
            return Optional.of(2);
        } else {
            return Optional.empty();
        }
    }
…
```

Non-Functional Languages' Workarounds

interface & implementation pattern in Java

```
import java.util.Optional;
interface SumTypeOfAB {}
class A implements SumTypeOfAB {}
class B implements SumTypeOfAB {}

…
    public Optional<Integer> func(SumTypeOfAB data) {
        if (data instanceof A) {
            return Optional.of(1);
        } else if (data instanceof B) {
            return Optional.of(2);
        } else {
            return Optional.empty();
        }
    }
…
```

1. *runtime type check*
2. *can't guarantee exhaustiveness*

Non-Functional Languages' Workarounds

optional in Java

```
package java.util;

public final class Optional<T> {
    private static final Optional<?> EMPTY = new Optional<>();
    private final T value;
    private Optional() {
        this.value = null;
    }
    public static<T> Optional<T> empty() {
        @SuppressWarnings("unchecked")
        Optional<T> t = (Optional<T>) EMPTY;
        return t;
    }
    private Optional(T value) {
        this.value = Objects.requireNonNull(value);
    }
    public static <T> Optional<T> of(T value) {
        return new Optional<>(value);
    }
    public static <T> Optional<T> ofNullable(T value) {
        return value == null ? empty() : of(value);
    }

    …
```

Non-Functional Languages' Workarounds

optional in Java

```
    public T get() {
        if (value == null) {
            throw new NoSuchElementException("No value present");
        }
        return value;
    }
    public boolean isPresent() {
        return value != null;
    }
    public void ifPresent(Consumer<? super T> consumer) {
        if (value != null)
            consumer.accept(value);
    }
    public T orElse(T other) {
        return value != null ? value : other;
    }
    public T orElseGet(Supplier<? extends T> other) {
        return value != null ? value : other.get();
    }
    public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)
throws X {
        if (value != null) {
            return value;
        } else {
            throw exceptionSupplier.get();
        }
    }
```

Non-Functional Languages' Workarounds

optional in Java

```java
public Optional<T> filter(Predicate<? super T> predicate) {
    Objects.requireNonNull(predicate);
    if (!isPresent())
        return this;
    else
        return predicate.test(value) ? this : empty();
}
public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent())
        return empty();
    else {
        return Optional.ofNullable(mapper.apply(value));
    }
}
public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent())
        return empty();
    else {
        return Objects.requireNonNull(mapper.apply(value));
    }
}
```

Non-Functional Languages’ Workarounds

optional in Java

Monad, Monoid, Functor, Endofunctor, …

🤔

Hmteresting…

But, it depends on developers’ level of understanding

Non-Functional Languages’ Workarounds

GADT in OOP

Generalized algebraic data types and object-oriented programming
Andrew Kennedy and Claudio V Russo. 2005.
[link]

Still, hard to avoid *runtime-checked cast*

Non-Functional Languages' Workarounds

GADT in OOP

```
interface Expr { Object eval(); }
class I implements Expr {
    Integer v;
    public I(Integer v) { this.v = v; }
    public Integer eval() { return this.v; }
}
class B implements Expr {
    Boolean b;
    public B(Boolean b) { this.b = b; }
    public Boolean eval() { return this.b; }
}
class Add implements Expr {
    Expr v1;
    Expr v2;
    public Add(Expr v1, Expr v2) { this.v1 = v1; this.v2 = v2; }
    public Integer eval() { return (Integer) v1.eval() + (Integer) v2.eval(); }
}
class And implements Expr {
    Expr b1;
    Expr b2;
    public And(Expr b1, Expr b2) { this.b1 = b1; this.b2 = b2; }
    public Boolean eval() { return (Boolean) b1.eval() && (Boolean) b2.eval(); }
}
```

Non-Functional Languages' Workarounds

GADT in OOP

```
public class Example {
    public static void main(String[] args) {
        (new Add(new Add(new I(13), new I(16)), new I(5))).eval();
        (new Add(new Add(new I(13), new I(16)), new B(false))).eval();
    }
}
```

```
kstreee:tmp/ $ javac Example.java
kstreee:tmp/ $ java Example
Exception in thread "main" java.lang.ClassCastException:
java.lang.Boolean cannot be cast to java.lang.Integer
        at Add.eval(Example.java:19)
        at Example.main(Example.java:74)
```

Non-Functional Languages' Workarounds

GADT in OOP

```
interface Expr<T> { T eval(); }
class I implements Expr<Integer> {
    Integer v;
    public I(Integer v) { this.v = v; }
    public Integer eval() { return this.v; }
}
class B implements Expr<Boolean> {
    Boolean b;
    public B(Boolean b) { this.b = b; }
    public Boolean eval() { return this.b; }
}
class Add implements Expr<Integer> {
    Expr<Integer> v1;
    Expr<Integer> v2;
    public Add(Expr<Integer> v1, Expr<Integer> v2) { this.v1 = v1; this.v2 = v2; }
    public Integer eval() { return v1.eval() + v2.eval(); }
}
class And implements Expr<Boolean> {
    Expr<Boolean> b1;
    Expr<Boolean> b2;
    public And(Expr<Boolean> b1, Expr<Boolean> b2) { this.b1 = b1; this.b2 = b2; }
    public Boolean eval() { return b1.eval() && b2.eval();}
}
```

Non-Functional Languages' Workarounds

GADT in OOP

```
public class Example {
    public static void main(String[] args) {
        (new Add(new Add(new I(13), new I(16)), new I(5))).eval();
        (new Add(new Add(new I(13), new I(16)), new B(false))).eval();
    }
}
```

```
kstreee:tmp/ $ javac Example.java
Example.java:74: error: incompatible types: B cannot be converted to Expr<Integer>
        (new Add(new Add(new I(13), new I(16)), new B(false))).eval();
```

Non-Functional Languages’ Workarounds

GADT in OOP

Still, hard to avoid *runtime-checked cast*

Non-Functional Languages’ Workarounds

Instead of using a general way, uses complex ways

- Object-Orient Programming Design Patterns
- Complex interface implementation hierarchy
- Developers’ hands

Modern Programming Languages’ Algebraic Data Type

Support Algebraic Data Type as a Primitive Data Type
i.e. Scala (Dotty), Rust, Swift, …

Modern Programming Languages

Scala (Dotty)

```scala
case class A()
case class B()

def exampleFun(data: A | B): Boolean | Int = {
  data match {
    case (a: A) => true
    case (b: B) => 1
  }
}
```

Modern Programming Languages

Scala (Dotty)

```scala
case class A()
case class B()

type X = A | B
type Y = Boolean | Int
def exampleFun(data: X): Y = {
  data match {
    case (a: A) => true
    case (b: B) => 1
  }
}
```

Modern Programming Languages

Rust

```rust
enum X {
    A { weight: f32 },
    B { weight: f32, height: f32 }
}

fn example_func(x: X) -> f32 {
    match x {
        X::A { weight, .. } |
        X::B { weight, .. } => weight
    }
}
```

Modern Programming Languages

Swift

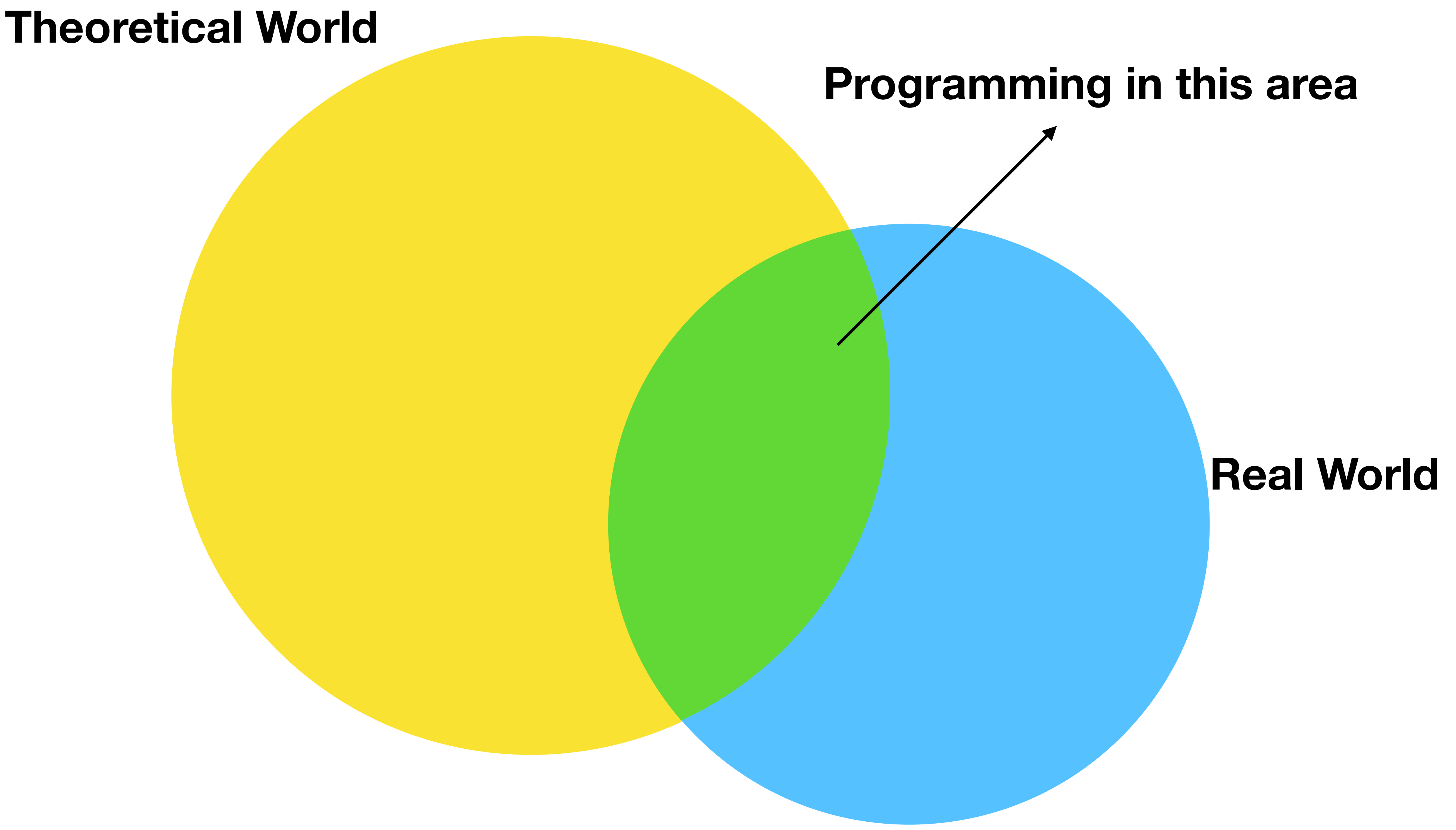
```swift
enum X {
    case a(Bool)
    case b(Int)
}

func exampleFunc(x: X) {
    switch x {
        case .a(let b):
            // ...
        case .b(let i):
            // ...
    }
}
```

Algebraic Data Type

Hopefully, what you can get through this talk

If you don't know :	**ability to use a new wave,**
If you already know :	**advanced uses,**
	fundamental materials of the type,
	etc.

In a slide



References

(1) **Algebraic Data Type: an essential concept for safe and compositional code**
Sol Kim. 2017.
[link]

(2) **Proposition as Types**
Philip Wadler. 2015.
[link]

(3) **Generalized algebraic data types and object-oriented programming**
Andrew Kennedy and Claudio V Russo. 2005.
[link]

(4) **Certified Programming with Dependent Types**
Adam Chilpala. 2011.
[link]

Q&A