Relationen-Algebra und Persistenz – Teil II

Implementierungskonzepte für und Anforderungen an Tupelstromoperatoren

Fabio Valdés

FernUniversität Hagen

05. Oktober 2012

Inhalt

1. Grundlegende Datentypen der RelationAlgebra
2. Grundlegende Operationen der RelationAlgebra
3. Tupel
4. Referenzzähler
5. Komplexe Type Mappings
6. Hinweise zur Fehlerbehebung

Inhalt

1 Grundlegende Datentypen der RelationAlgebra

2 Grundlegende Operationen der RelationAlgebra

3 Tupel

4 Referenzzähler

5 Komplexe Type Mappings

6 Hinweise zur Fehlerbehebung

Grundlegende Datentypen

```
             Name: tuple
        Signature: (IDENT x DATA)+ -> TUPLE
Example Type List: (tuple((name string)(age int)))
         List Rep: (<attr1> ... <attrn>)
     Example List: ("Roman" 32)

             Name: rel
        Signature: TUPLE -> REL
Example Type List: (rel(tuple((name string)(age int))))
         List Rep: (<tuple>*) where
                   <tuple> is (<attr1> ... <attrn>)
     Example List: (("Roman" 32)("Marco" 23))
```

Grundlegende Datentypen

```
              Name: tuple
         Signature: (IDENT x DATA)+ -> TUPLE
Example Type List: (tuple((name string)(age int)))
          List Rep: (<attr1> ... <attrn>)
      Example List: ("Roman" 32)

              Name: rel
         Signature: TUPLE -> REL
Example Type List: (rel(tuple((name string)(age int))))
          List Rep: (<tuple>*) where
                    <tuple> is (<attr1> ... <attrn>)
      Example List: (("Roman" 32)("Marco" 23))
```

Kinds

Grundlegende Datentypen

```
             Name: tuple
        Signature: (IDENT x DATA)+ -> TUPLE
Example Type List: (tuple((name string)(age int)))
         List Rep: (<attr1> ... <attrn>)
     Example List: ("Roman" 32)

             Name: rel
        Signature: TUPLE -> REL
Example Type List: (rel(tuple((name string)(age int))))
         List Rep: (<tuple>*) where
                   <tuple> is (<attr1> ... <attrn>)
     Example List: (("Roman" 32)("Marco" 23))
```

Typkonstruktoren

Inhalt

1 Grundlegende Datentypen der RelationAlgebra

2 Grundlegende Operationen der RelationAlgebra

3 Tupel

4 Referenzzähler

5 Komplexe Type Mappings

6 Hinweise zur Fehlerbehebung

Grundlegende Operationen I

```
     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
```

Grundlegende Operationen I

```
     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
```

Grundlegende Operationen II

```
     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.
  Example: ...[fun(t:STREAMELEM) attr(t,population) > 500000]...
```

Grundlegende Operationen II

```
     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.
  Example: ...[fun(t:STREAMELEM) attr(t,population) > 500000]...
```

Grundlegende Operationen II

```
     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.
  Example: ...[fun(t:STREAMELEM) attr(t,population) > 500000]...
```

Übersetzung Query in Listendarstellung

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

Übersetzung Query in Listendarstellung

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

(query
    (consume
        (filter
            (feed cities)
            (fun
                (streamelem1 STREAMELEM)
                (>
                    (attr streamelem1 population)
                    500000)))))
```

Übersetzung Query in Listendarstellung

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

(query
    (consume
        (filter
            (feed cities)
            (fun
                (streamelem1 STREAMELEM)
                (>
                    (attr streamelem1 population)
                    500000)))))
```

.spec-File

- Verwendung von Argumentfunktionen mit impliziten Parametern und des Typabbildungsoperators STREAMELEM:

```
operator filter alias FILTER
  pattern _ op [ fun ]
  implicit parameter streamelem type STREAMELEM !!
```

.spec-File

- Verwendung von Argumentfunktionen mit impliziten Parametern und des Typabbildungsoperators STREAMELEM:

```
operator filter alias FILTER
  pattern _ op [ fun ]
  implicit parameter streamelem type STREAMELEM !!
```

- Präfix für Typvariable

.spec-File

- Verwendung von Argumentfunktionen mit impliziten Parametern und des Typabbildungsoperators `STREAMELEM`:

```
operator filter alias FILTER
  pattern _ op [ fun ]
  implicit parameter streamelem type STREAMELEM !!
```

- Präfix für Typvariable
- zu verwendender Typabbildungsoperator

Type Mapping für den Operator `filter`

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

Type Mapping für den Operator `filter`

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

Eingabeliste

```
((stream (tuple ((name string) (population int))))
  (map (tuple((name string) (population int))) bool))
```

Type Mapping für den Operator `filter`

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

Eingabeliste

```
((stream (tuple ((name string) (population int))))
  (map (tuple((name string) (population int))) bool))
```

Type Mapping für den Operator `filter`

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

Eingabeliste

```
((stream (tuple ((name string) (population int))))
  (map (tuple((name string) (population int))) bool))
```

Rückgabeliste

```
(stream (tuple ((name string) (population int))))
```

Type Mapping für den Operator `filter`

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

Eingabeliste

```
((stream (tuple ((name string) (population int))))
  (map (tuple((name string) (population int))) bool))
```

Rückgabeliste

```
(stream (tuple ((name string) (population int))))
```

Type Mapping für den Operator `filter`

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

Eingabeliste

```
((stream (tuple ((name string) (population int))))
  (map (tuple((name string) (population int))) bool))
```

Rückgabeliste

```
(stream (tuple ((name string) (population int))))
```

Bildschirmausgabe (falls aktiv)

```
filter (algId=3, opId=5) accepted!
```

Inhalt

1 Grundlegende Datentypen der RelationAlgebra

2 Grundlegende Operationen der RelationAlgebra

3 Tupel

4 Referenzzähler

5 Komplexe Type Mappings

6 Hinweise zur Fehlerbehebung

Motivation

- bisher behandelt:
  - Attributdatentypen
  - Beispiele für Tupelstromoperatoren

Motivation

- bisher behandelt:
  - Attributdatentypen
  - Beispiele für Tupelstromoperatoren
- anschließend:

Motivation

- bisher behandelt:
  - Attributdatentypen
  - Beispiele für Tupelstromoperatoren
- anschließend:
  - Definition Tupel

Motivation

- bisher behandelt:
  - Attributdatentypen
  - Beispiele für Tupelstromoperatoren
- anschließend:
  - Definition Tupel
  - Erzeugen von Tupeln

Motivation

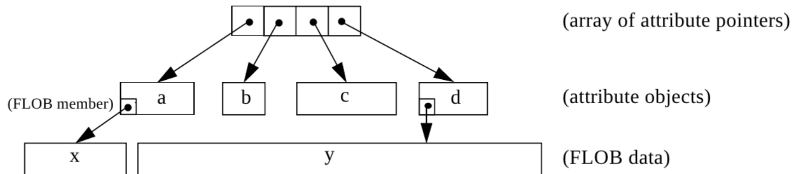
- bisher behandelt:
  - Attributdatentypen
  - Beispiele für Tupelstromoperatoren
- anschließend:
  - Definition Tupel
  - Erzeugen von Tupeln
  - Löschen von Tupeln

Darstellung

- Repräsentation durch ein Array von Zeigern auf Attributdatentyp-Objekte

Darstellung

- Repräsentation durch ein Array von Zeigern auf Attributdatentyp-Objekte



Konstruktoren

Konstruktoren

```
Tuple(TupleType* tupleType)
```

Konstruktoren

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

Konstruktoren

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

Zugriff auf Attribute

```
Attribute* GetAttribute(int index)
```

Konstruktoren

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

Zugriff auf Attribute

```
Attribute* GetAttribute(int index)
void PutAttribute(int index, Attribute* attr)
```

Konstruktoren

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

Zugriff auf Attribute

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

Konstruktoren

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

Zugriff auf Attribute

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

Beispiel: Operator `project`

```
     Name: project
Signature: ((stream (tuple ((a1 T1) ... (an Tn)))) (ai1 ... aik))
           -> (stream (tuple ((ai1 Ti1) ... (aik Tik))))
   Syntax: _ project [ list ]
  Meaning: Produces a projection tuple for
           each tuple of its input stream.
  Example: query cities feed project[cityname, population] count
```

Beispiel: Operator `project`

```
     Name: project
Signature: ((stream (tuple ((a1 T1) ... (an Tn)))) (ai1 ... aik))
           -> (stream (tuple ((ai1 Ti1) ... (aik Tik))))
   Syntax: _ project [ list ]
  Meaning: Produces a projection tuple for
           each tuple of its input stream.
  Example: query cities feed project[cityname, population] count
```

Beispiel: Operator `project`

```
     Name: project
Signature: ((stream (tuple ((a1 T1) ... (an Tn)))) (ai1 ... aik))
           -> (stream (tuple ((ai1 Ti1) ... (aik Tik))))
   Syntax: _ project [ list ]
  Meaning: Produces a projection tuple for
           each tuple of its input stream.
  Example: query cities feed project[cityname, population] count
```

Beispiel: Operator `project`

```
     Name: project
Signature: ((stream (tuple ((a1 T1) ... (an Tn)))) (ai1 ... aik))
           -> (stream (tuple ((ai1 Ti1) ... (aik Tik))))
   Syntax: _ project [ list ]
  Meaning: Produces a projection tuple for
           each tuple of its input stream.
  Example: query cities feed project[cityname, population] count
```

Beispiel: Operator `project`

```
     Name: project
Signature: ((stream (tuple ((a1 T1) ... (an Tn)))) (ai1 ... aik))
           -> (stream (tuple ((ai1 Ti1) ... (aik Tik))))
   Syntax: _ project [ list ]
  Meaning: Produces a projection tuple for
           each tuple of its input stream.
  Example: query cities feed project[cityname, population] count
```

Value mapping für den Operator `project` I

```
int Project(Word* args, Word& result, int message,
            Word& local, Supplier s) {
  switch(message) {
    case OPEN: {
      ListExpr resultType = GetTupleResultType(s);
      TupleType *tupleType = new TupleType(
                              nl->Second(resultType));
      local.addr = tupleType;
      qp->Open(args[0].addr);
      return 0;
    }
      [...]
  }
}
```

Value mapping für den Operator `project` I

```
int Project(Word* args, Word& result, int message,
            Word& local, Supplier s) {
  switch(message) {
    case OPEN: {
      ListExpr resultType = GetTupleResultType(s);
      TupleType *tupleType = new TupleType(
                                 nl->Second(resultType));
      local.addr = tupleType;
      qp->Open(args[0].addr);
      return 0;
    }
      [...]
  }
}
```

aus dem Operatorknoten `s` wird eine Liste der Form `(tuple ((name string)(population int)))` geholt

Value mapping für den Operator `project` I

```
int Project(Word* args, Word& result, int message,
            Word& local, Supplier s) {
  switch(message) {
    case OPEN: {
      ListExpr resultType = GetTupleResultType(s);
      TupleType *tupleType = new TupleType(
                             nl->Second(resultType));
      local.addr = tupleType;
      qp->Open(args[0].addr);
      return 0;
    }
      [...]
  }
}
```

erzeugt ein Objekt vom Typ `TupleType` aus der Attributliste `nl->Second(resultType)`

Value mapping für den Operator `project` II

```
case REQUEST: {
    [...]
  TupleType *tupleType = (TupleType*)local.addr;
  Tuple *t = new Tuple(tupleType);
  noOfAttrs = ((CcInt*)args[2].addr)->GetIntval();
  assert(t->GetNoAttributes() == noOfAttrs);
  for (int i = 0; i < noOfAttrs; i++) {
      [...]
    index = ((CcInt*)elem2.addr)->GetIntval();
    t->CopyAttribute(index - 1, (Tuple*)elem1.addr, i);
  }
  ((Tuple*)elem1.addr)->DeleteIfAllowed();
    [...]
}
```

Value mapping für den Operator `project` II

```
case REQUEST: {
    [...]
  TupleType *tupleType = (TupleType*)local.addr;
  Tuple *t = new Tuple(tupleType);
  noOfAttrs = ((CcInt*)args[2].addr)->GetIntval();
  assert(t->GetNoAttributes() == noOfAttrs);
  for (int i = 0; i < noOfAttrs; i++) {
      [...]
    index = ((CcInt*)elem2.addr)->GetIntval();
    t->CopyAttribute(index - 1, (Tuple*)elem1.addr, i);
  }
  ((Tuple*)elem1.addr)->DeleteIfAllowed();
    [...]
}
```

gibt die Anzahl der Attribute von `t` zurück

Value mapping für den Operator `project` II

```
case REQUEST: {
    [...]
  TupleType *tupleType = (TupleType*)local.addr;
  Tuple *t = new Tuple(tupleType);
  noOfAttrs = ((CcInt*)args[2].addr)->GetIntval();
  assert(t->GetNoAttributes() == noOfAttrs);
  for (int i = 0; i < noOfAttrs; i++) {
      [...]
    index = ((CcInt*)elem2.addr)->GetIntval();
    t->CopyAttribute(index - 1, (Tuple*)elem1.addr, i);
  }
  ((Tuple*)elem1.addr)->DeleteIfAllowed();
    [...]
}
```

kopiert das `i`-te Attribut von `(Tuple*)elem1.addr` an Position `index - 1` des aktuellen Tupels

Value mapping für den Operator `project` II

```
case REQUEST: {
    [...]
  TupleType *tupleType = (TupleType*)local.addr;
  Tuple *t = new Tuple(tupleType);
  noOfAttrs = ((CcInt*)args[2].addr)->GetIntval();
  assert(t->GetNoAttributes() == noOfAttrs);
  for (int i = 0; i < noOfAttrs; i++) {
      [...]
    index = ((CcInt*)elem2.addr)->GetIntval();
    t->CopyAttribute(index - 1, (Tuple*)elem1.addr, i);
  }
  ((Tuple*)elem1.addr)->DeleteIfAllowed();
    [...]
}
```

löscht das `Tuple`-Objekt inklusive aller enthaltenen Attribute unter Beachtung ggf. noch verwendeter Referenzen

Value mapping für den Operator `project` III

```
case CLOSE: {
  qp->Close(args[0].addr);
  if (local.addr) {
    ((TupleType*)local.addr)->DeleteIfAllowed();
    local.setAddr(0);
  }
  return 0;
}
```

Value mapping für den Operator `project` III

```
case CLOSE: {
  qp->Close(args[0].addr);
  if (local.addr) {
    ((TupleType*)local.addr)->DeleteIfAllowed();
    local.setAddr(0);
  }
  return 0;
}
```

löscht das `TupleType`-Objekt inklusive aller enthaltenen Attribute unter Beachtung ggf. noch verwendeter Referenzen

Inhalt

1 Grundlegende Datentypen der RelationAlgebra

2 Grundlegende Operationen der RelationAlgebra

3 Tupel

4 Referenzzähler

5 Komplexe Type Mappings

6 Hinweise zur Fehlerbehebung

Motivation

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen und Attributen

Motivation

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen und Attributen
- Attributdatentypen (automatische Verwaltung)

Motivation

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen und Attributen
- Attributdatentypen (automatische Verwaltung)
  - `Clone` erzeugt eine Tiefenkopie mit einer Referenz

Motivation

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen und Attributen
- Attributdatentypen (automatische Verwaltung)
  - `Clone` erzeugt eine Tiefenkopie mit einer Referenz
  - `Copy` erzeugt eine weitere Referenz

Motivation

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen und Attributen
- Attributdatentypen (automatische Verwaltung)
  - `Clone` erzeugt eine Tiefenkopie mit einer Referenz
  - `Copy` erzeugt eine weitere Referenz
  - `DeleteIfAllowed` löscht eine Referenz und ggf. auch das Attributobjekt

Motivation

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen und Attributen
- Attributdatentypen (automatische Verwaltung)
  - `Clone` erzeugt eine Tiefenkopie mit einer Referenz
  - `Copy` erzeugt eine weitere Referenz
  - `DeleteIfAllowed` löscht eine Referenz und ggf. auch das Attributobjekt
- Tupel (halbautomatische Verwaltung); Zähler = 1 zu Beginn

Motivation

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen und Attributen
- Attributdatentypen (automatische Verwaltung)
  - `Clone` erzeugt eine Tiefenkopie mit einer Referenz
  - `Copy` erzeugt eine weitere Referenz
  - `DeleteIfAllowed` löscht eine Referenz und ggf. auch das Attributobjekt
- Tupel (halbautomatische Verwaltung); Zähler = 1 zu Beginn
  - `IncReference` erhöht den Zähler um 1

Motivation

- Ziel: Vermeidung teurer Kopien von Tupeln, Tupeltypen und Attributen
- Attributdatentypen (automatische Verwaltung)
  - `Clone` erzeugt eine Tiefenkopie mit einer Referenz
  - `Copy` erzeugt eine weitere Referenz
  - `DeleteIfAllowed` löscht eine Referenz und ggf. auch das Attributobjekt
- Tupel (halbautomatische Verwaltung); Zähler = 1 zu Beginn
  - `IncReference` erhöht den Zähler um 1
  - `DeleteIfAllowed` löscht das Tupel (falls Zähler = 1) oder reduziert den Zähler um 1

Anwendungen

- neue Tupel werden erzeugt: `feed`

Anwendungen

- neue Tupel werden erzeugt: `feed`
- Tupel werden unverändert weitergereicht: **`filter`**

Anwendungen

- neue Tupel werden erzeugt: `feed`
- Tupel werden unverändert weitergereicht: `filter`
- Tupel werden verbraucht / gelöscht: **`consume`**

Anwendungen

- neue Tupel werden erzeugt: `feed`
- Tupel werden unverändert weitergereicht: `filter`
- Tupel werden verbraucht / gelöscht: `consume`
- Tupel werden verändert*: **`project`**

Inhalt

1. Grundlegende Datentypen der RelationAlgebra
2. Grundlegende Operationen der RelationAlgebra
3. Tupel
4. Referenzzähler
5. Komplexe Type Mappings
6. Hinweise zur Fehlerbehebung

Motivation

- Weitergabe (interner) Argumente an Value Mappings, z.B.

Motivation

- Weitergabe (interner) Argumente an Value Mappings, z.B.
  - generische Value-Mapping-Funktionen

Motivation

- Weitergabe (interner) Argumente an Value Mappings, z.B.
  - generische Value-Mapping-Funktionen
  - Übergabe von Attribut-Indizes

Motivation

- Weitergabe (interner) Argumente an Value Mappings, z.B.
  - generische Value-Mapping-Funktionen
  - Übergabe von Attribut-Indizes
- Schlüsselwort `APPEND` im Type Mapping

Motivation

- Weitergabe (interner) Argumente an Value Mappings, z.B.
  - generische Value-Mapping-Funktionen
  - Übergabe von Attribut-Indizes
- Schlüsselwort `APPEND` im Type Mapping
  - bei Rückgabe von
    `(APPEND (v1, ..., vn) <result-type-list>)`
    werden die `vi` als zusätzliche Argumente weitergereicht

Motivation

- Weitergabe (interner) Argumente an Value Mappings, z.B.
  - generische Value-Mapping-Funktionen
  - Übergabe von Attribut-Indizes
- Schlüsselwort `APPEND` im Type Mapping
  - bei Rückgabe von
    `(APPEND (v1, ..., vn) <result-type-list>)`
    werden die `vi` als zusätzliche Argumente weitergereicht
  - die `vi` können selbst Listen sein

Beispiel: Operator `attr` I

```
ListExpr AttrTypeMap(ListExpr args) {
  if (nl->ListLength(args) != 2) {
    return listutils::typeError("two arguments expected");
  }
  ListExpr first = nl->First(args);
  if (!Tuple::checkType(first)) {
    return listutils::typeError
           ("first argument must be tuple(...)");
  }
  ListExpr second = nl->Second(args);
  if (!listutils::isSymbol(second)) {
    return listutils::typeError
           ("second argument must be an attribute name");
  }
```

Beispiel: Operator `attr` I

```
ListExpr AttrTypeMap(ListExpr args) {
  if (nl->ListLength(args) != 2) {
    return listutils::typeError("two arguments expected");
  }
  ListExpr first = nl->First(args);
  if (!Tuple::checkType(first)) {
    return listutils::typeError
           ("first argument must be tuple(...)");
  }
  ListExpr second = nl->Second(args);
  if (!listutils::isSymbol(second)) {
    return listutils::typeError
           ("second argument must be an attribute name");
  }
```

Beispiel: Operator `attr` I

```
ListExpr AttrTypeMap(ListExpr args) {
  if (nl->ListLength(args) != 2) {
    return listutils::typeError("two arguments expected");
  }
  ListExpr first = nl->First(args);
  if (!Tuple::checkType(first)) {
    return listutils::typeError
         ("first argument must be tuple(...)");
  }
  ListExpr second = nl->Second(args);
  if (!listutils::isSymbol(second)) {
    return listutils::typeError
         ("second argument must be an attribute name");
  }
```

Beispiel: Operator `attr` I

```
ListExpr AttrTypeMap(ListExpr args) {
  if (nl->ListLength(args) != 2) {
    return listutils::typeError("two arguments expected");
  }
  ListExpr first = nl->First(args);
  if (!Tuple::checkType(first)) {
    return listutils::typeError
           ("first argument must be tuple(...)");
  }
  ListExpr second = nl->Second(args);
  if (!listutils::isSymbol(second)) {
    return listutils::typeError
           ("second argument must be an attribute name");
  }
```

Beispiel: Operator attr II

```
  string name = nl->SymbolValue(second);
  ListExpr attrtype;
  int j = listutils::findAttribute
                    (nl->Second(first), name, attrtype);
  if (j == 0) {
    return listutils::typeError
          ("Attr name " + name + " not found in attribute list");
  }
  return nl->ThreeElemList(nl->SymbolAtom(Symbol::APPEND()),
                          nl->OneElemList(nl->IntAtom(j)),
                          attrtype);
}
```

Beispiel: Operator `attr` II

```
  string name = nl->SymbolValue(second);
  ListExpr attrtype;
  int j = listutils::findAttribute
                   (nl->Second(first), name, attrtype);
  if (j == 0) {
    return listutils::typeError
          ("Attr name " + name + " not found in attribute list");
  }
  return nl->ThreeElemList(nl->SymbolAtom(Symbol::APPEND()),
                           nl->OneElemList(nl->IntAtom(j)),
                           attrtype);
}
```

gibt den Datentyp `attrtype` und den Index `j` zum Attributnamen `name` in der Attributliste `nl->Second(first)` zurück

Beispiel: Operator `attr` II

```
  string name = nl->SymbolValue(second);
  ListExpr attrtype;
  int j = listutils::findAttribute
                             (nl->Second(first), name, attrtype);
  if (j == 0) {
    return listutils::typeError
           ("Attr name " + name + " not found in attribute list");
  }
  return nl->ThreeElemList(nl->SymbolAtom(Symbol::APPEND()),
                                nl->OneElemList(nl->IntAtom(j)),
                                attrtype);
}
```

sendet Fehlerbericht, falls `name` nicht in der Attributliste vorkommt

Beispiel: Operator `attr` II

```
  string name = nl->SymbolValue(second);
  ListExpr attrtype;
  int j = listutils::findAttribute
                   (nl->Second(first), name, attrtype);
  if (j == 0) {
    return listutils::typeError
        ("Attr name " + name + " not found in attribute list");
  }
  return nl->ThreeElemList(nl->SymbolAtom(Symbol::APPEND()),
                           nl->OneElemList(nl->IntAtom(j)),
                           attrtype);
}
```

gibt die Datentypbeschreibung `attrtype` zurück; angehängt ist eine Liste mit dem Attributindex

Beispiel: Operator `attr` II

```
  string name = nl->SymbolValue(second);
  ListExpr attrtype;
  int j = listutils::findAttribute
                  (nl->Second(first), name, attrtype);
  if (j == 0) {
    return listutils::typeError
          ("Attr name " + name + " not found in attribute list");
  }
  return nl->ThreeElemList(nl->SymbolAtom(Symbol::APPEND()),
                           nl->OneElemList(nl->IntAtom(j)),
                           attrtype);
}
```

gibt die Datentypbeschreibung `attrtype` zurück; angehängt ist eine Liste mit dem Attributindex

Beispiel: Operator `attr` II

```
  string name = nl->SymbolValue(second);
  ListExpr attrtype;
  int j = listutils::findAttribute
                   (nl->Second(first), name, attrtype);
  if (j == 0) {
    return listutils::typeError
        ("Attr name " + name + " not found in attribute list");
  }
  return nl->ThreeElemList(nl->SymbolAtom(Symbol::APPEND()),
                           nl->OneElemList(nl->IntAtom(j)),
                           attrtype);
}
```

gibt die Datentypbeschreibung `attrtype` zurück; angehängt ist eine Liste mit dem Attributindex

Type Mapping für Operator `attr`

```
Eingabe: ((tuple ((name string) (population int))) name)
Rückgabe: (APPEND (1) string)
```

Value Mapping für Operator `attr`

```
int Attr(Word* args, Word& result,
         int message, Word& local, Supplier s) {
  Tuple* tupleptr = (Tuple*)args[0].addr;
  int index = ((CcInt*)args[2].addr)->GetIntval();
  assert(1 <= index && index <= tupleptr->GetNoAttributes());
  result.setAddr(tupleptr->GetAttribute(index - 1));
  return 0;
}
```

Type Mapping für Operator `attr`

```
Eingabe: ((tuple ((name string) (population int))) name)
Rückgabe: (APPEND (1) string)
```

Value Mapping für Operator `attr`

```
int Attr(Word* args, Word& result,
         int message, Word& local, Supplier s) {
  Tuple* tupleptr = (Tuple*)args[0].addr;
  int index = ((CcInt*)args[2].addr)->GetIntval();
  assert(1 <= index && index <= tupleptr->GetNoAttributes());
  result.setAddr(tupleptr->GetAttribute(index - 1));
  return 0;
}
```

Type Mapping für Operator `attr`

```
Eingabe: ((tuple ((name string) (population int))) name)
Rückgabe: (APPEND (1) string)
```

Value Mapping für Operator `attr`

```
int Attr(Word* args, Word& result,
         int message, Word& local, Supplier s) {
  Tuple* tupleptr = (Tuple*)args[0].addr;
  int index = ((CcInt*)args[2].addr)->GetIntval();
  assert(1 <= index && index <= tupleptr->GetNoAttributes());
  result.setAddr(tupleptr->GetAttribute(index - 1));
  return 0;
}
```

- `args[1]`, d.h. der vom Benutzer angegebene Attributname, wird nicht mehr benötigt
- im Type Mapping wurde `args[2]` (Attributindex) aus `args[1]` berechnet

Value Mapping für Operator `attr`

```
int Attr(Word* args, Word& result,
         int message, Word& local, Supplier s) {
  Tuple* tupleptr = (Tuple*)args[0].addr;
  int index = ((CcInt*)args[2].addr)->GetIntval();
  assert(1 <= index && index <= tupleptr->GetNoAttributes());
  result.setAddr(tupleptr->GetAttribute(index - 1));
  return 0;
}
```

Inhalt

1 Grundlegende Datentypen der RelationAlgebra

2 Grundlegende Operationen der RelationAlgebra

3 Tupel

4 Referenzzähler

5 Komplexe Type Mappings

6 Hinweise zur Fehlerbehebung

Hinweise zur Fehlerbehebung

- Anzeige zur Analyse und Verarbeitung von Anfragen mit `debug{1|2|3}` (in der TTY-Konsole)

Hinweise zur Fehlerbehebung

- Anzeige zur Analyse und Verarbeitung von Anfragen mit `debug{1|2|3}` (in der TTY-Konsole)
- Anzeige von Speicherlöchern (ungenutzter und nicht wieder freigegebener Speicher) durch `SecondoTTYBDB --valgrindlc`

Hinweise zur Fehlerbehebung

- Anzeige zur Analyse und Verarbeitung von Anfragen mit `debug{1|2|3}` (in der TTY-Konsole)
- Anzeige von Speicherlöchern (ungenutzter und nicht wieder freigegebener Speicher) durch
  `SecondoTTYBDB --valgrindlc`
- Fehlerquelle: wiederholtes Löschen von Zeigern, der Speicher könnte bereits anderweitig belegt sein;
  Empfehlung: nicht verwendete Zeiger auf 0 setzen

Hinweise zur Fehlerbehebung

- Anzeige zur Analyse und Verarbeitung von Anfragen mit `debug{1|2|3}` (in der TTY-Konsole)
- Anzeige von Speicherlöchern (ungenutzter und nicht wieder freigegebener Speicher) durch
  `SecondoTTYBDB --valgrindlc`
- Fehlerquelle: wiederholtes Löschen von Zeigern, der Speicher könnte bereits anderweitig belegt sein;
  Empfehlung: nicht verwendete Zeiger auf 0 setzen
  - unter Linux: Setzen der Umgebungsvariablen `MALLOC_CHECK=2` führt in diesem Fall zu sofortigem Programmabbruch

Fragerunde

Jetzt ist Zeit für Ihre Fragen und Anmerkungen.

Fragerunde

Vielen Dank für Ihre Aufmerksamkeit und viel Erfolg beim Fachpraktikum.