An Implementation of Algebraic Data Types in Java using the Visitor Pattern

Anton Setzer

1. Algebraic Data Types.
2. Naive Modelling of Algebraic Types.
3. Defining Algebraic Types using Elimination Rules.

1. Algebraic Data Types

There are **two basic constructions** for introducing types in Haskell:

- **Function types**, e.g.
  Int $\longrightarrow$ Int.

- **Algebraic data types**, e.g.
  data Nat = Z | S Nat

  data List = Nil | Cons Int List

Idea of Algebraic Data Types

- data Nat = Z | S Nat
  Elements of Nat are exactly those which can be constructed from
  - Z,
  - S applied to elements of Nat.

  i.e. Z, S Z, S(S Z), ...

- data List = Nil | Cons Int List
  Elements of List are exactly those which can be constructed from
  - Nil,
  - Cons applied to elements of Int and List,

  e.g. Nil, Cons 3 Nil, Cons 17 (Cons 9 Nil), etc.

Infinitary Elements of Algebraic Data Types

- Because of full recursion and lazy evaluation, algebraic data types in Haskell contain as well **infinitary elements:**

- E.g. infiniteNat = S(S(S(⋯))).
  Defined as
  infiniteNat :: Nat
  infiniteNat = S infiniteNat.

- E.g. increasingStream 0 = Cons(0,Cons(1,⋯)).
  Defined as
  increasingStream:: Int → List
  increasingStream n = Cons n (increasingStream (n+1))

More Examples

- The **Lambda types and terms**, built from basic terms and types, can be defined as

```
data LambdaType = BasicType String
                | Ar LambdaType LambdaType
data LambdaTerm = BasicTerm String
                | Lambda String LambdaTerm
                | Ap LambdaTerm LambdaTerm
```

- In general many **languages** (or **term structures**) can be introduced as algebraic data types.

Elimination

- Elimination is carried out using **case distinction.**
  Example:

```
listLength    :: List → Int
listLength l  = case l of
                  Nil → 0
                  (Cons n l) → (listLength l) + 1
```

- Note the use of **full recursion.**

Simultaneous Algebraic Data Types

- Haskell allows as well the definition of **simultaneous algebraic data types**.

- Example: we define simultanteously the type of **finitely branching trees** and the type of **lists of such trees:**

```
data FinTree = Root
             | MkTree FinTreeList
data FinTreeList = NilTree
                 | ConsTree FinTree FinTreeList
```

Model for Positive Algebraic Data Types

- Assume a set of **constructors** (notation $\mathrm{C}, \mathrm{C}_1, \mathrm{C}_2 \ldots$) and **deconstructors** (notation $\mathrm{D}, \mathrm{D}_1, \mathrm{D}_2 \ldots$).

- Assume a set of **terms** s.t. every term has one of the following forms (where r,s are terms):
  - variables x.
  - C.
  - D.
  - r s.
  - λ x.r.

- We identify α-equivalent terms (i.e. those which differ only in the choice of bounded variables).

Model for Positive Algebraic Data Types (Cont.)

- Assume a reduction relation $\longrightarrow_1$ between terms, s.t.
  - $\mathsf{C}\,\mathsf{t}_1,\ldots,\mathsf{t}_\mathsf{n}$ has no reduct.
  - $\lambda x.t$ has no reduct.
  - x has no reduct.
  - α-equivalent terms have α-equivalent reducts.
- Let $\longrightarrow$ be the closure of $\longrightarrow_1$ under
  - reduction of subterms (if a subterm reduces the whole term reduces correspondingly)
  - transitive closrue.
- Assume that the reduct of $\longrightarrow$ is always a term.
- Assume that $\longrightarrow$ is confluent.

Model for Strictly Positive Algebraic Data Types (Cont.)

- For sets of terms S, T let

$$S \to T := \{r \mid \forall s \in S \exists t \in T. r\, s \longrightarrow t\}$$

- Assuming that we have defined the interpretation $[\![\mathsf{B}^i_j]\!]$, $[\![\mathsf{E}^i_j]\!]$ of types B^i_j, E^i_j.

Model for Strictly Positive Algebraic Data Types (Cont.)

- Let A be defined by

$$\begin{aligned}\mathsf{data}\ \mathsf{A} = \quad & \mathsf{C}_1\, \mathsf{B}^1_1 \cdots \mathsf{B}^1_{\mathsf{k}_1}\, (\mathsf{E}^1_1 \to \mathsf{A}) \cdots (\mathsf{E}^1_{\mathsf{l}_1} \to \mathsf{A}) \\ & \mid \cdots \\ & \mid \mathsf{C}_\mathsf{n}\, \mathsf{B}^\mathsf{n}_1 \cdots \mathsf{B}^\mathsf{n}_{\mathsf{k}_\mathsf{n}}\, (\mathsf{E}^\mathsf{n}_1 \to \mathsf{A}) \cdots (\mathsf{E}^\mathsf{n}_{\mathsf{l}_\mathsf{n}} \to \mathsf{A})\end{aligned}$$

- Define $\Gamma : \mathcal{P}(\mathsf{Term}) \to \mathcal{P}(\mathsf{Term})$,

$$\begin{aligned}\Gamma(X) := \{t \mid\ & t \text{ closed} \wedge \\ & \exists i \in \{1, \ldots, n\}. \\ & \exists t_1 \in [\![\, B^i_1 \,]\!]. \cdots \exists t_{k_i} \in [\![\, B^i_{k_i} \,]\!]. \\ & \exists s_1 \in [\![\, E^i_1 \,]\!] \to X. \cdots \exists s_{l_i} \in [\![\, E^i_{l_i} \,]\!] \to X. \\ & t \longrightarrow \mathsf{C}_i\, t^i_1 \cdots t^i_{k_i}\, s^i_1 \cdots s^i_{l_i}\}\end{aligned}$$

Model for Strictly Positive Algebraic Data Types (Cont.)

- The **algebraic data type** corresponding to A is defined as the **least fixed point** of Γ:

$$[\![A^*]\!] = \bigcap \{ X \subseteq \mathrm{Term} \mid \Gamma(X) \subseteq X \}$$

Model for Strictly Positive Algebraic Data Types (Cont.)

- Because of the presence of infinite elements, the algebraic data types in Haskell are in fact **coalgebraic data types**, given by the **greatest fixed points**.

- Terms which **never reduce to a constructor** should as well be added to this fixed point.
  We define the set of terms:

$$
\begin{aligned}
\Delta := \{t \mid t \text{ closed} \\
&\wedge(\neg\exists \mathsf{C}, n, t_1, \ldots, t_n . t \longrightarrow \mathsf{C}\, t_1, \ldots, t_n) \\
&\wedge(\neg\exists x, s . t \longrightarrow \lambda x . s)\}
\end{aligned}
$$

Model for Strictly Positive Algebraic Data Types (Cont.)

- The largest fixed point is given by

$$[\![A^{\infty}]\!] = \bigcup \{ X \subseteq \mathrm{Term} \mid X \supseteq \Delta \cup \Gamma(X) \}$$

"A^{∞} is the largest set s.t. every element reduces to an element introduced by a constructor of A."

Extensions

- Model can be extended to simultaneous and to general positive (co)algebraic data types.

2. Naive Modelling of Algebraic Types in Java

- We take as example the **type of LambdaTerms:**

  data LambdaType = BasicType String
  | Ar LambdaType LambdaType

- **Version 1**.
  - Model the set as a record consisting of
    - one **variable determining the constructor.**
    - the **arguments of each of the constructors**.
  - Only the variables corresponding to the arguments of the constructor in question are defined.

Version 1

```
class LambdaType{
   public int constructor;
   public String BTypeArg;
   public LambdaType ArArg1, ArArg2;
}

public static LambdaType BType(String s){
   LambdaType t = new LambdaType();
   t.constructor = 0;
   t.BTypeArg = s;
   return t;
};
```

Version 1 (Cont.)

```
public static LambdaType Ar(LambdaType left, LambdaType right){
   LambdaType t = new LambdaType();
   t.constructor = 1;
   t.ArArg1 = left;
   t.ArArg2 = right;
   return t;
};
```

Elimination

- **Functions defined by case distinction** can be introduced using the following pattern:

```
public static String LambdaTypeToString(LambdaType l){
  switch(l.constructor){
    case 0:
      return "BType(" + l.BTypeArg + ")";
    case 1:
      return "Ar("
              +LambdaTypeToString(l.ArArg1) +","
              +LambdaTypeToString(l.ArArg2) +")";
    default:
      throw new Error("Error");
  }
}
```

Generic Case Distinction

- On the next slide a **generic case distinction** is introduced.
  - We use the extension of Java by function types.
  - We assume the extension of Java by templates (might be integrated in Java version 1.5).

Generic Case Distinction (Cont.)

```
public static <Result> Result elim
        (LambdaType                              l,
         String→Result                           caseBType,
         (LambdaType,LambdaType)→Result          caseAr){
  switch(l.constructor){
    case 0:
      return caseBType(l.BTypeArg);
    case 1:
      return caseAr(l.ArArg1,l.ArArg2);
    default:
      throw new Error("Error");
   }
};
```

Generic Case Distinction (Cont.)

```
public static String LambdaTypeToString(LambdaType l){
    return elim(l,
                λ(String s)→{return "BType(" + s + ")";
                λ(LambdaTerm left, LambdaTerm right)→{
                    return "Ar("
                           + LambdaTypeToString(left) + ","
                           + LambdaTypeToString(right) + ")");
    }
```

Version 2

- Standard way of moving to a **more object-oriented solution**:
  - elim **should be a non-static method** of LambdaType.
  - Similarly LambdaTypeToString can be a **non static method** (with canonical name toString).
  - The methods BType and Ar can be integrated as a **factory** into the class LambdaType.
  - Now the variable constructor and the variables representing the arguments of the constructors of the algebraic data type can be **kept private**.
    - Implementation is encapsulated.

Version 2 (Cont.)

```
class LambdaType{
  private int constructor;
  private String BTypeArg;
  private LambdaType ArArg1, ArArg2;

  public static LambdaType BType(String s){
    LambdaType t = new LambdaType();
    t.constructor = 0;
    t.BTypeArg = s;
    return t;
  };
```

Version 2 (Cont.)

```
public static LambdaType Ar(LambdaType left, LambdaType right){
    LambdaType t = new LambdaType();
    t.constructor = 1;
    t.ArArg1 = left;
    t.ArArg2 = right;
    return t;
};
```

Version 2 (Cont.)

```
public <Result> Result elim(
   String→Result                              caseBType,
   (LambdaType,LambdaType)→Result   caseAr){
   switch(constructor){
     case 0:
       return caseBType(BTypeArg);
     case 1:
       return caseAr(ArArg1,ArArg2);
     default:
       throw new Error("Error");
     }
   };
```

Version 2 (Cont.)

```
public String toString(){
   return elim(
     λ(String s)→{
       return "BType(" + s + ")";
     },
     λ(LambdaType left, LambdaType right)→{
       return "Ar(" + left + "," + right + ")";}
     });
   }
 }
}
```

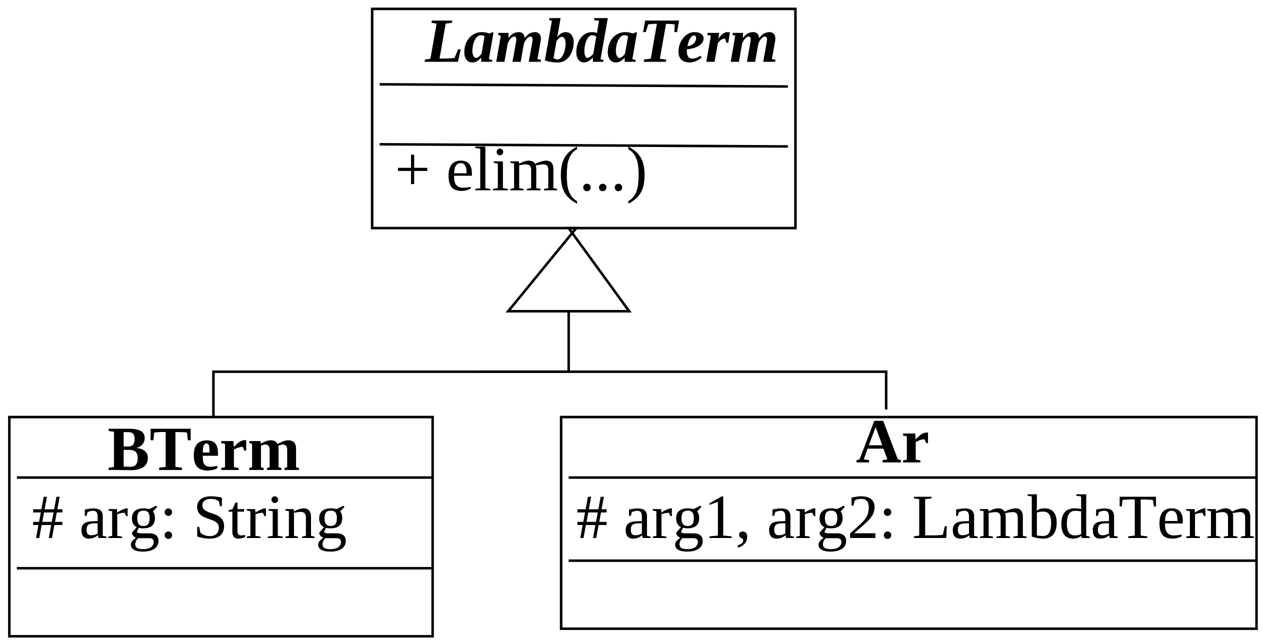
Problem with Version 2 (Cont.)

- We store **more variables than actually needed**:
  - The variable BTypeArg is only $\neq$null if constructor $= 0$.
  - The variables ArArg1, ArArg2 are only $\neq$null if constructor $= 1$.
  - Would be waste of storage for data type definitions with lots of constructors.

- **Solution:**
  Define LambdaType as an abstract class.
  - BType, Ar are subclasses of LambdaType which store the arguments of the constructor of the algebraic data type.

- Elim makes now **case distinction** on whether the current term is an instance of BType or Ar.
  - The element has then to be casted to an element of BType or Ar, in order to retrieve the arguments of the constructor of the algebraic data type.

Class Diagram for Version 3



Version 3

```
abstract class LambdaType{

  public <Result> Result elim(
      String→Result                              caseBType,
      (LambdaType,LambdaType)→Result   caseAr){
    if (this instanceof BType){
        return caseBType(((BType)this).arg);
    }
    else if (this instanceof Ar){
        return caseAr(((Ar)this).arg1,((Ar)this).arg2);
    }
    else
      {throw new Error("Error");
      };
    };
```

Version 3 (Cont.)

```
public String toString(){
   return elim(λ(String s){
         return "BType(" + s + ")";
         },
       λ(LambdaType left, LambdaType right){
         return "Ar(" + left + "," + right + ")";}
         });}
};
```

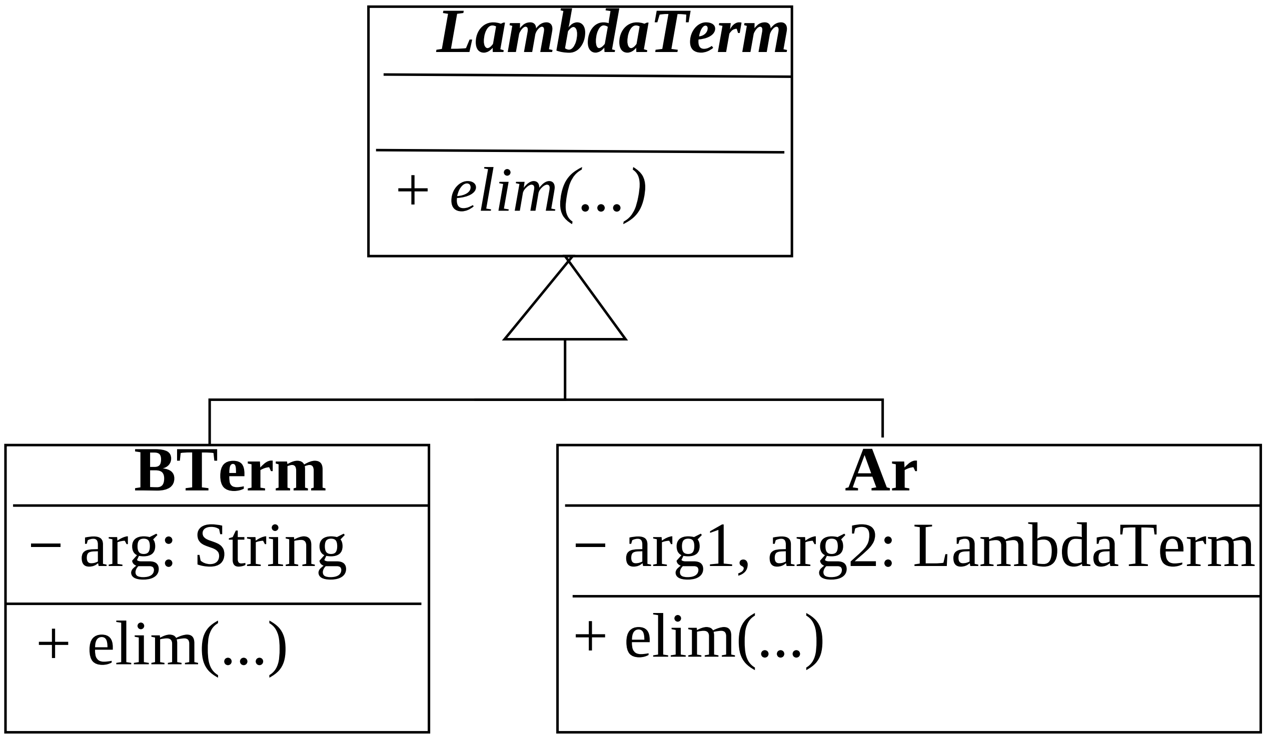
Version 3 (Cont.)

```
class BType extends LambdaType{
    protected String arg;
    public BType(String s){
      arg = s;
    };
};
 class Ar extends LambdaType{
    LambdaType arg1, arg2;
    protected Ar(LambdaType left, LambdaType right){
      arg1 = left;
      arg2 = right;
    }; };
```

Step to Version 4

- Instead of defining elim in LambdaType and then making case distinction, we can
  - leave elim in LambdaType **abstract**,
  - implement elim in BType and Ar.
- Then no more type casting is required.
- The arguments of the constructor of the algebraic data type can now be kept **private**.

Class Diagram for Version 4



Version 4

```
abstract class LambdaType{

    abstract public <Result> Result elim(
        String→Result                              caseBType,
        (LambdaType,LambdaType)→Result    caseAr);

    public String toString(){ ··· }
}
```

Version 4 (Cont.)

```
class BType extends LambdaType{
    private String arg;
    public BType(String s){
        arg = s;     };

    public <Result> Result elim(
        String→Result                          caseBType,
        (LambdaType,LambdaType)→Result    caseAr){
        return caseBType(arg);
    };

;
```

Version 4 (Cont.)

```
class Ar extends LambdaType{
    private LambdaType arg1, arg2;
    public Ar(LambdaType left, LambdaType right){
        arg1 = left;
        arg2 = right;     };

    public <Result> Result elim(
        String→Result                          caseBType,
        (LambdaType,LambdaType)→Result    caseAr){
        return caseAr(arg1,arg2);
    };
}
;
```

Further Simplification

- We can now form an interface which **subsumes all elimination steps**:

```
interface LambdaElim <Result>{
    Result caseBType(String s);
    Result caseAr(LambdaType left, LambdaType right);
;}
```

- An element of LambdaElim corresponds to the two elimination steps used previously.

Version 5

```
abstract class LambdaType{

    abstract public <Result> Result elim(LambdaElim<Result> steps);

    public String toString(){ ··· }
}
```

Version 5 (Cont.)

```
class BType extends LambdaType{
    private String s;
    public BType(String s){this.s = s;};

    public <Result> Result elim(LambdaElim <Result> steps){
        return steps.caseBType(s);}; };

class Ar extends LambdaType{
    private LambdaType left, right;
    public Ar(LambdaType left, LambdaType right){
        this.left = left; this.right = right;};

    public <Result> Result elim(LambdaElim <Result> steps){
        return steps.caseAr(left,right);};};
```

Generalization

- The above generalizes immediately to **all** (even non-positive) **algebraic data types**.
  - Using the implmentation of function types we can now for instance define **Kleene's O.**
- The type theory in Java allows simultaneous definitions of data types.
  - Therefore **simultanous algebraic definitions** are already **contained** in the above.

Example: Kleene's O

```
interface KleeneOElim <Result> {
  Result zeroCase();
  Result succCase(KleeneO x);
  Result limCase(Int→KleeneO x);};

abstract class KleeneO{
  public abstract <Result> Result elim(KleeneOElim <Result> steps);};

class Zero extends KleeneO{
  public Zero(){};

  public <Result> elim(KleeneOElim <Result> steps){
    return steps.zeroCase();};
};
```

Example: Kleene's O (Cont.)

```
class Succ extends KleeneO{
  private KleeneO pred;
  public Succ(KleeneO pred){
    this.pred = pred;
  };
  public <Result> elim(KleeneOElim <Result> steps){
    return steps.succCase(pred);};
};
class Lim extends KleeneO{
  private Int→KleeneO pred;
  public Lim(Int→KleeneO pred){
   this.pred = pred;};

  public <Result> elim(KleeneOElim <Result> steps){
   return steps.limCase(pred);};};
```

Example: FinTree

```
interface FinTreeElim <Result> {
  Result rootCase();
  Result mktreeCase(FinTreeList x);
};

abstract class FinTree{
  public abstract <Result> Result elim(FinTreeElim <Result> steps);};

class Root extends FinTree{
  public Root(){};

  public Object elim(FinTreeElim steps){
    return steps.rootCase();};
};
```

Example: FinTree (Cont.)

```
class MkTree extends FinTree{
   private FinTreeList pred;
   public MkTree(FinTreeList pred){
      this.pred = pred;
   };
   public <Result> Result elim(FinTreeElim <Result> steps){
      return steps.mktreeCase(pred);};
   };
```

Example: FinTree (Cont.)

```
interface FinTreeListElim <Result>{
   Result nilCase();
   Result consCase(FinTree x,FinTreeList l);
};

abstract class FinTreeList{
   public <Result> Result elim(FinTreeElimList <Result> steps);
};

class Nil extends FinTreeList{
   public Nil(){};
   public <Result> Result elim(FinTreeElimList <Result> steps){
      return steps.nilCase();};
};
```

Example: FinTree (Cont.)

```
class Cons extends FinTreeList{
  private FinTree arg1;
  private FinTreeList arg2;
  public Cons(FinTree arg1, FinTreeList arg2){
    this.arg1 = arg1;
    this.arg2 = arg2;
  };
  public <Result> Result elim(FinTreeElimList <Result> steps){
    return steps.consCase(arg1,arg2);
  };
};
```

3. Defining Algebraic Types using Elimination Rules

- If we look again at the Version-4-definition of class LambdaType, we see that it is defined by the elimination rule:

```
abstract class LambdaType{

   abstract public <Result> Result elim(
      String→Result                          caseBType,
      (LambdaType,LambdaType)→Result    caseAr);
}
```

- Note that we have no **recursion hypothesis**.
  - Not needed since we have **full recursion**.

Correctness

- For every element of the new types defined we have defined **case distinction**.
  - Given by **elim**.
- Further we have introduced the constructions corresponding to the **constructors of the algebraic data type**.
  - Given by the **Java-constructors of the subclasses** (BType, Ar) in the example above.

Correctness (Cont.)

- Finally the **equality rules** are fulfilled.
  - If $s = C_i\, a_1 \cdots a_{n_i}$, then

$$\begin{array}{ll} \mathsf{case}\ s\ \mathsf{of} & (C_1\, x^1_1 \cdots x^1_{n_1}) \longrightarrow \mathsf{case}_1\, x^1_1 \cdots x^1_{n_1} \\ & \cdots \\ & (C_k\, x^k_1 \cdots x^k_{n_k}) \longrightarrow \mathsf{case}_k\, x^k_1 \cdots x^k_{n_k} \end{array}$$

    should evaluate to

$$\mathsf{case}_k\, a_1 \cdots a_{n_i}$$

  - But thats the **definition of elim** in that case.

Correctness (Cont.)

- Therefore the algebraic data types are modelled in such a way that the **introduction, elimination and equality principles** for those types are fulfilled.

- This means that from a type theoretic point of view this is a **correct implementation of the algebraic data type**.

Comparision with System F

- The definition of an inductive data type by its elimination rules is similar to the **second order definition of algebraic data types in System F**:

- Example: Nat in System F is defined as

$$\forall X.X \rightarrow (X \rightarrow X) \rightarrow X$$

  - Zero is defined as $\Lambda X.\lambda z.\lambda s.z$.
  - Succ(n) is defined as $\Lambda X.\lambda z.\lambda s.s(n\,X\,z\,s)$

- In our definition, **references to the recursion hypothesis are replaced by references to the type to be defined**.

Nat

```
abstract class Nat{

  abstract public <X> X elim(
    ()→X     caseZero,
    Nat→X    caseSucc);
}
```

$$\mathsf{Nat} = \forall X.(() \to X) \to (\mathsf{Nat} \to X) \to X$$

Nat (Cont.)

```
class Zero extends Nat{
    public Zero(){};

    public <X> X elim(
        ()→X       caseZero,
        Nat→X    caseSucc){
        return caseZero();
    };
};
```

$$\mathsf{Zero} = \lambda \mathsf{X}.\lambda \mathsf{caseZero}.\lambda \mathsf{caseSucc}.\mathsf{caseZero}()$$

Nat (Cont.)

```
class Succ extends Nat{
    private Nat n;
    public Succ(Nat n){
        this.n = n;
    };

    public <X> X elim(
        ()→X      caseZero,
        Nat→X     caseSucc){
        return caseSucc(n);
    };
};
```

$$\text{Succ}\, n = \lambda \text{X}.\lambda \text{caseZero}.\lambda \text{caseSucc}.\text{caseSucc}(n)$$

Visitor Pattern

- The solution found is very close to the **visitor pattern.**

- An implementation of **LambdaTerm** by using the visitor pattern can be obtained from the above solution as follows:
  - Replace the return type of the elim method by **void**.
    - If one wants to obtain a result, one can export it using side effects.
    - However, to export it this way is rather cumbersome.

Visitor Pattern (Cont.)

- Instead of referring to the arguments of the constructor of the algebriac data type in elim, one **refers to the whole object** (which is an element of BType or Ar).
  - If their arguments are public we can access them.
- Now all **the steps** of LambdaElim have **different argument types**.
  - Because of polymorphism, we can give all steps the same name, "**visit**".
- LambdaElim is called in the visitor pattern **Visitor**.
- elim is called **accept**.

LambdaType Using the Visitor Pattern

```
interface Visitor{
    void visit(BType   t);
    void visit(Ar      t);
;}

abstract class LambdaType{
    abstract public void accept(Visitor v);
}
```

LambdaType Using the Visitor Pattern (Cont.)

```
class BType extends LambdaType{
    private String s;
    public BType(String s){this.s = s;};

    public void accept(Visitor v){
        v.visit(this);};
};

class Ar extends LambdaType{
    private LambdaType left, right;
    public Ar(LambdaType left, LambdaType right){
        this.left = left; this.right = right;};

    public void accept(Visitor v){
        v.visit(this);};
};
```

Disjoint Union

- If we look again at definition of the visitor:

```
interface Visitor{
    void visit(BType    t);
    void visit(Ar       t);
;}

abstract class LambdaType{
    abstract public void accept(Visitor v);
}
```

we see that this is **the definition of LambdaType as the disjoint union of BType and Ar**:

$$\text{LambdaType} = \text{BType} + \text{Ar}$$

Disjoint Union (Cont.)

- To define the **product of types** in Java is **trivial**:
  This is a record type.

- To define a **set recursively** is **built into the Java type checker**.

- The definition of

$$\mathsf{data}\ A = C_1(\cdots) \mid \cdots \mid C_k(\cdots)$$

  can be split into two parts:

  - $A = B_1 + \cdots + B_k$

  - $\mathsf{data}\ B_i = C_i(\cdots)$.

- The visitor pattern is essentially a (because of the return type void sub-optimal) **way of defining the disjoint union**.

Conclusion

- Started with a **naive implementation of algebraic data types**.
- Derived from this a definition, which defines algebraic data types by **elimination**.
- Carried out a **comparison with the Visitor pattern**.
- Found that the visitor pattern is an **implementation of the disjoint union**.
- The implementation in Java is considerably **much longer than the definition in Haskell**.
  - Need for **suitable syntactic sugar** for algebraic data types in Java.