Relationen-Algebra und Persistenz

Implementierungskonzepte und Anforderungen an Attributdatentypen

Christian Düntgen

12.10.2007

FernUniversität in Hagen

Inhalt

- Datentypen und Operationen der RelationAlgebra
- DBArrays und FLOBs
- Attributsdatentypen
- Tupel
- Referenzzähler
- Komplexere Type-Mappings
- Tipps zum Debugging

Grundlegende Datentypen

- Kinds: IDENT, DATA, TUPLE, REL
- Typkonstruktoren: *rel* und *tuple*

```
             Name: tuple
        Signature: (ident x DATA)+ -> TUPLE
Example Type List: (tuple((name string)(age int)))
         List Rep: (<attr1> ... <attrn>)
     Example List: ("Myers" 53)

             Name: rel
        Signature: TUPLE -> REL
Example Type List: (rel(tuple((name string)(age int))))
         List Rep: (<tuple>*)where
                    <tuple> is (<attr1> ... <attrn>)
     Example List: (("Myers" 53)("Smith" 21))
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume
```

```
     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
```


```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation
           by scanning the relation
           tuple by tuple.
  Example: query cities feed consume
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
```


```
     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume
```

```
     Name: filter
Signature: ((stream x)(map x bool))
           -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a
           certain condition are
           passed on to the output stream.
  Example: query cities feed filter
           [.population > 500000] consume
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)
           -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute
           at a given position.
  Remarks: not for use with sos-syntax
```

```
Example: query cities feed filter [.population > 500000] consume
```

```
     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Grundlegende Operationen

```
     Name: feed
Signature: (rel x) -> (stream x)
   Syntax: _ feed
  Meaning: Produces a stream from a relation by scanning
           the relation tuple by tuple.
  Example: query cities feed consume

     Name: consume
Signature: (stream x) -> (rel x)
   Syntax: _ consume
  Meaning: Collects objects from a stream.
  Example: query cities feed consume

     Name: filter
Signature: ((stream x) (map x bool)) -> (stream x)
   Syntax: _ filter [ fun ]
  Meaning: Only tuples, fulfilling a certain condition are
           passed on to the output stream.
  Example: query cities feed filter [.population > 500000] consume

     Name: attr
Signature: ((tuple ((x1 t1)...(xn tn))) xi)  -> ti)
   Syntax: attr ( _ , _ )
  Meaning: Returns the value of an attribute at a given position.
  Remarks: not for use with sos-syntax
```

Mit diesen Operationen kann die Selektion im folgenden Beispiel realisiert werden:

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;
```

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;
```

Übersetzung der Query in die Listendarstellung

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;

(query
    (consume
        (filter
            (feed cities)
            (fun
                (tuple1 TUPLE)
                (>
                    (attr tuple1 population)
                    500000)))))
```

Übersetzung der Query in die Listendarstellung

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;

(query
    (consume
        (filter
            (feed cities)
            (fun
                (tuple1 TUPLE)
                (>
                    (attr tuple1 population)
                    500000)))))
```

Übersetzung der Query in die Listendarstellung

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;

(query
    (consume
        (filter
            (feed cities)
            (fun
                (tuple1 TUPLE)
                (>
                    (attr tuple1 population)
                    500000)))))
```

Übersetzung der Query in die Listendarstellung

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;

(query
    (consume
        (filter
            (feed cities)
            (fun
                (tuple1 TUPLE)
                (>
                    (attr tuple1 population)
                    500000)))))
```

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;
```

.spec-File:
Benutzung von Argumentfunktionen mit
impliziten Parametern und des Typabbildungsoperators **TUPLE**:

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;
```

.spec-File:
Benutzung von Argumentfunktionen mit impliziten Parametern und des Typabbildungsoperators **TUPLE**:

```
operator filter alias FILTER
   pattern _ op [ fun ]
   implicit parameter tuple type TUPLE
```

Grundlegende Operationen

```
Secondo =>
   query cities feed filter[.population > 500000] consume;

(query
    (consume
        (filter
            (feed cities)
            (fun
                (tuple1 TUPLE)
                (>
                    (attr tuple1 population)
                    500000)))))
```

```
Type mapping for operator filter:
RelationAlgebra: filter (algId=3, opId=5) accepted!
IN: ((stream (tuple ((name string) (population int))))
     (map (tuple((name string) (population int))) bool))
OUT: (stream (tuple ((name string) (population int))))
```

FLOB

- Datenstruktur zur Speicherung beliebiger Datenmengen
- Eingebauter Persistenzmechanismus
- Verwaltet "**unstrukturierte Speicherblöcke**" (wie `malloc`)
- Zugriff mittels spezieller Funktionen und eines "**offsets**"

```
class FLOB
{
    FLOB( size_t size );
    size_t Size() const;
    void Put( size_t offset, size_t length, const void *source );
    void Get( size_t offset, const char **target,
                    const bool paged = false ) const;
    void Resize( size_t size );
    void Clean();
    void Destroy();
}
```

FLOB

Achtung!

FLOBs werden direkt “en bloc” auf die Platte geschrieben.
Flobs werden an irgendeine Stelle in den Speicher eingelesen.

Verwenden Sie keine Zeiger in FLOBs, sondern **offsets**!

```
class FLOB
{
    FLOB( size_t size );
    size_t Size() const;
    void Put( size_t offset, size_t length, const void *source );
    void Get( size_t offset, const char **target,
                    const bool paged = false ) const;
    void Resize( size_t size );
    void Clean();
    void Destroy();
}
```

DBArray

- **DBArray** ist eine Template-Klasse, die von **FLOB** abgeleitet ist.
- **DBArray** ist eine Template-Klasse, die von **FLOB** abgeleitet ist.Um die Implementierung von Datentypen, deren Speicherplatzbedarf stark variieren kann, effizient zu unterstützen, wurde die Klasse **DBArray** entworfen.
- Ein- und Auslagerung von Datenteilen muss daher (wie beim **FLOB**) nicht selbst programmiert werden!
- Wird ein **DBArray** sortiert abgespeichert, so ist eine effiziente externe binäre Suche möglich.

DBArray

```
template<class DBArrayElement> class DBArray : public FLOB
{
   DBArray( int n );
   int Size() const;
   void Resize( const int newSize );
   void Clear();
   void Put( int index, const DBArrayElement& elem );
   void Append( const DBArrayElement& elem );
   void Get( int index, DBArrayElement const*& elem ) const;
   void TrimToSize();
   void Sort( int (*cmp)( const void *a, const void *b) );
   bool Find( const void *key,
              int (*cmp)( const void *a, const void *b),
              int& result ) const
   void Destroy();
   [...]
}
```

Attributdatentypen

- Können in Relationen als Attribute verwendet werden.
- Attributdatentyp: Oberklasse `Attribute`, Kind `DATA`.
- Erfordernisse:

Attributdatentypen

- Können in Relationen als Attribute verwendet werden.
- Attributdatentyp: Oberklasse `Attribute`, Kind `DATA`.
- Erfordernisse:
  - Sicherstellen einer Mindest-Funktionalität
    (`Compare`, `IsDefined`, `SetDefined`, `Adjacent`, `Print`, ...)

Attributdatentypen

- Können in Relationen als Attribute verwendet werden.
- Attributdatentyp: Oberklasse `Attribute`, Kind `DATA`.
- Erfordernisse:
  - Sicherstellen einer Mindest-Funktionalität
    (`Compare`, `IsDefined`, `SetDefined`, `Adjacent`, `Print`, ...)
  - Support für den automatischen Persistenzmechanismus von Tupeln (C++ kennt keine Serialisierung!)
    (`NumOfFLOBs`, `GetFLOB`, `SizeOf`, `Open`, `Save`)

Attributdatentypen

- Können in Relationen als Attribute verwendet werden.
- Attributdatentyp: Oberklasse `Attribute`, Kind `DATA`.
- Erfordernisse:
  - Sicherstellen einer Mindest-Funktionalität (`Compare`, `IsDefined`, `SetDefined`, `Adjacent`, `Print`, ...)
  - Support für den automatischen Persistenzmechanismus von Tupeln (C++ kennt keine Serialisierung!) (`NumOfFLOBs`, `GetFLOB`, `SizeOf`, `Open`, `Save`)
  - **Verwenden keine dynamischen Objekte/Zeiger!**

Attributdatentypen

- Können in Relationen als Attribute verwendet werden.
- Attributdatentyp: Oberklasse `Attribute`, Kind `DATA`.

Beispiel

Der Datentyp **`Polygon`** aus der Polygon Algebra stellt einen Datentyp dar, der

- als **Attribut** in Relationen verwendet werden kann und
- den Speicherverwaltungstyp **`DBArray`** verwendet:

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Klasse `Polygon` wird von Klasse `Attribute` abgeleitet

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
  public:
    int Compare(const Attribute*) const;
      [...]
  private:
    DBArray<Vertex> vertices;
    PolygonState state;
};
```

Wichtig:

Bei Mehrfachvererbung darf keine Klasse mehrfach von derselben Klasse (z.B. `Attribute`) erben.

Sonst: Probleme bei der Typkonvertierung (cast).

Klasse `Polygon` wird von Klasse `Attribute` abgeleitet

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribu
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Standard-Konstruktor
Wird vom Persistenzmechanismus verwendet.
Darf das Objekt nicht verändern.

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
  public:
    [...]
  private:
    DBArray<Vertex> vertices;
    PolygonState state;
};
```

Wichtig:
Vermeiden Sie die Nutzung des Standard-Konstruktors.
Diese führt häufig wegen uninitialisierter Werte zu Fehlern.
Initialisieren Sie Objekte mittels anderer Konstruktoren.

Standard-Konstruktor
Wird vom Persistenzmechanismus verwendet.
Darf das Objekt nicht verändern.

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Einziges **FLOB-Objekt** in Polygon

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Att
  {
   public:
     Polygon() {}
     Polygon( const int n,

     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Anzahl der FLOBs zurückgeben:

```
int
Polygon::NumOfFLOBs() const
{
  return 1;
}
```

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Attributdatentypen (Bsp. Polygon)

```
class Polygon : pub
  {
   public:
     Polygon() {}
     Polygon( const

     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Den i-ten FLOB zurückgeben:

```
FLOB *Polygon::GetFLOB(const int i)
{
  assert( i >= 0 &&
          i < NumOfFLOBs() );
  return &vertices;
}
```

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Attributdatentypen (Bsp. Polygon)

```
class Polyg
  {
   public:
     Polygon
     Polygon

     ~Polygo

     int Num
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

`int Compare(const Attribute* rhs) const;`

Falls beide Objekte definiert:

`*this < *rhs` ⟶ `-1`
`*this = *rhs` ⟶ `0`
`*this > *rhs` ⟶ `1`

Sonst:

`undef x def` ⟶ `-1: *this < *rhs`
`undef x undef` ⟶ `0: *this = *rhs`
`def x undef` ⟶ `1: *this > *rhs`

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     Polygon( const int n, const int *X = 0,
                                const int *Y = 0 );
     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Attributdatentypen (Bsp. Polygon)

```
class Polygon : public Att
  {
   public:
     Polygon() {}
     Polygon( const int n,

     ~Polygon();

     int NumOfFLOBs() const;
     FLOB *GetFLOB(const int i);
     int Compare(const Attribute*) const;
       [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Weitere zu überschreibende Funktionen:

```
bool Adjacent(const Attribute*) const;
Polygon *Clone() const;
bool IsDefined() const;
void SetDefined( bool defined );
size_t Sizeof() const;
```

Attributdatentypen (Bsp. Polygon)

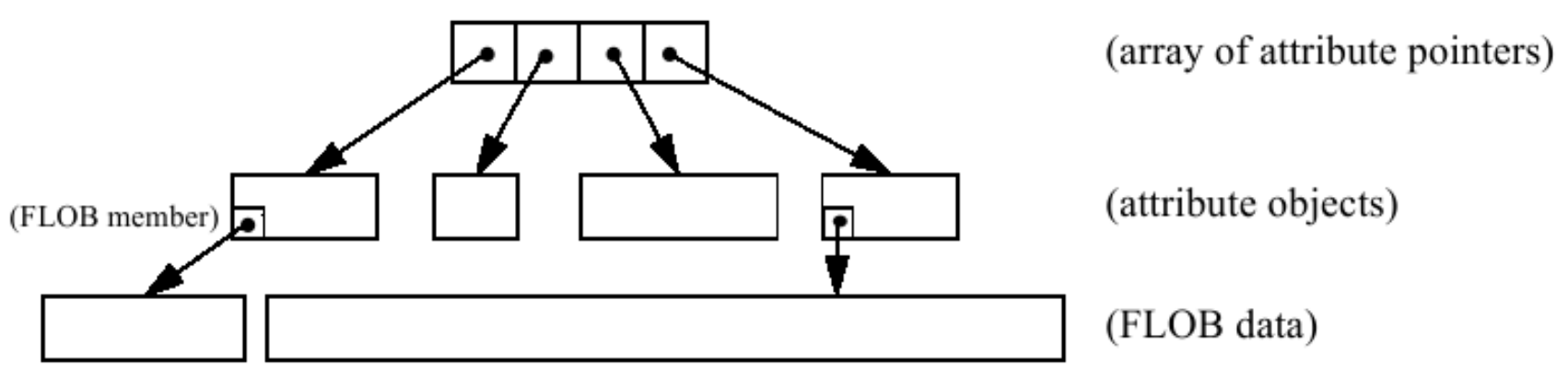
```
class Polygon : public Attribute
  {
   public:
     Polygon() {}
     int Compare(const Attribute*) const;
      [...]
   private:
     DBArray<Vertex> vertices;
     PolygonState state;
};
```

Wichtig:

Überschreibt man diese Funktionen nicht, z.B. weil man sich im Namen mit Groß- oder Kleinschreibung vertan hat, führt dies mitunter zu **Speicherfehlern**!

Tupel

- Ein Tupel wird (grob gesehen) dargestellt durch ein Array von Zeigern auf Attributdatentyp-Objekte:



- Konstruktoren

```
Tuple( TupleType* tupleType )
Tuple( ListExpr typeInfo )
```

- Zugriff auf Attribute mittels

```
Attribute* GetAttribute( int index )
void PutAttribute( int index, Attribute* attr )
void CopyAttribute(  int sourceIndex, Tuple source
                     int destIndex )
```

Referenzzähler

Referenzzähler

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen, und Attributen, z.B. in der Stromverarbeitung

Referenzzähler

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen, und Attributen, z.B. in der Stromverarbeitung
- **Attributsdatentypen**: Automatische Verwaltung

  `Clone`: Erzeugt eine Tiefenkopie (= 1. Referenz),

  `Copy`: Erzeugt eine weitere Referenz

  (max. 254 mal, dann `Clone`-Aufruf!)

  `DeleteIfAllowed`: Aufgabe einer Referenz

  (ggf. auch Löschen des Attributobjekts)

Referenzzähler

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen, und Attributen, z.B. in der Stromverarbeitung
- **Attributsdatentypen**: Automatische Verwaltung
  `Clone`: Erzeugt eine Tiefenkopie (= 1. Referenz),
  `Copy`: Erzeugt eine weitere Referenz
  (max. 254 mal, dann `Clone`-Aufruf!)
  `DeleteIfAllowed`: Aufgabe einer Referenz
  (ggf. auch Löschen des Attributobjekts)
- `CopyAttribute`: Aufruf von `Attributtyp::Copy` für ein Tupelattribut (automatische Verwaltung)

Referenzzähler

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen, und Attributen, z.B. in der Stromverarbeitung
- **Attributsdatentypen**: Automatische Verwaltung
  `Clone`: Erzeugt eine Tiefenkopie (= 1. Referenz),
  `Copy`: Erzeugt eine weitere Referenz
  (max. 254 mal, dann `Clone`-Aufruf!)
  `DeleteIfAllowed`: Aufgabe einer Referenz
  (ggf. auch Löschen des Attributobjekts)
- `CopyAttribute`: Aufruf von `Attributtyp::Copy` für ein Tupelattribut (automatische Verwaltung)
- **Tupel**: Manuelle Verwaltung (Beginn: Zähler = 0)
  `IncReference`: Zähler++,
  `DecReference`: Zähler--,
  `DeleteIfAllowed`: Löscht das Tupelobjekt, falls Zähler == 0.

Referenzzähler: Beispiele

Referenzzähler: Beispiele

- Operator “erzeugt” Tupel:
  1) `resultTuple = new Tuple(...);` Erzeuge neues Tupel

Referenzzähler: Beispiele

- Operator “erzeugt” Tupel:
  1) `resultTuple = new Tuple(...);` Erzeuge neues Tupel
- Operator reicht Original-Tupel weiter: Nichts zu tun.

Referenzzähler: Beispiele

- Operator “erzeugt” Tupel:
  1) `resultTuple = new Tuple(...);` Erzeuge neues Tupel
- Operator reicht Original-Tupel weiter: Nichts zu tun.
- Operator “verbraucht” Tupel:
  1) `oldTuple->DeleteIfAllowed;`

Referenzzähler: Beispiele

- Operator “erzeugt” Tupel:
  1) `resultTuple = new Tuple(...);` Erzeuge neues Tupel
- Operator reicht Original-Tupel weiter: Nichts zu tun.
- Operator “verbraucht” Tupel:
  1) `oldTuple->DeleteIfAllowed;`
- Operator “ändert” Tupel
  1) `resultTupel = new Tuple(...);` Erzeuge neues Tupel
  2) `Clone` zu *ändernde* Attribute. `Copy` für *sonstige* Attribute.
  2) `oldTuple->DeleteIfAllowed` Verbrauche altes Tupel

Referenzzähler: Beispiele

- Operator “erzeugt” Tupel:
  1) `resultTuple = new Tuple(...);` Erzeuge neues Tupel
- Operator reicht Original-Tupel weiter: Nichts zu tun.
- Operator “verbraucht” Tupel:
  1) `oldTuple->DeleteIfAllowed;`
- Operator “ändert” Tupel
  1) `resultTupel = new Tuple(...);` Erzeuge neues Tupel
  2) `Clone` zu *ändernde* Attribute. `Copy` für *sonstige* Attribute.
  2) `oldTuple->DeleteIfAllowed` Verbrauche altes Tupel
- **Optional**: Eine private “Kopie” (Referenz) anlegen
  1) `copiedTuple = theTuple;` Referenz anlegen
  2) `theTuple->IncReference;` Referenzzähler erhöhen
  3) ...
  4) `theTuple->DecReference;` Referenzzähler verringern
  5) `theTuple->DeleteIfAllowed;` Referenz ggf. löschen

Referenzzähler: Beispiele

- Operator “erzeugt” Tupel:
  1) `resultTuple = new Tuple(...);` Erzeuge neues Tupel
- Operator reicht Original-Tupel weiter: Nichts zu tun.
- Operator “verbraucht” Tupel:
  1) `oldTuple->DeleteIfAllowed;`
- Operator “ändert” Tupel
  1) `resultTupel = new Tuple(...);` Erzeuge neues Tupel
  2) `Clone` zu *ändernde* Attribute. `Copy` für *sonstige* Attribute.
  2) `oldTuple->DeleteIfAllowed` Verbrauche altes Tupel
- **Optional**: Eine private “Kopie” (Referenz) anlegen
  1) `copiedTuple = theTuple;` Referenz anlegen
  2) `theTuple->IncReference;` Referenzzähler erhöhen
  3) ...
  4) `theTuple->DecReference;` Referenzzähler verringern
  5) `theTuple->DeleteIfAllowed;` Referenz ggf. löschen

Der feed Operator

```
int
Feed(Word* args, Word& result, int message, Word& local, Supplier s)
{
  GenericRelation* r=0;
  GenericRelationIterator* rit=0;

  switch (message)
  {
    case OPEN :
      r = (GenericRelation*)args[0].addr;
      rit = r->MakeScan();

      local = SetWord(rit);
      return 0;

    case REQUEST :
      rit = (GenericRelationIterator*)local.addr;
      Tuple *t;
      if ((t = rit->GetNextTuple()) != 0)
      {
        result = SetWord(t);
        return YIELD;
      }
      else
      {
        return CANCEL;
      }

    case CLOSE :
      rit = (GenericRelationIterator*)local.addr;
      delete rit;
      return 0;
  }
  return 0;
}
```

Referenzzähler: Beispiele

- Operator “erzeugt” Tupel:
  1) `resultTuple = new Tuple(...);` Erzeuge neues Tupel
- Operator reicht Original-Tupel weiter: Nichts zu tun.
- Operator “verbraucht” Tupel:
  1) `oldTuple->DeleteIfAllowed;`
- Operator “ändert” Tupel
  1) `resultTupel = new Tuple(...);` Erzeuge neues Tupel
  2) `Clone` zu *ändernde* Attribute. `Copy` für *sonstige* Attribute.
  2) `oldTuple->DeleteIfAllowed` Verbrauche altes Tupel
- **Optional**: Eine private “Kopie” (Referenz) anlegen
  1) `copiedTuple = theTuple;` Referenz anlegen
  2) `theTuple->IncReference;` Referenzzähler erhöhen
  3) ...
  4) `theTuple->DecReference;` Referenzzähler verringern
  5) `theTuple->DeleteIfAllowed;` Referenz ggf. löschen

Der consume Operator

```
int
Consume(Word* args, Word& result, int message, Word& local, Supplier s)
{
  Word actual;

  GenericRelation* rel = (Relation*)((qp->ResultStorage(s)).addr);
  if(rel->GetNoTuples() > 0)
  {
    rel->Clear();
  }

  qp->Open(args[0].addr);
  qp->Request(args[0].addr, actual);
  while (qp->Received(args[0].addr))
  {
    Tuple* tuple = (Tuple*)actual.addr;
    rel->AppendTuple(tuple);
    tuple->DeleteIfAllowed();
    qp->Request(args[0].addr, actual);
  }
  result = SetWord(rel);
  qp->Close(args[0].addr);
  return 0;
}
```

Komplexere Type-Mappings

- Manchmal ist es hilfreich bzw. notwendig, daß durch das Type-Mapping weitere (interne) Argumente angehangen werden können, die dem Value-Mapping dann zur Verfügung stehen.
- Dazu dient das Schlüsselwort APPEND. Wird im Type-mapping (APPEND (v_1 ... v_n) <t*ype-list*>) zurückgegeben, so werden die Werte v_i als zusätzliche Argumente weitergereicht.
- Die Werte v_i können auch selbst wieder Listen sein, dann muß im Value-Mapping mittels `qp->GetSupplier(...)` darauf zugegriffen werden, siehe z.B. Operator **project**.

Komplexere Type-Mappings

- Manchmal ist es hilfreich bzw. notwendig, daß durch das Type-Mapping weitere (interne) Argumente angehangen werden können, die dem Value-Mapping dann zur Verfügung stehen.
- Dazu dient das Schlüsselwort APPEND. Wird im Type-mapping (APPEND (v_1 ... v_n) <t*ype-list*>) zurückgegeben, so werden die Werte v_i als zusätzliche Argumente weitergereicht.
- Die Werte v_i können auch selbst wieder Listen sein, dann muß im Value-Mapping mittels `qp->GetSupplier(...)` darauf zugegriffen werden, siehe z.B. Operator **project**.
- Hier als Beispiel: Operator **attr**

Beispiel: Operator attr

```
ListExpr AttrTypeMap(ListExpr args)
{
  ListExpr first, second, attrtype;
  string  attrname, argstr;
  int j;

  nl->WriteToString(argstr, args);
  CHECK_COND(nl->ListLength(args) == 2,  "Operator attr expects a list of length two. But got " + argstr + "!");

  first = nl->First(args);
  nl->WriteToString(argstr, first);
  CHECK_COND( (nl->ListLength(first) == 2) && (TypeOfRelAlgSymbol(nl->First(first)) == tuple),
  "Operator attr expects as first argument a list with structure (tuple ((a1 t1)...(an tn)))\n"
  "Operator attr gets a list with structure '" + argstr + "'.");

  second  = nl->Second(args);
  nl->WriteToString(argstr, second);
  CHECK_COND( nl->IsAtom(second) &&
              nl->AtomType(second) == SymbolType,
  "Operator attr expects as second argument a symbol atom  (attributename).\n
  ”Operator attr gets '" + argstr + "'.");

  attrname = nl->SymbolValue(second);
  j = FindAttribute(nl->Second(first), attrname, attrtype);
  if (j)
    return nl->ThreeElemList(nl->SymbolAtom("APPEND"),
           nl->OneElemList(nl->IntAtom(j)), attrtype);
  else
  {
    nl->WriteToString( argstr, nl->Second(first) );
    ErrorReporter::ReportError(
    "Attribute name '" + attrname + "' is not known in the tuple.\n"
    "Known Attribute(s): " + argstr);

    return nl->SymbolAtom("typeerror");
  }
}
```

Beispiel: Operator attr

```
ListExpr AttrTypeMap(ListExpr args)
{
  ListExpr first, second, attrtype;
  string  attrname, argstr;
  int j;

  nl->WriteToString(argstr, args);
  CHECK_COND(nl->ListLength(args) == 2,  "Operator attr expects a list of length two. But got " + argstr + "!");

  first = nl->First(args);
  nl->WriteToString(argstr, first);
  CHECK_COND( (nl->ListLength(first) == 2) && (TypeOfRelAlgSymbol(nl->First(first)) == tuple),
  "Operator attr expects as first argument a list with structure (tuple ((a1 t1)...(an tn)))\n"
  "Operator attr gets a list with structure '" + argstr + "'.");

  second  = nl->Second(args);
  nl->WriteToString(argstr, second);
  CHECK_COND( nl->IsAtom(second) &&
              nl->AtomType(second) == SymbolType,
  "Operator attr expects as second argument a symbol atom  (attributename).\n
  ”Operator attr gets '" + argstr + "'.");

  attrname = nl->SymbolValue(second);
  j = FindAttribute(nl->Second(first), attrname, attrtype);
  if (j)
    return nl->ThreeElemList(nl->SymbolAtom("APPEND"),
           nl->OneElemList(nl->IntAtom(j)), attrtype);
  else
  {
    nl->WriteToString( argstr, nl->Second(first) );
    ErrorReporter::ReportError(
    "Attribute name '" + attrname + "' is not known in the tuple.\n"
    "Known Attribute(s): " + argstr);

    return nl->SymbolAtom("typeerror");
  }
}
```

Beispiel: Operator attr

```
ListExpr AttrTypeMap(ListExpr args)
{
  ListExpr first, second, attrtype;
  string  attrname, argstr;
  int j;

  nl->WriteToString(argstr, args);
  CHECK_COND(nl->ListLength(args) == 2,  "Operator attr expects a list of length two. But got " + argstr + "!");

  first = nl->First(args);
  nl->WriteToString(argstr, first);
  CHECK_COND( (nl->ListLength(first) == 2) && (TypeOfRelAlgSymbol(nl->First(first)) == tuple),
  "Operator attr expects as first argument a list with structure (tuple ((a1 t1)...(an tn)))\n"
  "Operator attr gets a list with structure '" + argstr + "'.");

  second  = nl->Second(args);
    attrname = nl->SymbolValue(second);
    j = FindAttribute(nl->Second(first),
                      attrname, attrtype);
    if (j)
      return nl->ThreeElemList(nl->SymbolAtom("APPEND"),
             nl->OneElemList(nl->IntAtom(j)), attrtype);
    else
    {
      return nl->SymbolAtom("typeerror");
    }
  }
}
```

Beispiel: Operator attr

Type mapping for operator attr:

```
RelationAlgebra: attr (algId=3, opId=4) accepted!
IN: ((tuple ((name string) (population int))) population)
OUT: (APPEND (2) int)
```

Beispiel: Operator attr

Type mapping for operator attr:
RelationAlgebra: attr (algId=3, opId=4) accepted!
IN: ((tuple ((name string) (population int))) population)
OUT: **(APPEND (2) int)**

```
int Attr(Word* args, Word& result, int message,
                                    Word& local, Supplier s)
{
  Tuple* tupleptr = 0;
  int index = 0;

  tupleptr = (Tuple*)args[0].addr;
  // args[1] is not used since the attribute name is unimportant here.
  // Instead we need the attribute's index which was computed in the
  // operator's type map function.
  index = ((CcInt*)args[2].addr)->GetIntval();
  assert(1 <= index &&
         index <= tupleptr->GetNoAttributes() );
  result = SetWord(tupleptr->GetAttribute(index - 1));
  return 0;
}
```

Tipps zum Debugging

- In der TTY-Konsole können mit dem Kommando **debug {1|2|3}** detaillierte Informationen zur Analyse und Verarbeitung einer Query angezeigt werden.
- Bei der Implementierung von Stromoperatoren können schnell Speicherlöcher (ungenutzer Speicher wird nicht mehr frei gegeben) entstehen.
- Besonders schlimm ist es, wenn derselbe Zeiger versehentlich mehrfach gelöscht wird, da der Speicher mittlerweile schon anderweitig benutzt werden könnte.
- Das Setzen der Umgebungsvariablen **MALLOC_CHECK_=2** sorgt im obigen Fall für einen sofortigen Programmabbruch (Nur unter Linux möglich).

Noch Fragen ?!