ALGEBRAIC DATA TYPES (ADT)

Malte Neuss

ALGEBRA

Numbers:

```
2 + 2 + 2 =  3 * 2
```

ALGEBRA

Numbers:

```
2 + 2 + 2 =  3 * 2
```

Types:

```
class User
  verified: Bool
  email:    String
```

CONTENT

CONTENT

- Product Type

CONTENT

- Product Type
- Sum Type

CONTENT

- Product Type
- Sum Type
- Examples

CONTENT

- Product Type
- Sum Type
- Examples

Make illegal state unrepresentable

BASIC TYPES

```
type Void:                          // 0
type Unit: unit                     // 1
type Bool: true, false              // 2
...
type String: "", "a", "b" ...       // infty
```

PRODUCT TYPE

```
type ProductType = Type x Type
```

PRODUCT TYPE

```
type ProductType = Type x Type
```

```
type User = Bool x String
```

PRODUCT TYPE

```
type ProductType = Type x Type
```

```
type User = Bool x String
```

```
class User
  verified: Bool
  email:    String
```

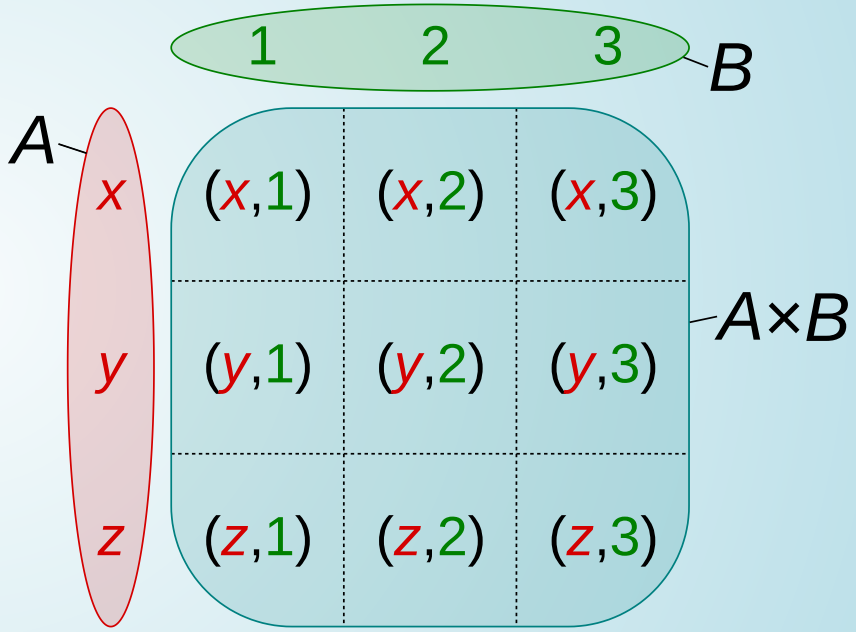
PRODUCT TYPE

Type:

```
type User = Bool x String
```

Values:

```
User(true,  "no@reply.com")
User(false, "no@reply.com")
User(true,  "ok@reply.com")
User(false, "ok@reply.com")
...
```



ALGEBRA

```
Bool x Bool: (true,true) (false,false) (true,false) (false,true)
   2 x 2   =  4
```

ALGEBRA

```
Bool x Bool: (true,true) (false,false) (true,false) (false,true)
   2 x 2   =  4
```

```
Bool x Unit: (true,unit) (false,unit)    ~ Boolean
   2 x 1   =  2
```

ALGEBRA

```
Bool x Bool: (true,true) (false,false) (true,false) (false,true)
   2 x 2   =  4
```

```
Bool x Unit: (true,unit) (false,unit)    ~ Boolean
   2 x 1   =  2
```

```
Bool x Void: (true,???)                  ~ Void
   2 x 0   =  0
```

SUM TYPE

```
type SumType = Type + Type
```

SUM TYPE

```
type SumType = Type + Type
```

```
type ID = Int + String
```

SUM TYPE

```
type SumType = Type + Type
```

```
type ID = Int + String
```

```
type ID = Int | String                                    Typescript
```

SUM TYPE

```
type SumType = Type + Type
```

```
type ID = Int + String
```

```
type ID = Int | String                                  Typescript
```

```
    ID = Union[int, str]                                Python
```

SUM TYPE

```
type SumType = Type + Type
```

```
type ID = Int + String
```

```
type ID = Int | String                                  Typescript
```

```
     ID = Union[int, str]                               Python
```

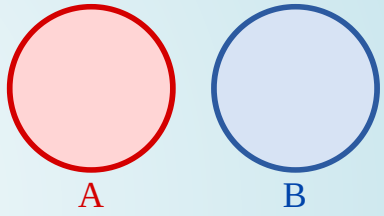
```
sealed trait ID                                         Scala

case class IntID(Int)       extends ID
case class StringID(String) extends ID
```

SUM TYPE

Type:

```
type ID = Int | String
```



By Stephan Kulla (User:Stephan Kulla) - Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=14978640

Values:

```
myID: ID = 1
        ✓
myID: ID = "123e4567-e89b..."
        ✓
...
```

ALGEBRA

```
Bool | Unit: true, false, unit
   2 + 1   =  3
```

ALGEBRA

```
Bool | Unit: true, false, unit
   2 + 1   =  3
```

```
Bool | Void: true, false          ~ Bool
   2 + 0   =  2                     2
```

ALGEBRA

```
Bool | Unit: true, false, unit
   2 + 1   =  3
```

```
Bool | Void: true, false          ~ Bool
   2 + 0   =  2                     2
```

```
Bool | Int:  true, false, 1, 2, 3, ...
   2 + 2^64
```

UNIT TYPE

```
type BoolOpt      = Bool | Unit

type Optional[T] =   T   | Unit
```

UNIT TYPE

```
type BoolOpt      = Bool | Unit

type Optional[T] =   T   | Unit
```

```
type Optional[T] =    T | undefined                Typescript
```

UNIT TYPE

```
type BoolOpt     = Bool | Unit

type Optional[T] =   T  | Unit
```

```
type Optional[T] =    T | undefined                Typescript
```

```
     Optional[T] = Union[T, NoneType]              Python
```

UNIT TYPE

```
type BoolOpt     = Bool | Unit

type Optional[T] =   T  | Unit
```

```
type Optional[T] =    T | undefined                Typescript
```

```
     Optional[T] = Union[T, NoneType]              Python
```

```
sealed trait Option[A]                             Scala

case class  Some[A] extends Option[A]
case object None    extends Option[A]
```

EXAMPLES

Modelling Data

Make illegal state unrepresentable

NO ADT: ERROR CODES

```
class UserResult:
  error_code: int              # 0 means ok
  user:       User             # contains dummy data on error

def fetchUser() -> UserResult:
  # network call
```

NO ADT: ERROR CODES

```
class UserResult:
  error_code: int              # 0 means ok
  user:       User             # contains dummy data on error

def fetchUser() -> UserResult:
  # network call
```

```
myUser = fetchUser()

if myUser.error_code == 0:
  # do sth.
  # what if myUser.user still has dummy data?
else
  # do fallback
```

NO ADT: ERROR CODES

```
class UserResult:
  error_code: int              # 0 means ok
  user:       User             # contains dummy data on error
```

NO ADT: ERROR CODES

```
class UserResult:
  error_code: int              # 0 means ok
  user:       User             # contains dummy data on error
```

Representable valid values:

```
UserResult(0, User("John"))
UserResult(0, User("Jane"))
UserResult(1, User("dummy"))
...
```

NO ADT: ERROR CODES

```
class UserResult:
  error_code: int              # 0 means ok
  user:       User             # contains dummy data on error
```

Representable valid values:

```
UserResult(0, User("John"))
UserResult(0, User("Jane"))
UserResult(1, User("dummy"))
...
```

Representable invalid values:

```
UserResult(1, User("John"))
UserResult(0, User("dummy"))
```

NO ADT: EXCEPTIONS

```
def fetchUser() -> User:
    # raise Exception on error
```

NO ADT: EXCEPTIONS

```
def fetchUser() -> User:
    # raise Exception on error
```

```
myUser = fetchUser()
# do sth.
```

NO ADT: EXCEPTIONS

```
def fetchUser() -> User:
    # raise Exception on error
```

```
myUser = fetchUser()
# do sth.
```

```
try:
  myUser = fetchUser()
  # do sth.
except ...
```

ADT: OPTIONAL

```
type Optional[T] = NoneType | T

def fetchUser() -> Optional[User]:
    # return None on error
```

ADT: OPTIONAL

```
type Optional[T] = NoneType | T

def fetchUser() -> Optional[User]:
    # return None on error
```

```
myUser = fetchUser()

if   isinstance(myUser, User):
  # do sth.
elif isinstance(myUser, NoneType):
  # do fallback
```

ADT: EITHER

```
type Either[E,T] = E | T

def fetchUser() -> Either[Error, User]:
    # return Error value on error
```

ADT: EITHER

```
type Either[E,T] = E | T

def fetchUser() -> Either[Error, User]:
    # return Error value on error
```

```
myUser = fetchUser()

if   isinstance(myUser, User):
  # do sth.
elif isinstance(myUser, Error):
  # analyze reason
```

ADT: CUSTOM

```
type User = Anonymous | LoggedIn

def fetchUser() -> User:
```

ADT: CUSTOM

```
type User = Anonymous | LoggedIn

def fetchUser() -> User:
```

```
myUser = fetchUser()

if   isinstance(myUser, Anonymous):
  # do anonymous stuff.
elif isinstance(myUser, LoggedIn):
  # do logged-in stuff.
```

ADT: CUSTOM EXTENDED

```
type User = Anonymous | LoggedIn | Admin
                                  # !new!
def fetchUser() -> User:
```

ADT: CUSTOM EXTENDED

```
type User = Anonymous | LoggedIn | Admin
                                  # !new!
def fetchUser() -> User:
```

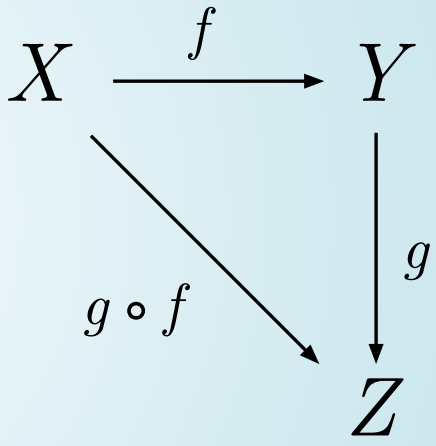
```
myUser = fetchUser()

if   isinstance(myUser, Anonymous):
  # do anonymous stuff.
elif isinstance(myUser, LoggedIn):
  # do logged-in stuff.
!!! Compiler error: forgot Admin case !!!
```

FURTHER TOPICS

- Scala
- Haskell
- Rust
- Dependent Types
- Linear Types

Category Theory



- Functor (map)
- Monad (flatMap)
- ...

Thanks