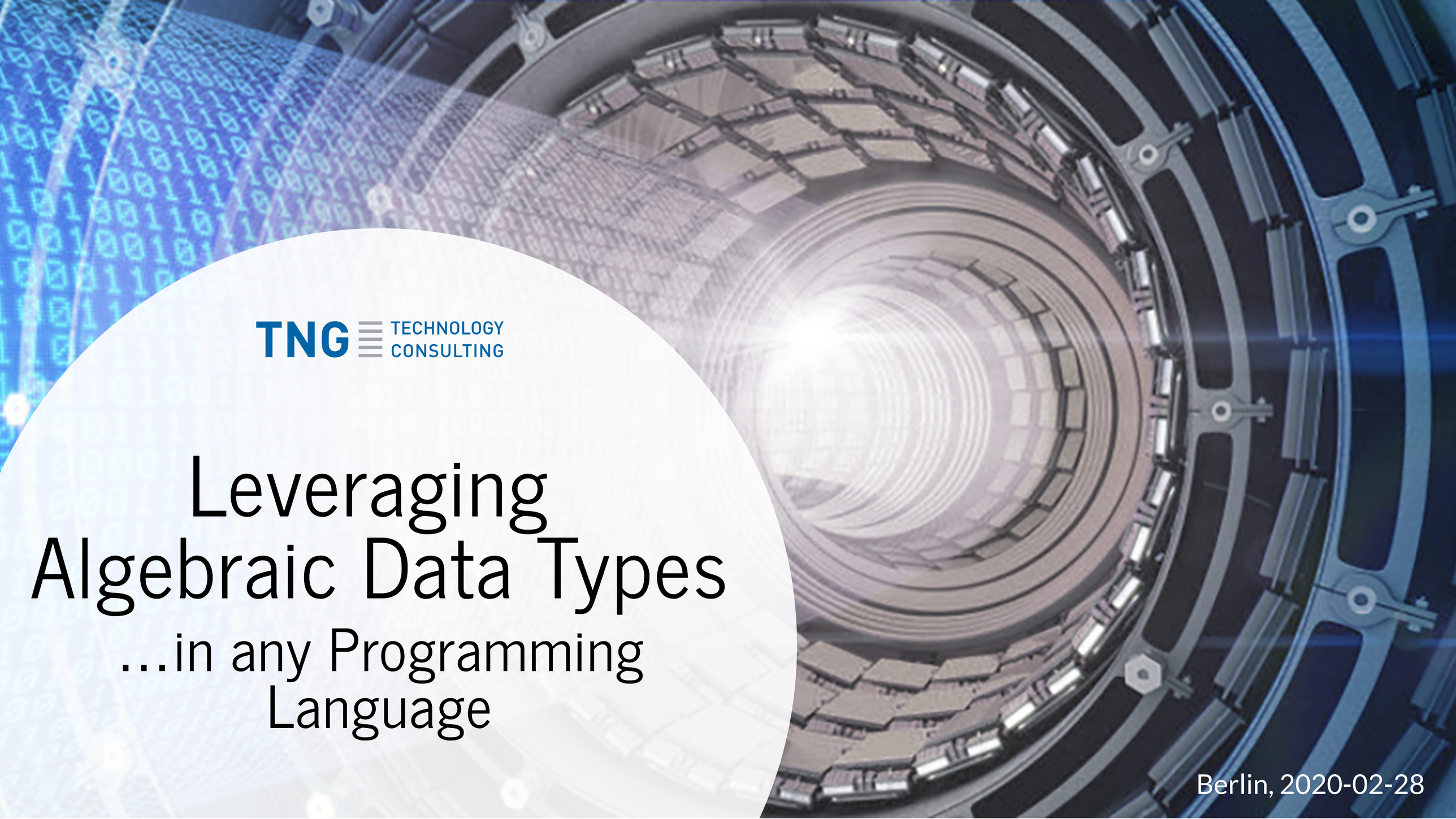


TNG TECHNOLOGY CONSULTING

Leveraging Algebraic Data Types

...in any Programming Language

Berlin, 2020-02-28

Algebraic Data Types

```haskell
data Bool = True | False
```

```haskell
data Maybe a = Nothing | Just a
-- Same as Optional<T> with Absent and Present
```

```haskell
data Pair a b = Pair a b
-- Record syntax: data Pair a b = Pair { first :: a, second :: b }
```

```haskell
data List a = Nil | Cons a (List a)
```

OOP data types

OOP classes have four purposes:

- Define a type
- Define a data structure
- Define operations
- Delimit a scope for all of these

```
class Pair<A, B> { // the type

    // Associated data structure (scoped)
    private A first;
    private B second;

    public Pair<A, B> (A first, B second) {
        this.first = first;
        this.second = second;

    // Some operations on the type
    public A first() {
        return first;
    }

    public B second () {
        return second;
    }

    public Pair<B, A> flip() {
        return new Pair<>(second, first);
    }
}
```

Algebraic Data Types

ADTs take a different approach, distinguishing between these four:

```haskell
data Pair a b  -- The type (no data)
    = Pair a b   -- The data structure
                 -- (associated to the type)


-- Operations associated with the type
-- decompose the structure via pattern matching
first :: Pair a b → a
first (Pair a b) = a

second :: Pair a b → b
second (Pair a b) = b

flip :: Pair a b → Pair b a
flip (Pair a b) = Pair b a

-- (Scoping is independent from the type,
-- done by the module system)
```

Why would I want to use ADTs?

- Decoupling operations from types
- Decoupling data from types
- Separation between data and operations
- Rigid type system allows strong guarantees (no casts necessary).
- Operations are extensible

Pattern Matching

```haskell
ifThenElse booleanExpr thenExpr elseExpr = case booleanExpr of
    True  -> thenExpr
    False -> elseExpr
```

```haskell
orElse :: Maybe a -> a -> a
orElse maybeThing fallback = case maybeThing of
    Nothing -> fallback
    Just a  -> a
```

Encoding **Maybe** in TypeScript

```
interface Maybe<T> {
  match<R>(matchNothing: () ⇒ R, matchJust: (thing: T) ⇒ R);
}

function nothing<T>(): Maybe<T> {
  return new (class extends Maybe<T> {
    match(matchNothing, matchJust) { return matchNothing(); }
  })();
}

function just<T>(thing: T): Maybe<T> {
  return new (class extends Maybe<T> {
    match(matchNothing, matchJust) { return matchJust(thing); }
  })();
}
```

```
class Maybe<T> {
  function orElse(fallback: T) {
    return maybeThing.match(
      () → fallback,
      thing → thing);
  }
}
```

Scott Encoding

Idea: Define a data type through *decomposition* rather than constructors.

Constructors:

```haskell
data Maybe a
    = Nothing
    | Just a
```

```haskell
Nothing ::      Maybe a
Just    :: a -> Maybe a
```

Decomposition:

```haskell
maybe
    :: r            -- matches the `Nothing` case
    -> (a -> r)     -- matches the `Just` case
    -> Maybe a
    -> r
maybe nothing just maybeThing = case maybeThing of
    Nothing -> nothing
    Just a -> just a
```

Duality:

```haskell
maybe Nothing Just ≡ id :: Maybe a -> Maybe a
```

```haskell
data Maybe a
    = Nothing   -- Nothing ::       Maybe a
    | Just a    -- Just    :: a -> Maybe a
```

```haskell
maybe :: r -> (a -> r) -> Maybe a -> r
maybe nothing just maybeThing = case maybeThing of
    Nothing -> nothing
    Just a  -> just a
-- maybe Nothing Just ≡ id :: Maybe a -> Maybe a
```

```haskell
data Bool
    = True    -- True  :: Bool
    | False   -- False :: Bool
```

```haskell
bool :: r -> r -> Bool -> r
bool true false boolExpr = case boolExpr of
    True  -> true
    False -> false
-- bool True False ≡ id :: Bool -> Bool
```

```haskell
data Pair a b
    = Pair a b    -- Pair :: a -> b -> Pair a b
```

```haskell
pairS :: (a -> b -> r) -> Pair a b -> r
pairS pair (Pair a b) = pair a b
-- pairS Pair ≡ id :: Pair a b -> Pair a b
```

```haskell
data List a
    = Nil                 -- Nil  ::                 L
    | Cons a (List a)     -- Cons :: a -> List a -> L
```

```haskell
listS :: r -> (a -> List a -> r) -> List a -> r
listS nil cons list = case list of
    Nil       -> nil
    Cons x xs -> cons x xs
-- listS Nil Cons ≡ id :: List a -> List a
```

Reversing ADTs

- Normal ADTs: Decomposition is defined via constructors.
- Scott encoded ADTs: Constructors are defined via decomposition.

Scott encoding only requires typed lambda calculus, which most modern languages provide!

Pulling ADTs Out of Thin Air

```
interface Maybe<T> {}   // The type `Maybe`…
```

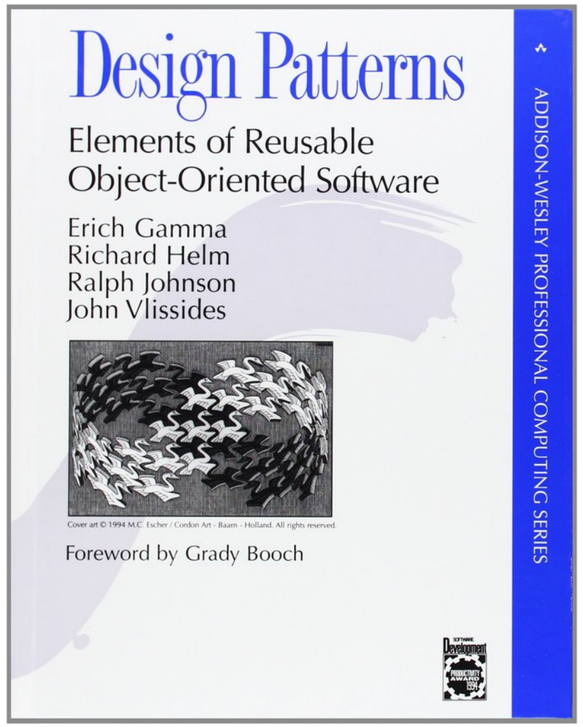
```
interface Maybe<T> {
  // …is defined by its decomposition, …
  maybe<R>(matchNothing: () → R, matchJust: (thing: T) → R): R;
}
```

```
// … and constructors are defined via the decomposition.
function nothing<T>(): Maybe<T> {
  return {
    maybe(matchNothing, matchJust) { return matchNothing(); }
  };
}

function just<T>(thing: T): Maybe<T> {
  return {
    maybe(matchNothing, matchJust) { return matchJust(thing); }
  };
}
```

```
// Verifying the duality:
just("x").match(nothing, just) === just("x");
nothing().match(nothing, just) === nothing();
```

The Visitor Pattern



The Visitor Pattern

```
interface Maybe<T> {
  accept<R>(visitor: MaybeVisitor<T, R>): R;
}

interface MaybeVisitor<T, R> {
  visitNothing(): R;
  visitJust(value: T): R;
}
```

```
class Nothing<T> implements Maybe<T> {
  accept<R>(visitor: MaybeVisitor<T, R>): R {
    return visitor.visitNothing();
  }
}

class Just<T> implements Maybe<T> {
  thing: T;

  accept<R>(visitor: MaybeVisitor<T, R>): R {
    return visitor.visitJust(thing);
  }
}
```

```
interface Maybe<T> {
  accept<R>(
    visitNothing: () ⇒ R,
    visitJust: (thing: T) ⇒ R): R;
}
```

```
function nothing<T>(): Maybe<T> {
  return {
    accept(visitNothing, visitJust) {
      return visitNothing;
    }
  };
}

function just<T>(thing: T): Maybe<T> {
  return {
    accept(visitNothing, visitJust) {
      return visitJust(thing);
    }
  };
}
```

Formal Description

1. Type Declaration with Decomposition
2. Declaration of Constructors
3. Write functions using the decomposition

1. Type Declaration with Decomposition

```
data Maybe a
    = Nothing
    | Just a
```

```
interface Maybe<T> {
  match<R>(
    matchNothing: () ⇒ R,
    matchJust: (thing: T) ⇒ R
  ): R;
}
```

2. Declaration of Constructors

```haskell
data Maybe a
    = Nothing
    | Just a
    -- constructors and patterns
    -- are declared in one declaration
```

```typescript
function nothing<T>(): Maybe<T> {
  return new (class extends Maybe<T> {
    match(matchNothing, matchJust) {
      return matchNothing();
    }
  })();
}

function just<T>(thing: T): Maybe<T> {
  return new (class extends Maybe<T> {
    match(matchNothing, matchJust) {
      return matchJust(thing);
    }
  })();
}
```

3. Profit!

```haskell
isNothing :: Maybe a -> Bool
isNothing maybeThing = case maybeThing of
    Nothing -> False
    Just _  -> True
```

```haskell
orElse :: Maybe a -> a -> a
maybeThing `orElse` fallback = case maybeThing of
    Nothing    -> fallback
    Just thing -> thing
```

```java
abstract class Maybe<T> {
  isNothing(): Boolean {
    return this.match(
      () -> false,
      thing -> true);
  }
```

```java
  orElse(T fallback): T {
    return match(
      () -> fallback,
      thing -> thing);
  }
}
```

More Examples

```haskell
data Either a b
    = Left a
    | Right b

isLeft :: Either a b -> Bool
isLeft either = case either of
    Left a -> True
    Right b -> False

flip :: Either a b -> Either b a
flip either = case either of
    Left a -> Right a
    Right b -> Left b

fmap :: (b -> c) -> Either a b -> Either a c
fmap f either = case either of
    Left a -> Left a
    Right b -> Right (f b)
```

```typescript
interface Either<A, B> {
  match<R>(
    matchLeft: (left: A) => R,
    matchRight: (right: B) => R
  ): R;
}

function left<A, B>(thing: A): Either<A, B> {
  return new (class extends Either<A, B> {
    match(matchLeft, matchRight) {
      return matchLeft(thing);
    }
  })();
}

function right<A, B>(thing: B): Either<A, B> {
  return new (class extends Either<A, B> {
    match(matchLeft, matchRight) {
      return matchRight(thing);
    }
  })();
}

abstract class Either<A, B> {
  isLeft(): Boolean {
```

In *any* Language?

Key ingredients:

- Interfaces (or abstract classes)
- Lambdas

Java Example

```
interface Either<A, B> {
  <R> R match(
      Function<A, R> matchLeft,
      Function<B, R> matchRight);

  static <A, B> Either<A, B> left(A thing) {
    return new Either<A, B>() {
      @Override public <R> R match(
          Function<A, R> matchLeft,
          Function<B, R> matchRight) {
        return matchLeft.apply(thing);
      };
    }
  }

  static <A, B> Either<A, B> right(B thing) {
    return new Either<A, B>() {
      @Override public <R> R match(
          Function<A, R> matchLeft,
          Function<B, R> matchRight) {
        return matchRight.apply(thing);
      };
    }
  }
```

What should I use it for?

- Standard ADTs (`Maybe`, `Either`, `Pair`, etc.)
- Syntax trees
- Parsed config files

Thank You!

franz.thoma@tngtech.com
github.com/fmthoma