How to Implement an Algebra in Secondo

The GuideAlgebra

Thomas Behr 2014

Last Change: 2015-10-07

Contents

1 Preface

This document describes how to integrate a new algebra into the Secondo DBMS. An algebra consists of types and operators.

It starts with a description of nested lists, a structure for representing objects externally and types internally. The implementation of different kinds of types are shown, these include

- simple types
- attribute data types of fixed size
- attribute data type of variable size
- large types, e.g., for implementing indexes

Some special sections also describe how to reduce the amount of disc storage required for an attribute data type.

An Operator consists of a type mapping, a set of value mappings, a selection function, and a description for the user. The type mapping checks whether the operator can handle a given set of argument types. The value mapping computes the result of the operator from the arguments. The selection function selects a value mapping for overloaded operators. Operators can occur in a lot of ways. This document describes

- simple operators
- overloaded operators
- stream processing operators

- operators having a function as an argument
- update operators
- operators where results from the type mapping are transferred to the value mapping
- operators having access to values within the type mapping

1.1 The PD-System

All files in Secondo are commented using comments in PD-style. The PD tool allows the creation of pretty formatted PDF documents from source files. For a complete documentation of this tool, read the file PDSystem1.pdf located in the *Documents* directory of Secondo.

If the PD system is running, a single pdf document from all files within the directory of this algebra can be created by entering *make doc* in the algebra directory.

1.2 Nested Lists

Because nested lists are used very often in Secondo, this document starts with a short description of this structure. Nested lists are required to import/export objects, within type mappings, selection functions, display functions and much more. After reading this document, the meaning of these things will be clear.

Each element of a nested list can be an atom or a nested list again. Atomar elements are the following:

```
Int (32 bit)
Real
Bool
String (maximum 48 characters)
Symbol (maximum 48 characters)
Text   (a string of arbitrary length)
```

For storing 64 bit integer values, the integer must be separated into a list consisting of two 32-bit integer values. The file *ListUtils.h* provides functions for encoding and decoding 64 bit integers.

Functions manipulating nested lists use a global list storage called **nl**. This list storage is cleaned after each query. This disables the possibility to store some frequently used list as static variables.

For using the global nested list storage, the following lines within the program code are required:

```
#include "NestedList.h"
extern NestedList* nl;
```

The variable **nl** is a singleton pattern defined somewhere in the system.

For the creation of a list atom, the following functions are available:

```
ListExpr li = nl->IntAtom(23);
ListExpr lr = nl->RealAtom(-8.3);
ListExpr lsy = nl->SymbolAtom("Symbol");
  ...
```

For creating short lists, several functions are provided.

```
ListExpr l1 = nl->OneElemList(li);
ListExpr l2 = nl->TwoElemList(li,lr);
  ...
ListExpr l6 = nl->SixElemList(li,lr,lsy,l1,l2,li);
bool correct = nl->ReadFromString("a 'b' 1.0 (TRUE 6))", list);
```

The return value of *ReadFromString* determines the success of this operation, in particular, whether the string represents a valid nested list. The list itself is returned via the output parameter *list*.

For the creation of long lists, the following code fragment can be used as a template:

```
ListExpr l1000 = nl->OneElemList(nl->IntAtom(0));
ListExpr last = l1000;
for(int i=1; i<1000; i++){
   last = nl->Append(last,nl->IntAtom(i));
}
```

The function *Append* takes the last element of a list and append its second argument to this list. The new last element is returned.

A nested list can be read from a file using the *ReadFromFile* function. This function works quite similar to *ReadFromString*. The input is not the nested list as a string, but the name of a file containing the list.

```
bool correct = nl->ReadFromString("mylist.txt", list);
```

To request the type of a given list, the function *AtomType* ist used:

```
int r = nl->AtomType(list);
```

This function returns one of the predefined values *NoAtom*, *IntType*, *RealType*, *BoolType*, *StringType*, *SymbolType*, or *TextType*.
The value of an atom can be taken by:

```
int i = nl->IntValue(list);
string s = nl->SymbolValue(list);
....
string t = nl->Text2String(list);
```

Note that the try to get the value of wrong type, e.g. calling

---

```
int k = nl->IntValue(list);
```

---

where *list* is a string, will crash the system by throwing an assertion.
The following functions provide information about the length of a list:

```
nl->IsEmpty(list);
nl->ListLength(list); // -1 for an atom
nl->hasLength(list,3); // check list to have length 3
```

A list can be separated by:

```
nl->First(list);
nl->Second(list);
...
nl->Sixth(list);

nl->Rest(list); // returns list without the first element
```

Two lists can be checked for equality and a single list can be checked to be a symbol atom having a certain value (the third argument is for case sensitivity) using the functions:

```
bool eq = nl->Equal(list1,list2);
bool isSym = nl->IsEqual(list, "symbol", false);
```

For debugging purposes, sometimes an output of a list is required. This can be done by:

```
cout << nl->ToString(list) << endl;
```

Within *ListUtils.h* a collection of auxiliary functions are implemented, to make the life with nested lists easier, e.g.:

```
#include "NestedList.h"
#include "ListUtils.h"

// example: read in an integer or a real from list
if( listutils::isNumeric(list)){
  double d = listutils::getNumValue(list);
}
```

Note: the nested list implementation contains a lot of assertions. Thereby any wrong usage of these lists leads to a crash of Secondo. Before a function is called, always the correctness of the list's format,e.g. length and type, has to be checked.

A more objectoriented interface to nested list is provided in NList.h. Here, exceptions instead of assertions are used to handle wrong calls. If you want to use this interface, please take a look to the file *NList.h* within the *include* directory of Secondo.

1.3 What is an Algebra

An algebra contains types and/or operators. It uses the types defined in the algebra itself and –if required– types of other algebras. Secondo provides a well defined interface for integrating a new algebra. Each activated algebra is linked together with the kernel of Secondo. Each type and each operator extends Secondo's executable language.

1.4 Preliminary Steps

For creating a new algebra, some steps are required. Firstly, create a new subdirectory having the algebra's name within the *Algebra* directory of Secondo.

After that, insert a new entry in the file *Algebras/Management/AlgebraList.i.cfg* having the format:

```
ALGEBRA_INCLUDE(<Number>, <Name>)
```

Note that number and name must be different to all existing entries.

Now, activate the algebra by modifying the file *makefile.algebras* in Secondo's main directory. Insert the two lines:

```
ALGEBRA_DIRS += <DirectoryName>
ALGEBRAS += <AlgebraName>
```

---

in the file. If the algebra uses third party libraries, add the line:

```
ALGEBRA_DEPS += <libname>
```

---

in the file. If the library is not stored within the standard directories, add the line:

```
ALGEBRA_DEP_DIRS += <directory>
```

If the library required special flags for linking, add the line

```
ALGEBRA_LINK_FLAGS += <flags>
```

After adding the required entries to the files, go back to the newly created algebra directory. Copy the *makefile* of the *StandardAlgebra* into this directory. If there a dependencies to other algebras, modify the copyied file by inserting the lines:

```
CURRENT_ALGEBRA := <AlgebraName>
ALGEBRA_DEPENDENCIES := StandardAlgebra
ALGEBRA_DEPENDENCIES += <AlgebraName>
...
```

after the line

```
include ../../makefile.env
```

This provides understandable error messages if one of the required algebras is not activated. Without these entries a required but non-activated algebra will lead to strange error messages during linking the system.

After these steps, the algebra file(s) can be implemented. It is not required to implement an algebra within a single file. The algebra files can be split into header and implementation files as usual in C++. This algebra is implemented within a single file for easy creating a single documentation file.

1.5 Includes

To be able to implement an algebra, different header files must be included. Here, each include is commented with some functionality used.

```
#include "Attribute.h"          // implementation of attribute types
#include "Algebra.h"            // definition of the algebra
#include "NestedList.h"         // required at many places
#include "QueryProcessor.h"     // needed for implementing value mappings
#include "AlgebraManager.h"     // e.g., check for a certain kind
#include "Operator.h"           // for operator creation
#include "StandardTypes.h"      // priovides int, real, string, bool type
#include "FTextAlgebra.h"
#include "Symbols.h"            // predefined strings
#include "ListUtils.h"          // useful functions for nested lists
#include "Stream.h"             // wrapper for secondo streams

#include "GenericTC.h"          // use of generic type constructors

#include "LogMsg.h"             // send error messages

#include "../../Tools/Flob/DbArray.h"  // use of DbArrays

#include "RelationAlgebra.h"           // use of tuples


#include <math.h>               // required for some operators
#include <stack>
#include <limits>
```

1.6 Global Variables

Secondo uses some variables designed as singleton pattern. For accessing these global variables, these variables have to be declared to be extern:

```
extern NestedList *nl;
extern QueryProcessor *qp;
extern AlgebraManager *am;
```

1.7 Namespace

Each algebra file defines a lot of functions. Thus, name conflicts may arise with function names defined in other algebra modules during compiling/linking the system. To avoid these conflicts, the algebra implementation should be embedded into a namespace.

```
namespace guide{
```

2 Implementation of a simple Type

In the next section, the creation of a simple (non-attribute) type and it's integration into a Secondo algebra is described. A value of such an object can be stored and accessed within a Secondo database but cannot be part of a relation. In general, a class encapsulating the type and a set of functions must be implemented. These functions are

- Description of the type
- Import, Export
- Creation, Deletion
- Open, Close, Save
- Clone
- Cast
- required storage size
- type check

Using these function, an instance of a type constructor is created. This type constructor is added to the algebra within the algebra constructor. Later, the user interfaces can be extended for a pretty output of the newly created type. The extension of user interfaces is out of the scope of this document.

All these steps are now described on an example.

2.1 The Class

This class encapsulates the Secondo type within C++. The class must provide one constructor doing nothing. This constructor is later used within the cast function and should be called only in the cast function.

For an easy use of this class in Secondo, two additional functions are implemented, namely *BasicType* and *checkType*. The *BasicType* function return Secondo's basic type for this class. We call the type of the example *scircle* (standing for simple circle). For non-nested types, a string holding the type's name is the result of the *BasicType* function. For more complex types, e.g. *(rel(tuple(...)))*, this function returns only the main type, e.g. *rel*. The function *checkType* takes a nested list and checks whether this list represents a valid type description for this class. Note that the *checkType* function not only checks the main type but the complete type expression, e.g. for relations, *checkType(rel)* will return *false* while *checktype(rel(tuple((A int)(B string))))* will return *true*.

All other functions and members are usual C++ stuff.

```
class SCircle{
    public:
      // constructor doing nothing
      SCircle() {}
      // constructor initializing the object
      SCircle(const double _x, const double _y, const double _r):
         x(_x), y(_y), r(_r) {}
      // destructor
      ~SCircle(){}
      static const string BasicType(){ return "scircle";}
      // the checktype function for non-nested types looks always
      // the same
      static const bool checkType(const ListExpr list) {
```

```
        return listutils::isSymbol(list, BasicType());
      }

      double perimeter() const{
        return 2*M_PI*r;
      }
      double getX() const{ return x; }
      double getY() const{ return y; }
      double getR() const{ return r; }
    private:
      double x;
      double y;
      double r;
};
```

2.2 The Property Function

The *Property* function provides a description of the Secondo type to the user. It returns a nested list, which must be have exactly the format given in this example. The first element of the list is always the same and the second element of the list contains type specific descriptions.

```
ListExpr SCircleProperty() {
  return ( nl->TwoElemList (
      nl->FourElemList (
        nl->StringAtom("Signature"),
        nl->StringAtom("Example Type List"),
        nl->StringAtom("List Rep"),
        nl->StringAtom("Example List")),
      nl->FourElemList (
        nl->StringAtom("-> SIMPLE"),
        nl->StringAtom(SCircle::BasicType()),
        nl->StringAtom("(real real real) = (x,y,r)"),
        nl->StringAtom("(13.5 -76.0 1.0)")
      )));
}
```

2.3 In Function

For the creation of a constant value within a query and for importing objects or whole databases from a file, object values are described by nested lists. The task of the *IN*-function is to convert such a list into the internal object representation, i.e. into an instance of the class above. The list may have an invalid format. If the list does not have the expected format, the output parameter *correct* must be set to *false* and the *addr*-pointer of the result must be set to 0. A detailed error description can be provided to the user by calling the *inFunError* of the global *cmsg* object. In case of success, the argument *correct* has to be set to *true* and the *addr* pointer of the result points to an object instance having the value represented by the *instance* argument. The parameters of the function are:

- *typeInfo*: contains the complete type description and is required for nested types like tuples
- *instance*: the value of the object in external (nested list) representation

- *errorPos*: output parameter reporting the position of an error within the list (set types)
- *errorInfo*: can provide information about an error to the user
- *correct*: output parameter returning the success of this call

For the *scircle* class, the external representation consists of three numeric values standing for the x position, the y position and the radius of the circle. The radius must be greater than zero.

```
Word InSCircle( const ListExpr typeInfo, const ListExpr instance,
     const int errorPos, ListExpr& errorInfo, bool& correct ){
   // create a result with addr pointing to 0
   Word res((void*)0);
   // assume an incorrect list
   correct = false;
   // check whether the list has three elements
   if(!nl->HasLength(instance,3)){
     cmsg.inFunError("expected three numbers");
     return res;
   }
   // check whether all elements are numeric
   if(   !listutils::isNumeric(nl->First(instance))
      || !listutils::isNumeric(nl->Second(instance))
      || !listutils::isNumeric(nl->Third(instance))){
      cmsg.inFunError("expected three numbers");
      return res;
   }
   // get the numeric values of the elements
   double x = listutils::getNumValue(nl->First(instance));
   double y = listutils::getNumValue(nl->Second(instance));
   double r = listutils::getNumValue(nl->Third(instance));
   // check for a valid radius
   if(r<=0){
      cmsg.inFunError("invalid radius (<=0)");
      return res;
   }
   // list was correct,  create the result
   correct = true;
   res.addr = new SCircle(x,y,r);
   return res;
 }
```

2.4 Out Function

This function is used to create the external representation of an object as nested list. Note that the *IN* function must be able to read in the result of this function. The arguments are:

- *typeInfo*: nested list representing the type of the object (required for complex types)
- *value*: the *addr* pointer of *value* points to the object to export. The Secondo framework ensures that the type of this object is the correct one. The cast in the first line will be successful.

This function must be able to convert **each** instance into a nested list. For this reason, there is no function for error reporting as in the *IN* function.

```
ListExpr OutSCircle( ListExpr typeInfo, Word value ) {
   SCircle* k = (SCircle*) value.addr;
     return nl->ThreeElemList(
                nl->RealAtom(k->getX()),
                nl->RealAtom(k->getY()),
                nl->RealAtom(k->getR()));
}
```

2.5 Create Function

This function creates an object instance having an arbitrary value. The *typeInfo* argument represents the type of the object and is required for nested types like tuples.

```
Word CreateSCircle( const ListExpr typeInfo ) {
  Word w;
  w.addr = (new SCircle(0,0,1.0));
  return w;
}
```

2.6 Delete Function

Removes the complete object (inclusive disc parts if there are any, see section 12). The Secondo framework ensures that the type behind the *addr* pointer of *w* is the expected one. The arguments are:

- *typeInfo*: the type description (for complex types)
- *w*: the *addr* pointer of this argument points to the object to delete.

```
void DeleteSCircle( const ListExpr typeInfo, Word& w ) {
  SCircle *k = (SCircle *)w.addr;
  delete k;
  w.addr = 0;
}
```

2.7 Open Function

Reads an object from disc via an *SmiRecord*.

- *valueRecord*: here, the disc representation of the object is stored
- *offset*: the object representation starts here in *valueRecord* After the call of this function, *offset* must be after the object's value
- *typeInfo*: the type description (required for complex types)
- *value*: output argument

The function reads data out of the SmiRecord and creates a new object from them in case of success. The created object is stored in the *addr*-pointer of the *value* argument. In the case of an error, the *addr* pointer has to be set to *NULL*. The result of this functions reports the success of reading. To implement this function, the function *Read* of the *SmiRecord* is used. Its first argument is a pointer to the storage where the data should be written. The second argument determines how may data should be transferred from the record to the buffer. The last argument indicates the position of the data within the record. The return value of this function corresponds to the actual read amount of data. In case of success, this number is the same as given in the second argument.

```
bool OpenSCircle( SmiRecord& valueRecord,
                  size_t& offset, const ListExpr typeInfo,
                  Word& value ){
  size_t size = sizeof(double);
  double x,y,r;
  bool ok = (valueRecord.Read(&x,size,offset) == size);
  offset += size;
  ok = ok && (valueRecord.Read(&y,size,offset) == size);
  offset += size;
  ok = ok && (valueRecord.Read(&r, size, offset) == size);
  offset += size;
  if(ok){
    value.addr = new SCircle(x,y,r);
  } else {
    value.addr = 0;
  }
  return ok;
}
```

2.8 Save Function

Saves an object to disc (via SmiRecord). This function has to be symmetrically to the *OPEN* function. The result reports the success of the call. The arguments are

- *valueRecord*: here the object will be stored
- *offset*: the object has to be stored at this position in *valueRecord*; after the call of this function, *offset* must be after the object's representation
- *typeInfo*: type description as a nested list (required for complex types)
- *value*: the addr pointer of this argument points to the object to save

The used *Write* function of the *SmiRecord* works similar to its *Read* function but transfers the data into the other direction.

```
bool SaveSCircle( SmiRecord& valueRecord, size_t& offset,
                  const ListExpr typeInfo, Word& value ) {
   SCircle* k = static_cast<SCircle*>( value.addr );
   size_t size = sizeof(double);
   double v = k->getX();
   bool ok = valueRecord.Write( &v, size, offset );
   offset += size;
```

```
  v = k->getY();
  ok = ok && valueRecord.Write(&v,size,offset);
  offset += size;
  v = k->getR();
  ok = ok && valueRecord.Write(&v,size,offset);
  offset += size;
  return ok;
}
```

2.9 Close Function

Removes the main memory part of an object. In contrast to delete, the disc part of the object is untouched (if there is one).

- *typeInfo*: type description as a nested list
- w: the *addr* pointer of w points to the object which is to close

```
void CloseSCircle( const ListExpr typeInfo, Word& w ) {
  SCircle *k = (SCircle *)w.addr;
  delete k;
  w.addr = 0;
}
```

2.10 Clone Function

Creates a depth copy (inclusive disc parts) of an object.

- *typeInfo*: type description as nested list
- w: holds a pointer to the object which is to clone

```
Word CloneSCircle( const ListExpr typeInfo, const Word& w ){
     SCircle* k = (SCircle*) w.addr;
     Word res;
     res.addr = new SCircle(k->getX(), k->getY(), k->getR());
     return res;
}
```

2.11 Cast Function

Casts a void pointer to the type using a special call of new operator. The argument points to a memory block which is to cast to the object. The used C++ constructor cannot initialize the object, e.g. the used constructur must do nothing.

```
void*  CastSCircle( void* addr ) {
    return (new (addr) SCircle);
}
```

2.12 Type Check

Checks whether a given list corresponds to the type. This function is quit similar to the *checkType* function within the class. The result is *true* if *type* represents a valid type description for the type, *false* otherwise. The argument *errorInfo* can be used to report an error message to the user.

```
bool SCircleTypeCheck(ListExpr type, ListExpr& errorInfo){
  return nl->IsEqual(type, SCircle::BasicType());
}
```

2.13 SizeOf Function

Returns the size required to store an instance of this object to disc using the *Save* function from above. Because an *scircle* is represented by three double numbers, the size is three times the size of a single double.

```
int SizeOfSCircle() {
    return 3*sizeof(double);
}
```

2.14 The TypeConstructor Instance

We define a Secondo type by creating an instance of *TypeConstructor* feeded with the functions defined before.

```
TypeConstructor SCircleTC(
   SCircle::BasicType(),        // name of the type
   SCircleProperty,             // property function
   OutSCircle, InSCircle,        // out and in function
   0, 0,                        // deprecated, don't think about it
   CreateSCircle, DeleteSCircle, // creation and deletion
   OpenSCircle, SaveSCircle,     // open and save functions
   CloseSCircle, CloneSCircle,   // close and clone functions
   CastSCircle,                 // cast function
   SizeOfSCircle,               // sizeOf function
   SCircleTypeCheck);           // type checking function
```

After creating the type constructor, the algebra can be defined and initialized. (please read the code in section 18). After the creation and initialization of the algebra, the algebra containing the type *scircle* can be integrated into secondo.

For compiling the algebra module, just type *make* within the algebra directory. For linking the algebra together with the kernel, navigate to Secondo's main directory and enter *make TTY*. If these calls were successful, you can start with first tests.

Start SecondoTTYBDB:

list algebras: the new algebra should be part of the result.

list algebra GuideAlgebra: the type constructor for *scircle* is displayed

Open a database (create one if no database exists)

query [const scircle value (9.0 10.0 20.0)]: a scircle object is displayed as a nested list

query [const scircle value (a + b)]: leads to an error message

let k1 = [const scircle value (9.0 10.0 20.0)] is successful

query k1: displays the created scircle object as a nested list

let k2 = k1: copies k1

delete k1: removed the object k1

If running secondo on a linux system with valgrind installed, Secondo can be started with:

```
SecondoTTYBDB --valgrind
```

This call starts Secondo within a valgrind environment and reports memory errors and memory leaks. If there is a memory leak, the reason of it is reported by:

```
SecondoTTYBDB --valgrindlc
```

3 Operator Implementation

Each operator implementation in Secondo contains of a type mapping, a set of value mappings, a selection function, a description, and a creation of an operator instance.

Furthermore, the syntax of the operator is described in the file *AlgebraName.spec* and at least one example must be given in the file *AlgebraName.examples*. If there is no example, the operator will be switched off by the Secondo framework.

The following sections present the implementation of a very simple operator without any specials. The operator takes a single *scircle* value as its argument and returns the perimeter (a real number) of this circle.

3.1 Type Mapping

The type mapping gets a nested list containing the argument types for this operator. Within the implementation, the number and the types of the arguments are checked to be a valid input for the operator. If the argument types cannot be handled by this operator, a type error is returned. Otherwise, the result of this function is the result type of the operator in nested list format.

```
ListExpr perimeterTM(ListExpr args){
   string err = "scircle expected";
   // check the number of arguments
   if(!nl->HasLength(args,1)){
     return listutils::typeError(err + " (wrong number of arguments)");
   }
   // check type of the argument
   if(!SCircle::checkType(nl->First(args))){
     return listutils::typeError(err);
   }
   // return the result type
   return listutils::basicSymbol<CcReal>();
}
```

3.2 Value Mapping

The value mapping takes values as arguments and computes the result. If the result of the operator is not a stream as here, the result storage of the operator tree node must be used for returning the result. The arguments are provided as an array of *Word* holding the arguments in the *addr* pointers. The type mapping ensures that only the expected types are behind these pointers and the cast will be successful. The parameters *message* and *local* are used for stream operators only. The parameter *s* is the operator's node within the operator tree. The result of the operator for non-stream operators is always 0.

The arguments are :

- *args*: array with the arguments of the operator
- *result*: output parameter, for non stream operators, the result storage must be used
- *message*: message used by stream operators
- *local*: possibility to store the state of an operator, used in stream operators
- *s*: node of this operator within the operator tree

```
int perimeterVM (Word* args, Word& result, int message,
                 Word& local, Supplier s) {
    SCircle* k = (SCircle*) args[0].addr;  // get the argument and cast it
    result = qp->ResultStorage(s);        // use the result storage
    CcReal* res = (CcReal*) result.addr; // cast the result
    res->Set(true, k->perimeter());       // compute and set the result
    return 0;
}
```

3.3 Specification

The specification provides an operator description for the user. The first argument of the *OperatorSpec* constructor is a description of the type mapping, the second argument describes the syntax of the operator, than comes the operator's meaning and the last argument used here is an example query. If required, an additional argument can provide some remark to this operator.

```
OperatorSpec perimeterSpec(
      "scircle -> real",
      "perimeter(_)",
      "Computes the perimeter of a disc.",
      "query perimeter([const scircle value (1.0 8.0 16.0)])"
 );
```

3.4 Operator Instance

Here, we create an instance of the operator using a constructor of the class *Operator* and feeding it with the defined functions. For non-overloaded operators, always the selection function *Operator::SimpleSelect* is used.

```
Operator perimeterOp(
   "perimeter",             // name of the operator
   perimeterSpec.getStr(),  // specification
   perimeterVM,             // value mapping
   Operator::SimpleSelect,  // selection function
   perimeterTM              // type mapping
);
```

After the creation of the operator instance, the operator must be added to the algebra within the algebra's constructor. Please take a look to section 18.

Furthermore, the syntax of the operator is described in the file *Guide.spec* (see section 20) and an example query including the result must be inserted into the file *Guide.examples* (see section 21).

After that, the operator can be tested. After the start of Secondo, the operator appears if *list algebra GuideAlgebra* is entered.

After opening a database, some queries can be entered, e.g.

```
query perimeter([const scircle value (1 2 3)])
```

The examples are processed by closing a running Secondo and entering:

```
Selftest tmp/Guide.examples
```

---

within Secondo's *bin* directory.

4 Overloaded Operators

In this section, we describe the implementation of an overloaded operator using the example of the *distN* operator. An overloaded operator can handle more the one type combination as its arguments and may have a different result type for each type combination.

The *distN* operator accepts two integers or two real numbers as arguments and returns the distance of these numbers as its result. If the input types are integers, the result is also an integer. In the case of a real number input, the result is a real number too.

4.1 Type Mapping

The type mapping of an overloaded operator handles all combinations of accepted input types.

```
ListExpr distNTM(ListExpr args){
    string err = "int x int or real x real expected";
    // check length
    if(!nl->HasLength(args,2)){
      return listutils::typeError(err + " (wrong number of arguments)");
    }
    // check for two integers
    if(   CcInt::checkType(nl->First(args))
       && CcInt::checkType(nl->Second(args))){
       return listutils::basicSymbol<CcInt>();
    }
    // check for two real numbers
    if(   CcReal::checkType(nl->First(args))
       && CcReal::checkType(nl->Second(args))){
       return listutils::basicSymbol<CcReal>();
    }
    // not accepted
    return listutils::typeError(err);
}
```

4.2 Value Mapping

For each type combination of an overloaded operator exists one value mapping. Here solved by a template parameter. If the handling of other type combinations differs, it is also possible to define more value mapping functions.

```
template<class T>
int distNVMT( Word* args, Word& result, int message,
              Word& local, Supplier s ){
  // get and casts the arguments
  T* a1 = (T*) args[0].addr;
  T* a2 = (T*) args[1].addr;
  // use the result storage of s
  result = qp->ResultStorage(s);
  T* res = (T*) result.addr;
  // in secondo, integers and reals can be undefined
  // if one of the arguments is not defined,
  // set the result to be undefined
  if(!a1->IsDefined() || !a2->IsDefined()){
     res->SetDefined(false);
```

```
    return 0;
  }
  // compute the distance and store it in the
  // result
  res->Set(true, a1->GetValue() - a2->GetValue());
  if(res->GetValue() < 0){
    res->Set(true, res->GetValue() * -1);
  }
  return 0;
}
```

4.3 Value Mapping Array and Selection Function

Each type combination has its own value mapping. Each value mapping is put into an array of value mappings. The Selection function picks the correct index within the value mapping array. Note that the selection function is only called if the type mapping function of the operator is passed. Thereby, here the check for correct list format can be omitted.

```
ValueMapping distNVM[] = {
    distNVMT<CcInt>, // value mapping handling two integers
    distNVMT<CcReal> // value mapping handling two reals
};

int distNSelect(ListExpr args){
  // int case at index 0
  if ( CcInt::checkType(nl->First(args)) ){
    return 0;
  }
  // real case at index 1
  if ( CcReal::checkType(nl->First(args)) ){
    return 1;
  }
  // should never be reached
  return -1;
}
```

4.4 Specification

In principle, there is no difference between the specification of a non-overloaded operator and an overloaded one. For overloaded operators each accepted type combination must be recognized from the description.

```
OperatorSpec distNSpec(
      " t x t -> t , with t in {int,real}",
      " _ distN _ ",
      "Computes the distance between two numbers",
      "query  1 distN 3"
);
```

4.5 Operator Instance

For an overloaded operator, another *Operator* constructor is used.

```
Operator distNOp(
     "distN",                 // operator's name
      distNSpec.getStr(),     // specification
      2,                      // number of Value Mappings
      distNVM,                // value mapping array
      distNSelect,            // selection function
      distNTM                 // type mapping
 );
```

After the creation of the operator instance, add the operator to the algebra, define the syntax in the *spec* file and give an example in the *examples* file. Do not forget to test the operator in Secondo.

5 Streams as Arguments of Operators

Streams are used to avoid the materialization of a large sets of objects. Streams can be an argument and/or the result of an operator.

If a stream is an argument of an operator, each element of the stream must be requested (similar to an iterator in a programming language). Before an element can be requested, the stream must be opened. After usage of the stream, the stream must be closed.

We describe the usage of streams by implementing the operator *countNumber*. This operator takes a stream of integers as its first argument and a single integer number as the second argument. The result of this operator is an integer value, reporting how often the second argument is contained in the stream.

For easy usage of streams, we use the *Stream* class defined in *Stream.h* as a wrapper. Please read this file (*include* directory of Secondo) for more information.

5.1 Type Mapping

A stream type representing a stream of type X has the nested list representation (stream X). The Stream class provides a *checkType* function doing this test.

```
ListExpr countNumberTM(ListExpr args){
  // check for correct number of arguments
  if(!nl->HasLength(args,2)){
    return listutils::typeError("wrong number of arguments");
  }
  // first argument must be a stream of integers
  // second argument must be a single integer
  if(  !Stream<CcInt>::checkType(nl->First(args))
    || !CcInt::checkType(nl->Second(args))){
    return listutils::typeError("stream(int)  expected");
  }
  // result is an integer
  return listutils::basicSymbol<CcInt>();
}
```

5.2 Value Mapping

Firstly, the first argument is put into the *Stream* constructor. The second argument is cast to a *CcInt*. The stream is opened. While the stream is not exhausted, we get the next element from the stream via the *request* function. We compare the stream element with the second argument and in the case of equality, we increase a counter. All stream elements are deleted. Because in Secondo attribute data types provide reference counting, we use the function *DeleteIfAllowed* (instead of direct delete) for this purpose. If the stream is exhausted (*request* returns null), we set the result *res* to the counter's value and close the stream. Because this value mapping produces no stream, the return value is 0.

```
int countNumberVM( Word* args, Word& result, int message,
                   Word& local, Supplier s ){
   result = qp->ResultStorage(s);  // use result storage for the result
   Stream<CcInt> stream(args[0]);  // wrap the stream
   CcInt* num = (CcInt*) args[1].addr;
   int count = 0;
   stream.open(); // open the stream
   CcInt* elem;
   while( (elem = stream.request()) ){  // request next element
      if(num->Compare(elem) == 0){
         count++;
      }
      elem->DeleteIfAllowed();
   }
   CcInt* res = (CcInt*) result.addr;
   res->Set(true,count);
   stream.close();
   return 0;
}
```

5.3 Specification

The specification is implemented as usual.

```
OperatorSpec countNumberSpec(
      " stream(int) x int -> int",
      " _ countNumber[_] ",
      " Computes how often a given number occurs within a stream",
      "query  intstream(1,10) countNumbers[2]"
 );
```

5.4 Operator Instance

Also the operator instance has no specials.

```
Operator countNumberOp(
   "countNumber",
   countNumberSpec.getStr(),
   countNumberVM,
   Operator::SimpleSelect,
   countNumberTM
);
```

After the last steps for the creation of an operator (adding to the algebra, defining syntax and providing an example), do not forget to test the operator. Example queries are:

```
query intstream(1,10) countNumber[4]

# result is 1

query intstream(1,10) countNumber[12]

# result is 0

query intstream(1,10) intstream(2,10) concat countNumber[8]

# result is 2
```

6 Streams as Results of Operators (stream operators)

If a stream is the result of an operator, we call such an operator stream-operator. The main difference to other operators is in the value mapping function.

We explain the implementation of a stream operator by the operator *getChars*. This operator gets a single string as its argument and returns a stream of strings where each string corresponds to a single character of the argument.

6.1 Type Mapping

The type mapping of a stream operator has no specials. The creation of the result is a little bit more complicated as for simple types because the typed stream must be returned.

```
ListExpr getCharsTM(ListExpr args){
  // check number of arguments
  if(!nl->HasLength(args,1)){
    return listutils::typeError("wrong number of arguments");
  }
  // argument must be of type string
  if(!CcString::checkType(nl->First(args))){
    return listutils::typeError("string expected");
  }
  // create the result type (stream string)
  return nl->TwoElemList( listutils::basicSymbol<Stream<CcString > >(),
                          listutils::basicSymbol<CcString>());
}
```

6.2 LocalInfo Class

The value mapping of a stream operator is called many times during the execution of a query. We need a structure, storing the current state of the operator. In the implementation of the

getChars operator, we have to store the current position within the input string. We encapsulate the state of the operator within a class and let do this class the whole work.

```
class getCharsLI{
  public:
   // constructor: initializes the class from the string argument
      getCharsLI(CcString* arg) : input(""), pos(0){
         if(arg->IsDefined()){
            input = arg->GetValue();
         }
      }
   // destructor
      ~getCharsLI(){}

   // this function returns the next result or null if the input is
   // exhausted
      CcString* getNext(){
         if(pos >= input.length()){
            return 0;
         }
         CcString* res = new CcString(true, input.substr(pos,1));
         pos++;
         return res;
      }
  private:
    string input;  // input string
    size_t pos;    // current position
};
```

6.3 Value Mapping

The value mapping of stream operators has a lot of differences compared to the value mapping of non-stream operator. One difference is that the *message* argument must be used to select the action to do. The messages are OPEN, REQUEST, and CLOSE. (if the operator supports progress estimation, some more messages must be handled). Furthermore, the *local* argument is used to store the current state of the operator (and doing the computations). The *addr* pointer of *local* is null at the first call of this operator. The operator is responsible to this pointer. After receiving a close message, the pointer must be set to null. Another difference to non-stream operators is that the result storage of *s* is not used. Instead, we write newly created objects into the *addr* pointer of *result*.

When an OPEN message is received, we firstly check whether a *localInfo* is already stored by checking the *addr* unequal to null. If so, we delete this structure and create a new one. We set the *addr* pointer of the *local* argument to this structure. The result of an OPEN message is always 0.

If a REQUEST message is received. We first look, whether we have already created a local info. If not, we set the *addr* pointer of *result* to null. If there is already such a structure, we compute the next result and store it into the *addr* pointer of *result*. The computation of the next result is delegated to the *getNext* function of the localInfo class. If there is a next result (addr pointer of result is not null), the operator returns YIELD, otherwise CANCEL.

In the case of a CLOSE message, we free the memory allocated by the local info class and set the *addr* pointer of *local* to null. The result to a CLOSE message is always 0.

```
int getCharsVM( Word* args, Word& result, int message,
                Word& local, Supplier s ){
    getCharsLI* li = (getCharsLI*) local.addr;
    switch(message){
      case OPEN : if(li) {
                    delete li;
                  }
                  local.addr = new getCharsLI( (CcString*) args[0].addr);
                  return 0;
      case REQUEST: result.addr = li?li->getNext():0;
                    return result.addr?YIELD:CANCEL;
      case CLOSE:  if(li){
                     delete li;
                     local.addr = 0;
                   }
                   return 0;
    }
    return 0;
}
```

6.4 Specification

The specification of a stream operator has no specials.

```
 OperatorSpec getCharsSpec(
       " string -> stream(string)",
       " getChars(_) ",
       " Seperates the characters of a string. ",
       " query  getChars(\"secondo\") count"
  );
```

6.5 Operator instance

The creation of the operator instance is the same as for non-stream operators.

```
Operator getCharsOp(
   "getChars",
   getCharsSpec.getStr(),
   getCharsVM,
   Operator::SimpleSelect,
   getCharsTM
);
```

As usual, the final steps are:

- add the operator to the algebra
- define the syntax in the *spec* file
- give an example in the *examples* file
- test the operator in Secondo

7 Streams as Both, Arguments and Results

Some operators have a stream as an argument and return also a stream. The implementation combines stream consuming with stream producing operators.

We show as an example the operator *startsWithS*. This is a kind of filter operator. It receives a stream of strings and a single string argument. All elements in the stream starting with the second argument pass the operator, all others are filtered out.

7.1 Type Mapping

The type mapping is quite usual.

```
ListExpr startsWithSTM(ListExpr args){
 if(!nl->HasLength(args,2)){
   return listutils::typeError("wrong number of args");
 }
 if(    !Stream<CcString>::checkType(nl->First(args))
     || !CcString::checkType(nl->Second(args))){
    return listutils::typeError("stream(string) x string expected");
 }
 return nl->First(args);
}
```

7.2 LocalInfo Class

As for other stream operators, we create a local info class storing the state of this operator and computing the next result element.

Because we create an instance of this class in case of a OPEN message and delete the instance in case of a CLOSE message, we open the argument stream in the constructor and close it in the destructor.

Elements passing the test are just returned as the next result. Filtered out strings are deleted.

```
class startsWithSLI{
   public:

     // s is the stream argument, st the string argument
     startsWithSLI(Word s, CcString* st): stream(s), start(""){
        def = st->IsDefined();
        if(def){ start = st->GetValue(); }
        stream.open();
     }

     ~startsWithSLI(){
        stream.close();
      }

     CcString* getNext(){
        if(!def){ return 0; }
        CcString* k;
        while( (k = stream.request())){
           if(k->IsDefined() && stringutils::startsWith(k->GetValue(), start)){
             return k;
```

```
            }
            k->DeleteIfAllowed();
          }
          return 0;
        }

    private:
      Stream<CcString> stream;
      string start;
      bool def;
  };
```

7.3 Value Mapping

Because the complete functionality is outsourced to the *LocalInfo* class, the implementation of the actual value mapping is straightforward.

```
int startsWithSVM( Word* args, Word& result, int message,
                   Word& local, Supplier s ){
    startsWithSLI* li = (startsWithSLI*) local.addr;
    switch(message){
      case OPEN : if(li) {
                    delete li;
                  }
                  local.addr = new startsWithSLI(args[0],
                                                 (CcString*) args[1].addr);
                  return 0;
      case REQUEST: result.addr = li?li->getNext():0;
                    return result.addr?YIELD:CANCEL;
      case CLOSE:  if(li){
                     delete li;
                     local.addr = 0;
                   }
                   return 0;
    }
    return 0;
}
```

7.4 Specification

```
 OperatorSpec startsWithSSpec(
       " stream(string) x string -> stream(string)",
       " _ startsWithS[_]",
       " All strings in the stream not starting with the second "
       " are filtered out from the stream",
       " query  plz feed projecttransformstream[Ort] startsWithS(\"Ha\") count"
  );
```

7.5 Operator Instance

```
Operator startsWithSOp(
   "startsWithS",
   startsWithSSpec.getStr(),
   startsWithSVM,
```

```
   Operator::SimpleSelect,
   startsWithSTM
);
```

The final steps are the same as for other operators.

8 Functions as Arguments

Functions are part of the Secondo framework. Sometimes, an operator needs to evaluate a function during its execution. For example, the filter operator of Secondo has a function as its second argument representing the filter condition.

The usage of functions as arguments to an operator is explained at the example of the *replaceElem* operator. This operator has a stream of some attribute data type X as its first argument while the second argument is a function having the type X as its only argument and returns another (or may be the same) attribute data type.

8.1 Type Mapping

The type mapping itself has no speciality. The type of a function is given by *(map arguments result)*, e.g. *(map real int bool)* for a function computing a *bool* value from a *real* and an *int*.

```
ListExpr replaceElemTM(ListExpr args){
   if(!nl->HasLength(args,2)){
      return listutils::typeError("wrong number of arguments");
   }
   if(!Stream<Attribute>::checkType(nl->First(args))){
      return listutils::typeError("first argument has to be a "
                                  "stream of attributes");
   }
   if(!listutils::isMap<1>(nl->Second(args))){
      return listutils::typeError("second arg has to be a map "
                                  "with 1 argument");
   }
   ListExpr StreamElem = nl->Second(nl->First(args));
   ListExpr MapArg = nl->Second(nl->Second(args));
   if(!nl->Equal(StreamElem, MapArg)){
      return listutils::typeError("map arg not equal to stream elem");
   }
   ListExpr res = nl->Third(nl->Second(args));
   // result of the function must be an attribute again, e.g. in
   // kind DATA
   if(!listutils::isDATA(res)){
     return listutils::typeError("map result is not in kind DATA");
   }
   return nl->TwoElemList( listutils::basicSymbol<Stream<Attribute> >(),
                           res);
}
```

8.2 LocalInfo

Before a function can be evaluated, the arguments of this function must be set. All arguments of a function can be stored into the argument vector of the function. The type of this array

in called ArgVectorPointer. The argument vector of a specific function can be get using the *Argument* function of the query processor. After putting the arguments into the argument vector, the function can be evaluated using the *Request* function of the query processor. Note that the result of a function returning a single value (i.e. not a stream) should not be deleted and hence also not put into the output stream. For this reason, the Clone function is used in the *getNext* function below.

```
class replaceElemLI{
   public:
     replaceElemLI(Word st, Word f): stream(st), fun(f){
        // open the input stream
        stream.open();
        // get the argument vector
        funargs = qp->Argument(f.addr);
     }

     ~replaceElemLI(){
        // close the input stream
        stream.close();
      }

     Attribute* getNext(){
        // get the next element from the input stream
        Attribute* funarg = stream.request();
        if(!funarg){ return 0; }
        // put this argument to the argument vector
        (*funargs[0]) = funarg;
        //
        Word funres;
        qp->Request(fun.addr, funres);
        // free the memory of the input element
        funarg->DeleteIfAllowed();
        // cast the function result
        Attribute* res = (Attribute*) funres.addr;
        // return a clone of the function result to
        // avoid a deletion of the result
        return res->Clone();
     }
   private:
      Stream<Attribute> stream;
      Word fun;
      ArgVectorPointer funargs;
};
```

8.3 Value Mapping

As usual, the local info class does the work and the actual value mapping is quite simple.

```
int replaceElemVM( Word* args, Word& result, int message,
                   Word& local, Supplier s ){
   replaceElemLI* li = (replaceElemLI*) local.addr;
   switch(message){
      case OPEN:
           if(li) { delete li; }
           local.addr = new replaceElemLI(args[0], args[1]);
            return 0;
```

```
      case REQUEST:
             result.addr = li?li->getNext():0;
             return result.addr?YIELD:CANCEL;
      case CLOSE:
              if(li){
                 delete li;
                 local.addr = 0;
              }
              return 0;
   }
   return 0;
}
```

8.4 Specification

```
OperatorSpec replaceElemSpec(
   "stream(X) x (fun : X -> Y) -> stream(Y)",
   " _ replaceElem[_] ",
   " replaces the element in the stream by the function results",
   " query intstream(1,30) replaceElem[fun(i : int) i * 1.5) count"
);
```

8.5 Operator Instance

```
Operator replaceElemOp(
   "replaceElem",
   replaceElemSpec.getStr(),
   replaceElemVM,
   Operator::SimpleSelect,
   replaceElemTM
);
```

Do not forget the final steps for the operator.

9 Implementing Attribute Data Types

Relations (tables) of a database systems store sets of tuples which in turn consist of some attributes. Types which can be part of a tuple are called attribute data types. Because relations in Secondo use a generic approach for storing tuples to disc, the internal structure of attribute data types is restricted. Furthermore, attribute data types must provide implementations of a set of functions to support some operators, e.g. a *compare* function to support *sort* operators. To mark a type as an attribute data type, it must be in the kind *DATA*. Attributes can be defined or not. In operators using attribute data types, always a check of the defined state is required.

9.1 Restricted Structure

Firstly, an attribute data type must be derived from the class *Attribute*. Because of the frequent use of void pointers in secondo, multiple inheritance is not allowed and may lead to a crash of the system under certain circumstances. But it is allowed to build chains of inheritances having the class *Attribute* as the root.

The use of pointers is forbidden within a class representing an attribute. This restriction also forbids member variables using pointers, for example the most of the STL classes. For implementing attributes of variable size, FLOBs are used (see section 10).

The standard constructor (the one without any argument) cannot do anything.

All other constructors must call the constructor of the Attribute class taking a *bool* argument.

9.2 Golden Rules for Implementing Attribute Types

- Derive from class *Attribute*
- Never use pointers within the class (also do not use members having pointers)
- Always define the standard constructor (the one without any argument) having an empty implementation
- Always implement at least another constructor
- In all non-standard constructors, call the constructor of the class *Attribute* having a boolean argument. Initialize **all** members of your class. If the class has FLOB or DbArray members, use the constructor receiving the initial size.
- Never use the standard constructor except in the cast function

If you don't take these rules to heart, Secondo will run instable.

9.3 Additional Functions

Besides the normal functions of the class, the following functions are required to form the internal structure of an attribute data type.

- virtual int NumOfFLOBs() const
  When implementing an attribute type having variable size, the class contains a fixed number of FLOBs. This function returns this number.
- virtual Flob* GetFLOB(const int i)
  This funtion returns a pointer to one of the flobs.
- int Compare(const Attribute* arg) const
  This function compares the current instance of the class with another attribute. It is ensured that the *arg* argument has the same C++ type as the instance itself. This function is used for example for the *sort* operator. This function returns a negative value if the called object is smaller than the argument, 0 if both objects are equal and a positive value otherwise. Note that two non-defined objects are equal and an undefined object is smaller than a defined one.
- bool Adjacent(const Attribute* arg) const
  This operator checks whether this instance and arg (having the same type as the instance) are adjacent. This operator is used to build generic range types. Because these generic range types was never implemented, this function can just return false if there is no meaningful adjacent relation between instances of this class.

- size_t Sizeof() const
  This function returns the size of the memory block allocated by this class. The implementation of this function is always:
  `return sizeof(*this);`

- size_t HashValue() const
  This function returns a hash value for this object and supports among others the *hashjoin* operator. If there is no meaningful implementation for this function, the *hashjoin* operator will be extremely slow if there is a large input.

- void CopyFrom(const Attribute* arg)
  This operations copyies the value of *arg* into the called instance. The C++ type of *arg* is the same as the one of the called instance. This operator is used by operations like *extract*, *max*, *min* and many other.

- Attribute* Clone() const
  Returns a depth copy of the attribute. This operation is used for example in the *extend* operator.

Because Secondo is developed since many years, there are several possibilities to define an attribute data type. In this document two of them are presented.

This first method is the classical one, defining a lot of independent functions as input for a Secondo type constructor. This method should be used if an existing Secondo data type is to tranfer into an attribute data type or in some special cases. If the type is planned to be an attribute data type from the beginning, the secondo method is to prefer. The second method uses static member functions of the class for defining an attribute data type using a generic template class.

9.4 The Classical Method

9.4.1 Defining the Class

We call the class *ACircle* standing for (attribute circle). Because a circle can be represented using three double values, this class has fixed size and hence no pointers are used.

```
class ACircle: public Attribute{
    public:
      ACircle() {}  // cannot do anything

      ACircle(const double _x, const double _y, const double _r):
         Attribute(true),  //  do not forget the initialization of
                           // the super class
         x(_x), y(_y), r(_r) {}

      // copy constructor
      ACircle(const ACircle & c) : Attribute(c.IsDefined()),
       x(c.x), y(c.y),r(c.r) {}

      // assignment operator
      ACircle& operator=(const ACircle& src){
         SetDefined(src.IsDefined());
```

```
      x = src.x;
      y = src.y;
      r = src.r;
      return *this;
   }

    // destructor
   ~ACircle(){}

   // auxiliary functions
   static const string BasicType(){ return "acircle"; }
   static const bool checkType(const ListExpr list) {
      return listutils::isSymbol(list, BasicType());
   }

   // perimeter construction
   double perimeter() const{
      return 2*M_PI*r;
   }
  // here, the functions required for an attribute
  // data type are defined


  // NumOfFLOBs
  // for class without FLOB members, this function
  // can be omitted or it returns 0
  inline virtual int NumOfFLOBs() const {
      return 0;
  }
  // this class contains no FLOB to return
  inline virtual Flob* GetFLOB( const int i ) {
    assert(false);
    return 0;
  }

   // compare: always implement this function
   // if there is no natural order, just use any
   // valid order to the objects, in this example,
   // a lexicographical order is chosen
   int Compare(const Attribute* arg) const{
      // first compare defined and undefined flag
      if(!IsDefined()){
        return arg->IsDefined()?-1:0;
      }
      if(!arg->IsDefined()){
         return 1;
      }
      ACircle* c = (ACircle*) arg;
      if(x < c->x) return -1;
      if(x > c->x) return 1;
      if(y < c->y) return -1;
      if(y > c->y) return 1;
      if(r < c->r) return -1;
      if(r > c->r) return 1;
      return 0;
   }
```

```
    // we dont want to create a range type over acircles,
    // thus, just false is returned
    bool Adjacent(const Attribute* arg) const{
       return false;
    }
    // standard implementation of Sizeof
    size_t Sizeof() const{
       return sizeof(*this);
    }

    // defines a meaningful hash function e.g., for
    // support of hash joins
    size_t HashValue() const{
       if(!IsDefined()){
          return 0;
       }
       return (size_t) (x + y + r);
    }

    // takes the values from arg over
    // delegated to the assignment operator
    void CopyFrom(const Attribute* arg){
        *this = *((ACircle*)arg);
    }

    // returns a depth copy from this object
    Attribute* Clone() const{
       return new ACircle(*this);
    }


      // usual functions
      double getX() const{ return x; }
      double getY() const{ return y; }
      double getR() const{ return r; }
    private:
      double x;
      double y;
      double r;
};
```

9.4.2 Secondo Interface Support

9.4.2.1 Property function This function is very similar to the SCircle property function. Note that the name and the signature have been changed.

```
ListExpr ACircleProperty () {
return ( nl -> TwoElemList (
   nl->FourElemList (
     nl->StringAtom ( " Signature " ) ,
     nl->StringAtom ( " Example Type List " ) ,
     nl->StringAtom ( " List Rep " ) ,
     nl->StringAtom ( " Example List " )) ,
   nl->FourElemList (
     nl->StringAtom ( " -> DATA " ) ,
     nl->StringAtom ( ACircle::BasicType ()) ,
```

```
    nl->StringAtom ( " ( real real real ) = (x ,y ,r ) " ) ,
    nl->StringAtom ( " (13.5 -76.0 1.0) " )
)));
}
```

9.4.2.2 IN function Because each attribute may be undefined, a special treatment is necessary for this case.

```
Word InACircle( const ListExpr typeInfo, const ListExpr instance,
    const  int errorPos, ListExpr& errorInfo, bool& correct ){
   // create a result with addr pointing to 0
   Word res((void*)0);
   // assume an incorrect list
   correct = false;

   // check for undefined
   if(listutils::isSymbolUndefined(instance)){
      correct = true;
      ACircle* c = new ACircle(1,2,3);
      c->SetDefined(false);
      res.addr = c;
      return res;
   }

   // check whether the list has three elements
   if(!nl->HasLength(instance,3)){
     cmsg.inFunError("expected three numbers");
     return res;
   }
   // check whether all elements are numeric
   if(   !listutils::isNumeric(nl->First(instance))
      || !listutils::isNumeric(nl->Second(instance))
      || !listutils::isNumeric(nl->Third(instance))){
      cmsg.inFunError("expected three numbers");
      return res;
   }
   // get the numeric values of the elements
   double x = listutils::getNumValue(nl->First(instance));
   double y = listutils::getNumValue(nl->Second(instance));
   double r = listutils::getNumValue(nl->Third(instance));
   // check for a valid radius
   if(r<=0){
      cmsg.inFunError("invalid radius");
      return res;
   }
   // list was correct,  create the result
   correct = true;
   res.addr = new ACircle(x,y,r); // is defined, see constructor
   return res;
 }
```

9.4.2.3 OUT function Also in this function, undefined values must be handled in a special case.

```
ListExpr OutACircle( ListExpr typeInfo, Word value ) {
```

```
   ACircle* k = (ACircle*) value.addr;
   if(!k->IsDefined()){
    return listutils::getUndefined();
   }
   return nl->ThreeElemList(
              nl->RealAtom(k->getX()),
              nl->RealAtom(k->getY()),
              nl->RealAtom(k->getR()));
}
```

9.4.2.4 Create function, Delete function, Close function

Again very similar to the *SCircle* versions.

```
Word CreateACircle( const ListExpr typeInfo ) {
  Word w;
  w.addr = (new ACircle(0,0,1.0));
  return w;
}


void DeleteACircle ( const ListExpr typeInfo , Word & w ) {
   ACircle * k = ( ACircle *) w.addr ;
   delete k ;
   w.addr = 0;
}

void CloseACircle ( const ListExpr typeInfo , Word & w ) {
  ACircle * k = ( ACircle *) w.addr ;
  delete k ;
  w.addr = 0;
}
```

9.4.2.5 Open and Save Functions

For attribute data types, generic *Open* and *Save* functions are available. Thus we don't need an implementation here.

9.4.2.6 Clone function

Because a class representing an attribute data type has an own *Clone* function, the implementation of Secondo's Clone function is simpler than for non-attribute types.

```
Word CloneACircle ( const ListExpr typeInfo , const Word & w ){
    ACircle * k = ( ACircle *) w . addr ;
    Word res(k->Clone());
    return res;
}
```

9.4.2.7 Cast function

The cast function is implemented as before.

```
void * CastACircle ( void * addr ) {
   return ( new ( addr ) ACircle );
}
```

9.4.2.8 Type check

Within the *TypeCheck* function, we can call the appropriate class function.

```
bool ACircleTypeCheck ( ListExpr type , ListExpr & errorInfo ){
   return ACircle::checkType(type);
}
```

9.4.2.9 SizeOf function

Because of the generic Open and Save functions, we return the size of the class.

```
int SizeOfACircle () {
    return sizeof ( ACircle );
}
```

9.4.2.10 Type Constructor instance

This is the same constructor as for non-attribute data types. Note the usage of the generic open and save functions.

```
TypeConstructor ACircleTC(
   ACircle::BasicType(),         // name of the type
   ACircleProperty,              // property function
   OutACircle, InACircle,         // out and in function
   0, 0,                        // deprecated, don't think about it
   CreateACircle, DeleteACircle, // creation and deletion
   OpenAttribute<ACircle>,        // open function
   SaveAttribute<ACircle>,         //  save functions
   CloseACircle, CloneACircle,    // close and clone functions
   CastACircle,                  // cast function
   SizeOfACircle,                // sizeOf function
   ACircleTypeCheck);            // type checking function
```

After adding this type constructor to the algebra, the type constructor is inserted into kind *DATA* to mark it as an attribute data type. See section 18.

9.5 Attribute Data Type – the Modern Way

When defining an attribute data type in the way described above, a lot of functions having a standard implementation must be implemented. For attribute data types there is another way for rapid implementing them without the stupid repeatation of code. Here, all functions which cannot be handled automatically are part of the class (the most of them are static member functions). We show the class *GCircle* (generic circle) as an example. Note that for using this method the header *GenericTC.h* must be included.

9.5.1 The class implementation

Besides the usual functions required to implement an attribute data type, the following functions must be implemented if the generic approach should be used. In return, no non-class functions must be implemented.

- a type constructor taking an *int* value as an argument
- a static *Property* function

- a *ToListExpr* function replacing the *OUT* function
- a function called *CheckKind* checking the type (and having a wrong name ;-))
- a static *ReadFrom* function replacing the *IN* function

```
class GCircle: public Attribute{
    public:
      GCircle() {}  // cannot do anything

      GCircle(const double _x, const double _y, const double _r):
         Attribute(true),  //  do not forget the initialization of
                           // the super class
         x(_x), y(_y), r(_r) {}

      // copy constructor
      GCircle(const GCircle & c) : Attribute(c.IsDefined()),
       x(c.x), y(c.y),r(c.r) {}

      // assignment operator
      GCircle& operator=(const GCircle& src){
         SetDefined(src.IsDefined());
         x = src.x;
         y = src.y;
         r = src.r;
         return *this;
      }

       // destructor
      ~GCircle(){}

      // auxiliary functions
      static const string BasicType(){ return "gcircle"; }
      static const bool checkType(const ListExpr list) {
         return listutils::isSymbol(list, BasicType());
      }

      // perimeter computation
      double perimeter() const{
         return 2*M_PI*r;
      }
     // here, the functions required for an attribute
     // data type are defined


     // NumOfFLOBs
     // for class without FLOB members, this function
     // can be omitted
     inline virtual int NumOfFLOBs() const {
         return 0;
     }
     // this class contains no FLOB to return
     inline virtual Flob* GetFLOB( const int i ) {
       assert(false);
       return 0;
     }
```

```
    // compare, always implement this function
    // if there is no natural order, just use any
    // valid order to the objects, in this example,
    // a lexicographical order is chosen
    int Compare(const Attribute* arg) const{
       if(!IsDefined()){
         return arg->IsDefined()?-1:0;
       }
       if(!arg->IsDefined()){
          return 1;
       }
       GCircle* c = (GCircle*) arg;
       if(x < c->x) return -1;
       if(x > c->x) return 1;
       if(y < c->y) return -1;
       if(y > c->y) return 1;
       if(r < c->r) return -1;
       if(r > c->r) return 1;
       return 0;
    }

  // we dont want to create a range type over GCircles,
  // thus, just false is returned
  bool Adjacent(const Attribute* arg) const{
     return false;
  }
  // standard implementation of Sizeof
  size_t Sizeof() const{
     return sizeof(*this);
  }

  // define a meaningful hash function for
  // support of hash joins
  size_t HashValue() const{
     if(!IsDefined()){
        return 0;
     }
     return (size_t) (x + y + r);
  }

  void CopyFrom(const Attribute* arg){
      *this = *((GCircle*)arg);
  }

  Attribute* Clone() const{
     return new GCircle(*this);
  }
```

Here, the additional functions start

```
// Additional type constructor taking some int as argument
  GCircle(int dummy): Attribute(false), x(0), y(0), r(1) {}

// Property functions
     static ListExpr Property(){
        return gentc::GenProperty("-> DATA", // signature
                        BasicType(),         // type description
                        "(real real real)",  // list rep
```

```
                        "1.0 2.0 3.0)");      // example list
    }

    //  Type check function
    static bool CheckKind(ListExpr type, ListExpr& errorInfo){
        return checkType(type);
    }

    // replacement for the IN function
    bool ReadFrom(ListExpr LE, const ListExpr typeInfo){
       // handle undefined value
       if(listutils::isSymbolUndefined(LE)){
          SetDefined(false);
          return true;
        }

        if(!nl->HasLength(LE,3)){
          cmsg.inFunError("three numbers expected");
          return false;
        }
        if(    !listutils::isNumeric(nl->First(LE))
            || !listutils::isNumeric(nl->Second(LE))
            || !listutils::isNumeric(nl->Third(LE))){
          cmsg.inFunError("three numbers expected");
          return false;
         }
         double x = listutils::getNumValue(nl->First(LE));
         double y = listutils::getNumValue(nl->Second(LE));
         double r = listutils::getNumValue(nl->Third(LE));
         if( r<=0){
            cmsg.inFunError("invalid radius");
            return false;
         }
         SetDefined(true);
         this->x = x;
         this->y = y;
         this->r = r;
         return true;
    }


    // replacement for the out function
    ListExpr ToListExpr(ListExpr typeInfo) const{
       if(!IsDefined()){
         return listutils::getUndefined();
       }
       return nl->ThreeElemList(
           nl->RealAtom(x),
           nl->RealAtom(y),
           nl->RealAtom(r)
        );
    }

// normal stuff

     double getX() const{ return x; }
     double getY() const{ return y; }
```

```
      double getR() const{ return r; }
    private:
      double x;
      double y;
      double r;
};
```

9.5.2 Creating a type constructor instance

Because all required functionality is encapsulated within the class, no non-class functions have to be implemented. Just instantiate the *GenTC* template class using the class name to define the type constructor instance named *GCircleTC*.

```
GenTC<GCircle> GCircleTC;
```

10 Attribute Types having Variable Size

Up to now, the defined attribute types have had a fixed size. The problem is how to implement attribute types having variable size although pointers are forbidden. The solution provided in the Secondo system are so called FLOBs (Faked Large Objects) which can be embedded into an attribute data type. Basically a FLOB is a unstructured memory block. For using FLOBs directly see the *BinaryFileAlgebra* implementation. Mostly, a set of structured data should be part of an attribute. To realize this, Secondo provides a *DbArray* implementation derived from the FLOB class. A *DbArray* can store an arbitrary number of a structure and offers random access to its elements. Pointers are not allowed to be part of *DbArray* elements. If required, use logical pointers (indexes in *DbArrays*) to realize pointers. *FLOBs* and *DbArrays* cannot be nested.

We show the implementation of an integer list as an example. In all constructors except the standard constructor, all *DBArray* members have to be initialized. Otherwise Secondo will crash when using it.

Do not forget the include:

```
#include "../../Tools/Flob/DbArray.h"
```

```
class IntList: public Attribute{
    public:
      IntList() {}  // cannot do anything

      IntList(int dummy): // must initialize attribute and the DbArray using
                          // non-standard constructors
           Attribute(true), content(0) {}
```

```
    // copy constructor
    IntList(const IntList & c) : Attribute(c.IsDefined()),
       content(c.content.Size()){
       content.copyFrom(c.content);
    }


    // assignment operator
    IntList& operator=(const IntList& src){
       SetDefined(src.IsDefined());
       content.copyFrom(src.content);
       return *this;
    }

     // desctructor
    ~IntList(){}

    // auxiliary functions
    static const string BasicType(){ return "intlist"; }
    static const bool checkType(const ListExpr list) {
       return listutils::isSymbol(list, BasicType());
    }

    void append(CcInt* i) {
       if(!i->IsDefined()){
         SetDefined(false);
       } else if(IsDefined()){
         content.Append(i->GetValue());
       }
    }

    void append(int i) {
      content.Append(i);
    }


   // NumOfFLOBs
   // this class has one FLOB in form of a DbArray
   inline virtual int NumOfFLOBs() const {
       return 1;
   }
   // return the flob if index is correct
   inline virtual Flob* GetFLOB( const int i ) {
     assert(i==0);
     return &content;
   }

    // compare, always implement this function
    // if there is no natural order, just use any
    // valid order to the objects, in this example,
    // a lexicographical order is chosen
    int Compare(const Attribute* arg) const{
       if(!IsDefined()){
         return arg->IsDefined()?-1:0;
       }
       if(!arg->IsDefined()){
          return 1;
```

```
        }
        IntList* i = (IntList*) arg;
        // first criterion number of entries
        if(content.Size() < i->content.Size()){
           return -1;
        }
        if(content.Size() < i->content.Size()){
           return 1;
        }
        for(int k=0;k<content.Size();k++){
           int i1, i2;
           content.Get(k,i1);
           i->content.Get(k,i2);
           if(i1<i2) return -1;
           if(i1>i2) return 1;
        }
        return 0;
     }

   // there is no meaningful Adjacent implementation
   bool Adjacent(const Attribute* arg) const{
      return false;
   }
   // standard implementation of Sizeof
   size_t Sizeof() const{
      return sizeof(*this);
   }

   // define a meaningful hash function for
   // support of hash joins and others
   size_t HashValue() const{
      if(!IsDefined()){
         return 0;
      }
      // sum up the first 5 elements
      int sum = 0;
      int max = min(5,content.Size());
      for(int i=0;i<max; i++){
         int v;
         content.Get(i,v);
         sum += v;
      }
      return (size_t) sum;
   }

   void CopyFrom(const Attribute* arg){
       *this = *((IntList*)arg);
   }

   Attribute* Clone() const{
      return new IntList(*this);
   }


// Additionall  functions
     static ListExpr Property(){
        return gentc::GenProperty("-> DATA", // signature
```

```
                        BasicType(),          // type description
                        "(int int ...)",   // list rep
                        "(1 2 3)");      // example list
    }

    //  Type check function
    static bool CheckKind(ListExpr type, ListExpr& errorInfo){
        return checkType(type);
    }

    // replacement for the IN function
    bool ReadFrom(ListExpr LE, const ListExpr typeInfo){
       // handle undefined value
       if(listutils::isSymbolUndefined(LE)){
           SetDefined(false);
           return true;
        }
        if(nl->AtomType(LE)!=NoAtom){
           return false;
        }
        SetDefined(true);
        content.clean();
        while(!nl->IsEmpty(LE)){
           ListExpr f = nl->First(LE);
           LE = nl->Rest(LE);
           if(nl->AtomType(f)!=IntType){
              return false;
           }
           append(nl->IntValue(f));
        }
        return true;
    }


    // replacement for the out function
    ListExpr ToListExpr(ListExpr typeInfo) const{
       if(!IsDefined()){
         return listutils::getUndefined();
       }
       if(content.Size()==0){
         return nl->TheEmptyList();
       }
       int v;
       content.Get(0,v);
       ListExpr res = nl->OneElemList(nl->IntAtom(v));
       ListExpr last = res;
       for(int i=1;i<content.Size(); i++){
          content.Get(i,v);
          last = nl->Append(last,nl->IntAtom(v));
       }
       return res;
    }

    private:
     DbArray<int> content;
};
```

```
GenTC<IntList> IntListTC;
```

11 Advanced Type Mappings

For some operators it is desirable to transfer information computed within the type mapping to the value mapping. The standard case is the index of an attribute within a tuple for a certain attribute name. Another application are default arguments. For this purpose, Secondo provides the so-called APPEND mechanism. In general is works as follows. Instead of returning just the result type within the type mapping, a list of length three of the form (*APPEND args result*) where APPEND is a keyword (symbol), *args* is a list containing additional arguments and *result* is the normal result type. The content of args is accessible within the value mapping as when the user had given additional arguments directly.

This mechanism is explained at the *attrIndex* operator. This operator gets a stream of tuples and an attribute name. The result of this operator is the index of the attribute with given name in the tuple. The content of the tuple stream remains untouched.

11.1 Type Mapping

Here, the APPEND mechanism explained above is used.

```
ListExpr attrIndexTM(ListExpr args){

   if(!nl->HasLength(args,2)){
      return listutils::typeError("wrong number of arguments");
   }
   if(!Stream<Tuple>::checkType(nl->First(args))){
     return listutils::typeError("first arg is not a tuple stream");
   }
   if(nl->AtomType(nl->Second(args))!=SymbolType){
      return listutils::typeError("second arg is not a valid attribute name");
   }
   // extract the attribute list
   ListExpr attrList = nl->Second(nl->Second(nl->First(args)));
   ListExpr type;
   string name = nl->SymbolValue(nl->Second(args));
   int j = listutils::findAttribute(attrList, name, type);
   // the append mechanism
   return nl->ThreeElemList(
            nl->SymbolAtom(Symbols::APPEND()),
            nl->OneElemList(nl->IntAtom(j)),
            listutils::basicSymbol<CcInt>());
}
```

11.2 Value Mapping

Within the value mapping the appended arguments are accessible in the normal way.

```
int attrIndexVM ( Word * args , Word & result , int message ,
                  Word & local , Supplier s ) {
  result = qp->ResultStorage(s);
```

```
  CcInt* append = (CcInt*) args[2].addr; // the appended value
  CcInt* res = (CcInt*) result.addr;
  int v = append->GetValue();
  if(v==0){
    res->SetDefined(false);
  } else {
    res->Set(true,v-1);
  }
  return 0;
}
```

11.3 Specification

```
OperatorSpec attrIndexSpec (
    " stream(tuple(X) ) x symbol  -> int " ,
    " _ attrIndex[ _ ]  " ,
    " Returns the index of the attribute with given name. " ,
    " query plz feed attrIndex[Ort]  "
);
```

11.4 Operator Instance

```
Operator attrIndexOp (
  "attrIndex" , // name of the operator
  attrIndexSpec.getStr() , // specification
  attrIndexVM , // value mapping
  Operator::SimpleSelect , // selection function
  attrIndexTM // type mapping
);
```

12 Implementing Large Structures

Up to now, all implemented data types within this algebra were more or less small. This section deals with the implementation of real big data types. This means, this type can massive exceed the main memory of the underlying system if it would be completely loaded.

To solve this problem, only small parts of the whole structure are in memory, the main part is located on disc. Using files directly would violate the ACID properties of a database system because there is no synchronization if several users work in parallel. For this reason, file structures provided by the Secondo-SMI have to be used for such types. The SecondoSMI provides several file types, *RecordFiles*, *HashFiles*, and *Queues*. The used example will use the *RecordFile* class. A *RecordFile* stores records within a file where each record is assigned to a unique id (the *RecordId*). Depending on the used constructor, records within a file may have fixed or variable size.

We explain the usage of RecordFiles at the example of an AVL-tree, a balanced version of a binary searchtree. Because pointers are not possible within persistent structures, pointers are simulated by *RecordID*s. For simplicity, we allow to store only integer values within the tree.

12.1 Node Class

A node of an AVL-tree consists of its content, pointers to the two sons and a value denoting the height which is used for balancing these nodes. Each node corresponds to a record within the

file representing the tree. As mentioned above, we simulate pointers by *RecordId*s. For technical reasons, each node contains also its own *RecordId*. This makes it easier to update the node after changes. The *Node* class provides methods for reading its value from a record and writing to a record.

```
class Node{
 public:
  typedef uint16_t htype;
```

12.1.1 Constructor

This constructor is used to create a new node not assigned to a record. The node will have no childs (represented by the record id 0.

```
  Node(int v): value(v), left(0), right(0),height(1),myId(0){}
```

12.1.2 Constructor

This constructor creates the main memory representation of a node from a Record.

```
  Node(SmiRecord& r){
     myId = r.GetId();
     size_t offset = 0;
     r.Read(&value,sizeof(CcInt::inttype),offset);
     offset += sizeof(CcInt::inttype);
     r.Read(&left,sizeof(SmiRecordId),offset);
     offset += sizeof(SmiRecordId);
     r.Read(&right,sizeof(SmiRecordId),offset);
     offset += sizeof(SmiRecordId);
     r.Read(&height,sizeof(htype),offset);
  }

   // copy constructor
   Node(const Node& n):
      value(n.value), left(n.left), right(n.right),
       height(n.height),myId(n.myId) {}

   // assignment operator
   Node& operator=(const Node& n){
      value = n.value;
      left = n.left;
      right = n.right;
      height = n.height;
      myId = n.myId;
      return *this;
   }
```

12.1.3 *getStorageSize*

This function computes how many space a single node will require on disc.

```
  static size_t getStorageSize(){
     return sizeof(CcInt::inttype) + 2*sizeof(SmiRecordId)
```

```
            + sizeof(htype);
}

// getter and setter methods
CcInt::inttype getValue()const { return value; }
htype getHeight() const { return height; }
SmiRecordId getLeft() const{ return left; }
SmiRecordId getRight() const { return right; }
SmiRecordId getID() const{ return myId; }
void setId(SmiRecordId& id) { myId = id; }
void setLeft(const SmiRecordId id) { left = id; }
void setRight(const SmiRecordId id) { right = id; }
void setHeight(const htype h){ height = h; }
```

12.1.4 *append*

This functions appends a node which is not already part of a record file to a record file. The id of the node will be changed to the used record id. This id is also the return value of this method.

```
SmiRecordId append(SmiRecordFile& file){
   // allow only non-written node to be appended
   assert(myId == 0);
   SmiRecord record;
   SmiRecordId id;
   file.AppendRecord(id, record);
   myId = id;
   assert(id!=0);
   writeToRecord(record);
   record.Finish();
   return id;
}
```

12.1.5 *update*

Overwrites the data stored on disc by the current values of this node.

```
void update(SmiRecordFile& file) const{
   assert(myId != 0);
   SmiRecord record;
   file.SelectRecord(myId, record,SmiFile::Update);
   writeToRecord(record);
   record.Finish();
}


private:
  CcInt::inttype value;
  SmiRecordId left;
  SmiRecordId right;
  htype height;
  SmiRecordId myId;
```

12.1.6 *writeToRecord*

This function writes the current values of this node to the given record.

```
    void writeToRecord(SmiRecord& r)const{
      size_t offset = 0;
      r.Write(&value,sizeof(CcInt::inttype),offset);
      offset += sizeof(CcInt::inttype);
      r.Write(&left,sizeof(SmiRecordId),offset);
      offset += sizeof(SmiRecordId);
      r.Write(&right,sizeof(SmiRecordId),offset);
      offset += sizeof(SmiRecordId);
      r.Write(&height,sizeof(htype),offset);
    }

};
```

12.2 Tree Class

The class tree contains the file and the *RecordID* of the root node. If the tree is empty, this id is 0. To be able to store also undefined *CcInt* values, a flag is used whether a undefined value is part of the tree or not. Additionally, the number of entries is stored.

```
class PAVLTree{
 public:
```

12.2.1 Constructors

The *PAVLTree* class provides two constructors. The first one creates a new *PAVLTree* including the record file storing the tree. The tree will be empty. The second variant is used to open an existing *PAVLTree*. Details of these constructors can be found at their implementations.

```
  PAVLTree();

  PAVLTree(SmiFileId fileId,
           SmiRecordId rootId,
           bool containsUndef,
           size_t noEntries);
```

12.2.2 Destructor

Each opened file must be closed at the end of a Secondo session. Otherwise the database directory may be corrupt. Hence is is strongly required to close the file in the destructor.

```
  ~PAVLTree(){
    if(file.IsOpen()){
      file.Close();
    }
  }
```

The usual *BasicType* function.

```
  static string BasicType(){
     return "pavl";
  }
```

The ususal *checkType* function.

```
  static bool checkType(ListExpr args){
     return nl->IsEqual(args,BasicType());
  }
```

Declarations of functions for inserting elements and test for containtness.

```
  bool insert(CcInt* value);

  bool contains(CcInt* value);
```

If a database object is deleted, all corresponding files must also be removed from disc. The deletion of the file is done in the next function. Before a file can be removed from disc (using the *Drop* function), it must be closed.

```
  void deleteFile(){
     if(file.IsOpen()){
         file.Close();
     }
     file.Drop();
  }
```

Functions for converting into and from nested lists.

```
  bool readFrom(ListExpr args);
  ListExpr toListExpr();
```

The clean function removes all elements from a tree.

```
  void clean();
```

Some Getter functions.

```
  SmiFileId getFileID() { return file.GetFileId() ; }
  SmiRecordId getRootID() const { return rootId; }
  bool getContainsUndef() const { return containsUndef; }
  size_t getNoEntries() const { return noEntries; }

  // clone function
  PAVLTree* clone();

private:
   SmiRecordFile file;
   SmiRecordId rootId;
   bool containsUndef;
   size_t noEntries;
```

Auxiliary functions

```
   bool readFrom(ListExpr args, SmiRecordId& r, int& min, int& max);

   ListExpr toListExpr(SmiRecordId root);

   SmiRecordId clone( SmiRecordId root, PAVLTree* res);
```

```
Node getNode(SmiRecordId id);



// returns the height of the subtree specified by id
Node::htype getHeight(SmiRecordId id){
   if(id==0) return 0;
   return getNode(id).getHeight();
}

// computes the balance value for given heights
static int balanceVal( const Node::htype left, Node::htype right){
   if(left > right){
     return left - right;
   } else {
     return -1 * ((int) ( right - left));
   }
}

// returns the balance value for a specified node
int balanceVal(const Node& n){
   return balanceVal(getHeight(n.getLeft()), getHeight(n.getRight()));
}

// balances an unbalanced subtree
void balance(Node& badNode, stack<Node>& parents);

// recomputes the height of a node and writes the changed node to file
void updateHeight(Node n){
   Node::htype h1 = getHeight(n.getLeft());
   Node::htype h2 = getHeight(n.getRight());
   n.setHeight( max(h1,h2) + 1);
   n.update(file);
}

// sets a new root for the subtree at the top of
// predecessors. If the stack is empty, the root of the entire
// tree is changed. Additionally, the heights of all nodes
// within predecessors are updated.
void setRoot(Node& newRoot, stack<Node>& predecessors){
   // special case: new root of the entire tree
   if(predecessors.empty()){
      rootId = newRoot.getID();
      updateHeight(getNode(rootId));
      return;
   }

   Node f = predecessors.top();
   predecessors.pop();
   if(newRoot.getValue() < f.getValue()){
     f.setLeft(newRoot.getID());
   } else {
     f.setRight(newRoot.getID());
   }
   updateHeight(f);
   while(!predecessors.empty()){
      Node p = predecessors.top();
```

```
        predecessors.pop();
        updateHeight(p);
      }
    }
}; // end of class PAVLTree
```

12.2.3 First constructor

This constructor creates a new empty avl-tree. The file containing the nodes of the tree must be created. Because each node has a fixed size, we can use a *SmiRecordFile* using fixed size records. The size is requested from the *Node* class.

```
PAVLTree::PAVLTree():
      file(true,  Node::getStorageSize()),
      rootId(0), containsUndef(false), noEntries(0){
  file.Create();
}
```

12.2.4 Second constructor

This constructor is used for opening an existing tree.

```
PAVLTree::PAVLTree(SmiFileId fileID,
                   SmiRecordId _rootId,
                   bool _containsUndef,
                   size_t _noEntries):
   file(true), rootId(_rootId), noEntries(_noEntries) {
   file.Open(fileID);
}
```

12.2.5 Clone

This function creates a depth clone of the tree.

```
PAVLTree* PAVLTree::clone(){
  PAVLTree* res = new PAVLTree();
  res->containsUndef = containsUndef;
  res->noEntries = noEntries;
  // call recursive clone function
  res->rootId = clone(rootId, res);
  return res;
}
```

This functions clones a subtree given by root. The function returns the *RecordId* of the newly created root in *res*.

```
SmiRecordId PAVLTree::clone(SmiRecordId root, PAVLTree* res){
  if(root == 0){
    return 0;
  }
  Node n = getNode(root);
  n.setLeft( clone(n.getLeft(),res));
  n.setRight( clone(n.getRight(),res));
  return n.append(res->file);
}
```

12.2.6 getNode

This function creates the main memory representation of a node stored within a certain record.

```
Node PAVLTree::getNode(SmiRecordId id){
    assert(id!=0);
    SmiRecord record;
    file.SelectRecord(id,record);
    Node res(record);
    record.Finish();
    return res;
}
```

12.2.7 Insert function

This function inserts a new value into the tree. Whereas undefined values are handled separately, normal values are inserted into the file. The return value corresponds to the success of this operation, i.e. whether the value was not already part of the tree.

```
bool PAVLTree::insert(CcInt* value){
  if(!value->IsDefined()){
    bool res = !containsUndef;
    containsUndef=true;
    if(res){ noEntries++;}
    return res;
  }
  CcInt::inttype v = value->GetValue();
  // create a new node to be inserted
  Node n(v);
  // special case: empty tree
  if(rootId==0){
     rootId = n.append(file);
     noEntries = 1;
     return true;
  }
  // search insertion position storing the path  into a stack
  stack<Node> s;
  Node cn = getNode(rootId);
  if(cn.getValue()==v){ // value already exists
   return false;
  }
  SmiRecordId son = cn.getValue()>v?cn.getLeft():cn.getRight();
  while(son != 0){
     s.push(cn);
     cn = getNode(son);
     if(cn.getValue()==v){
        return false;
     }
     son = cn.getValue()>v?cn.getLeft():cn.getRight();
  }

  // found insertion position, append record
  SmiRecordId id = n.append(file);
  if(cn.getValue()>v){
    cn.setLeft(id);
  } else {
```

```
    cn.setRight(id);
  }
  cn.update(file);
  s.push(cn);
  // now, we have to correct the heights of the predecessors
  while(!s.empty()){
    Node parent = s.top();
    s.pop();
    Node::htype leftHeight = getHeight(parent.getLeft());
    Node::htype rightHeight = getHeight(parent.getRight());
    if(abs(balanceVal(leftHeight, rightHeight)) > 1){
       // correct unbalanced node
       balance(parent,s);
       return true;
    }
    parent.setHeight(max(leftHeight,rightHeight)+1);
    parent.update(file);
  }
  noEntries++;
  return true;
}
```

12.2.8 Balance of an unbalanced subtree

```
void PAVLTree::balance( Node& root, stack<Node>& predecessors){
   if(balanceVal(root) == -2){ // right subtree heigher
       Node r = getNode(root.getRight());
       if(balanceVal(r) == -1){ // left rotation
          Node y = r;
          SmiRecordId b = y.getLeft();
          root.setRight(b);
          updateHeight(root);
          y.setLeft(root.getID());
          updateHeight(y);
          setRoot(y,predecessors);
          return;
       } else { // right left rotation
          Node x = root;
          Node z = getNode(x.getRight());
          Node y = getNode(z.getLeft());
          SmiRecordId  B1 = y.getLeft();
          SmiRecordId  B2 = y.getRight();
          x.setRight(B1);;
          updateHeight(x);
          z.setLeft(B2);
          updateHeight(z);
          y.setLeft(x.getID());
          y.setRight(z.getID());
          updateHeight(y);
          setRoot(y,predecessors);
          return;
       }
   } else if(balanceVal(root)==2){ // left subtree is height
      Node left = getNode(root.getLeft());
      if(balanceVal(left) == 1){  // right rotation
        Node  y = getNode(root.getLeft());
```

```
        SmiRecordId B = y.getRight();
        SmiRecordId C = root.getRight();
        root.setLeft(B);
        root.setRight(C);
        updateHeight(root);
        y.setRight(root.getID());
        updateHeight(y);
        setRoot(y,predecessors);
        return;
      } else {  // leftRightRotation
         Node x = root;
         Node z = getNode(root.getLeft());;
         Node y = getNode(z.getRight());
         SmiRecordId  A = z.getLeft();
         SmiRecordId  B = y.getLeft();
         SmiRecordId  C = y.getRight();
         z.setLeft(A);
         z.setRight(B);
         updateHeight(z);
         x.setLeft(C);
         updateHeight(x);
         y.setLeft(z.getID());
         y.setRight(x.getID());
         updateHeight(y);
         setRoot(y,predecessors);
         return;
      }
   } else {
      assert(false);
   }
}
```

12.2.9 *contains*

This function checks whether a value is part of the tree.

```
bool PAVLTree::contains(CcInt* value){

  if(!value->IsDefined()){
    return containsUndef;
  }
  SmiRecordId son = rootId;
  int v = value->GetValue();
  while(son){
    Node n = getNode(son);
    if(n.getValue()== v){
       return true;
     }
     son = n.getValue()>v?n.getLeft():n.getRight();
  }
  return false;
}

void PAVLTree::clean(){
   file.Truncate();
   rootId = 0;
   noEntries = 0;
}
```

12.2.10 *readFrom*

This function reads a nested list and stores the tree represented by the list into the file. The id of the subtree's root is returned in the output parameter *rootId*. Furthermore, the minimum and the maximum value of the tree are returned in the appropriate output parameters. If the list represents a valid AVL-tree, the return value of this function is true, false otherwise.

```
bool PAVLTree::readFrom(ListExpr list, SmiRecordId& rootId,
                        int& min, int& max){
  if(nl->AtomType(list)!=NoAtom){
     cmsg.inFunError("wrong list format");
     return false;
  }
  if(nl->IsEmpty(list)){
    rootId=0;
    return true;
  }

  if(!nl->HasLength(list,3)){
     cmsg.inFunError("wrong list format");
     return false;
  }
  if(nl->AtomType(nl->First(list))!=IntType ||
     nl->AtomType(nl->Second(list))!=NoAtom ||
     nl->AtomType(nl->Third(list))!=NoAtom){
     clean();
     cmsg.inFunError("wrong list format");
     return false;
  }

  int v = nl->IntValue(nl->First(list));

  min = v;
  max = v;
  int min1 = v;


  SmiRecordId id1;
  if(!readFrom(nl->Second(list),id1,min,max)){
    return false;
  }
  if(id1){
    if(max>v){
       return false;
    }
    min1  = min;
  }
  SmiRecordId id2;
  if(!readFrom(nl->Third(list),id2,min,max)){
    return false;
  }
  if(id2){
     if(min<v){
        return false;
     }
  }
  min = min1;
```

```
  Node n(nl->IntValue(nl->First(list)));
  n.setLeft(id1);
  n.setRight(id2);
  Node::htype h1 = getHeight(n.getLeft());
  Node::htype h2 = getHeight(n.getRight());
  n.setHeight(std::max(h1,h2) + 1);
  if(abs(balanceVal(h1,h2))>1){
     cmsg.inFunError("unbalanced tree found");
     clean();
     return false;
  }
  rootId = n.append(file);
  return true;
}
```

12.2.11 *readFrom*

This function converts a nested list into a tree. The return value is *true* if the list represents a valid AVL tree, *false* otherwise.

```
bool PAVLTree::readFrom(ListExpr list){
   if(rootId){
     clean();
   }
   if(!nl->HasLength(list,2)){
      return false;
   }
   if(nl->AtomType(nl->First(list))!=BoolType){
      return false;
   }
   containsUndef = nl->BoolValue(nl->First(list));
   list = nl->Second(list);
   int min, max;
   if(!readFrom(list,rootId,min,max)){
      clean();
      return false;
   }
   return true;
}
```

12.2.12 toListExpr

This function converts the tree into a list.

```
ListExpr PAVLTree::toListExpr(){

 return nl->TwoElemList(nl->BoolAtom(containsUndef),
                        toListExpr(rootId));
}


ListExpr PAVLTree::toListExpr(SmiRecordId root){
  if(root==0){
    return nl->TheEmptyList();
  }
```

```
  Node n = getNode(root);
  return nl->ThreeElemList(
            nl->IntAtom(n.getValue()),
            toListExpr(n.getLeft()),
            toListExpr(n.getRight()));

}
```

12.3 Secondo Supporting Functions

12.3.1 Property function

```
ListExpr PAVLProperty(){
  return ( nl->TwoElemList (
      nl->FourElemList (
        nl->StringAtom("Signature"),
        nl->StringAtom("Example Type List"),
        nl->StringAtom("List Rep"),
        nl->StringAtom("Example List")),
      nl->FourElemList (
        nl->StringAtom("-> SIMPLE"),
        nl->StringAtom(PAVLTree::BasicType()),
        nl->StringAtom("(value left right)"),
        nl->StringAtom("(5 (3 () 4) (7)) ")
      )));
}
```

12.3.2 In-function

```
Word InPAVL(const ListExpr typeInfo, const ListExpr instance,
            const int errorPos, ListExpr & errorInfo, bool& correct){

   PAVLTree* tree = new PAVLTree();
   assert(!tree->getRootID());
   if(!tree->readFrom(instance)){
      tree->deleteFile();
      delete tree;
      correct = false;
      return Word((void*)0);
   } else {
      correct = true;
      return Word(tree);
   }
}
```

12.3.3 Out function

```
ListExpr OutPAVL(ListExpr typeInfo, Word value){
 return ((PAVLTree*)value.addr)->toListExpr();
}
```

12.3.4 Create

```
Word CreatePAVL(const ListExpr typeInfo){
   return Word(new PAVLTree);
}
```

12.3.5 Delete

Here the complete object instance is removed inclusive the file.

```
void DeletePAVL(const ListExpr typeInfo, Word& w){
  PAVLTree* t = (PAVLTree*) w.addr;
  t->deleteFile();
  delete t;
  w.addr = 0;
}
```

12.3.6 OPEN

Within the open function, we extract all required information from the record to build the main structure of the tree. The tree itself is keept in the file without loading it into main memory.

```
bool OpenPAVL ( SmiRecord & valueRecord ,
       size_t & offset , const ListExpr typeInfo ,
       Word & value ){
   SmiFileId fileId;
   SmiRecordId rootId;
   bool containsUndef;
   size_t noEntries;
   bool ok = valueRecord.Read(&fileId, sizeof(SmiFileId),offset);
   offset += sizeof(SmiFileId);
   ok = ok && valueRecord.Read(&rootId, sizeof(SmiRecordId), offset);
   offset +=  sizeof(SmiRecordId);
   ok = ok && valueRecord.Read(&containsUndef,sizeof(bool),offset);
   offset +=sizeof(bool);
   ok = ok && valueRecord.Read(&noEntries,sizeof(size_t), offset);
   offset += sizeof(size_t);
   if(ok){
     value.addr = new PAVLTree(fileId, rootId, containsUndef,noEntries);
   } else {
     value.addr = 0;
   }
   return ok;
}
```

12.3.7 SAVE

The save function stores the main information into the record. The tree's file is not required (except its id).

```
bool SavePAVL( SmiRecord & valueRecord , size_t & offset ,
     const ListExpr typeInfo , Word & value ) {
  PAVLTree* t = (PAVLTree*) value.addr;
  SmiFileId fileId = t->getFileID();
  SmiRecordId rootId = t->getRootID();
  bool containsUndef = t->getContainsUndef();;
  size_t noEntries = t->getNoEntries();
  bool ok = valueRecord.Write(&fileId, sizeof(SmiFileId), offset);
  offset += sizeof(SmiFileId);
  ok = ok && valueRecord.Write(&rootId, sizeof(SmiRecordId), offset);
  offset += sizeof(SmiRecordId);
```

```
  ok = ok && valueRecord.Write(&containsUndef, sizeof(bool), offset);
  offset += sizeof(bool);
  ok = ok && valueRecord.Write(&noEntries, sizeof(size_t), offset);
  offset += sizeof(size_t);
  return ok;
}
```

12.3.8 CLOSE

Here, the disc part remains untouched.

```
void ClosePAVL ( const ListExpr typeInfo , Word & w ) {
   PAVLTree* t = (PAVLTree*) w.addr;
   delete t;
   w.addr = 0;
}
```

12.3.9 Clone

```
Word ClonePAVL ( const ListExpr typeInfo , const Word & w ){
  PAVLTree* t = (PAVLTree*) w.addr;
  return Word(t->clone());
}
```

12.3.10 Cast

```
void * CastPAVL ( void * addr ) {
   return addr;
}


bool PAVLTypeCheck ( ListExpr type , ListExpr & errorInfo ){
    return nl->IsEqual ( type , PAVLTree::BasicType ());
}

int SizeOfPAVL () {
    return sizeof(SmiFileId) + sizeof(SmiRecordId)
           + sizeof(bool) + sizeof(size_t);;
}


TypeConstructor PAVLTC (
PAVLTree::BasicType (), // name of the type
PAVLProperty , // property function
OutPAVL , InPAVL , // out and in function
0, 0, // deprecated , don not think about it
CreatePAVL , DeletePAVL , // creation and deletion
OpenPAVL , SavePAVL , // open and save functions
ClosePAVL , ClonePAVL , // close and clone functions
CastPAVL , // cast function
SizeOfPAVL , // sizeOf function
PAVLTypeCheck );
 // type checking functi
```

12.4 Operators Creating a PAVL Tree

12.4.1 Creation of a pavltree

```
ListExpr createPAVLTM(ListExpr args){
  string err = "stream(int) expected";
  if(!nl->HasLength(args,1)){
    return listutils::typeError(err);
  }
  if(!Stream<CcInt>::checkType(nl->First(args))){
     return listutils::typeError(err);
  }
  return listutils::basicSymbol<PAVLTree>();
}

int createPAVLVM ( Word * args , Word & result , int message ,
                   Word & local , Supplier s ) {

  result = qp->ResultStorage(s);
  PAVLTree* res = (PAVLTree*) result.addr;
  Stream<CcInt> stream(args[0]);
  stream.open();
  CcInt* elem;
  while( (elem = stream.request()) != 0){
     res->insert(elem);
     elem->DeleteIfAllowed();
  }
  stream.close();
  return 0;
}

OperatorSpec createPAVLSpec (
    " stream(int)  -> pavl " ,
    " _ createPAVL  " ,
    " creates an avl-tree from an integer stream " ,
    " query intstream(1,100) createPAVL  "
);


Operator createPAVLOp (
  "createPAVL" , // name of the operator
  createPAVLSpec.getStr() , // specification
  createPAVLVM , // value mapping
  Operator::SimpleSelect , // selection function
  createPAVLTM // type mapping
);
```

12.4.2 Checking for containtness

```
ListExpr containsTM(ListExpr args){
  string err = "pavl x int expected";
  if(!nl->HasLength(args,2)){
    return listutils::typeError(err);
  }
  if(!PAVLTree::checkType(nl->First(args))
    || !CcInt::checkType(nl->Second(args))){
     return listutils::typeError(err);
```

```
  }
  return listutils::basicSymbol<CcBool>();
}

int containsVM ( Word * args , Word & result , int message ,
                 Word & local , Supplier s ) {

  result = qp->ResultStorage(s);
  CcBool* res = (CcBool*) result.addr;
  PAVLTree* a1 = (PAVLTree*) args[0].addr;
  CcInt* a2 = (CcInt*) args[1].addr;
  res->Set(true, a1->contains(a2));
  return 0;
}

OperatorSpec containsSpec (
    "  -> pavl x int -> bool" ,
    " _ contains _  " ,
    " Checks whether an avl tree contains an int " ,
    " query p1 contains 23  "
);


Operator containsOp (
  "contains" , // name of the operator
  containsSpec.getStr() , // specification
  containsVM , // value mapping
  Operator::SimpleSelect , // selection function
  containsTM // type mapping
);
```

13 Update Operators

The update command in Secondo replaces a stored object completely by another one. Sometimes it is required to modify an existing object, e.g. for inserting new tuples into an existing relation. To realize that, update operators have to be written. Such operators manipulate their arguments, which is quite unsual for Secondo operators. To make the changes persistent, the manipulated argument must be marked. otherwise the changes are not written back to the disc.

The implementation of update operators is explained at the example of an *insert* operator inserting new elementes into an existing avl-tree.

13.1 Type Mapping

The type mapping of an update operator has no special features.

```
ListExpr insertTM(ListExpr args){
  string err = "stream(int) x pavl expected";
  if(!nl->HasLength(args,2)){
    return listutils::typeError(err);
  }
  if(   !Stream<CcInt>::checkType(nl->First(args))
     || !PAVLTree::checkType(nl->Second(args))){
    return listutils::typeError(err);
  }
```

```
  // the number of new elements is returned
  return listutils::basicSymbol<CcInt>();
}
```

13.2 Value Mapping

Within the value mapping the tree argument is manipulated. At the end of this operator, this argument is marked as modified. For stream operators, this marking can be done within the *CLOSE* section. In non-stream operators like here, this is done somewhere in the value mapping implementation.

```
int insertVM ( Word * args , Word & result , int message ,
                   Word & local , Supplier s ) {
  result = qp->ResultStorage(s);
  CcInt* res = (CcInt*) result.addr;
  Stream<CcInt> stream(args[0]);
  PAVLTree* avl = (PAVLTree*) args[1].addr;
  int count = 0;
  stream.open();
  CcInt* elem;
  while( (elem=stream.request())!=0){
     if(avl->insert(elem)){
       count++;
     }
     elem->DeleteIfAllowed();
  }
  stream.close();
  // mark the argument as modified
  qp->SetModified(qp->GetSon(s, 1));
  res->Set(true,count);
  return 0;
}
```

13.3 Specification and Operator Instance

Here, no specials must be considered.

```
OperatorSpec insertSpec (
    "  stream(int) x pavl -> int " ,
    " _ _ insert " ,
    " inserts new elements into an exixisting avl tree" ,
    " query instream (1,200) p1 insert "
);


Operator insertOp (
  "insert" , // name of the operator
  insertSpec.getStr() , // specification
  insertVM , // value mapping
  Operator::SimpleSelect , // selection function
  insertTM // type mapping
);
```

14 Accessing values in Type Mappings

Sometimes it is necessary to know a value of an object for computing the result type. An example is an import operator reading some object from a file. If the file contains the type, the filename (a string or a text) must be known in the type mapping. Secondo provides a possibility to get not only the types in the argument list of the type mapping but additionally the part of the query forming this type. To enable this feature for an operator, the function *SetUsesArgsInTypeMapping()* must be called for this operator. This happens within the Algebra constructor.

This feature is explained at the example of the *importObject* operator. This operator gets a filename and returns the object located in the file. The object is coded in the way like *save object to ...* it does.

14.1 Type Mapping

If the argument feature is enabled for an operator, each argument is not longer only described by its type, but by a list (<type> <expression>) where <expression> corresponds to the part of the query forming the argument.

```
ListExpr importObjectTM(ListExpr args){
  string err="text expected";

  if(!nl->HasLength(args,1)){
    return listutils::typeError("expected one argument");
  }
  ListExpr arg = nl->First(args);
  // the list is coded as (<type> <query part>)
  if(!nl->HasLength(arg,2)){
     return listutils::typeError("internal error");
  }

  if(!FText::checkType(nl->First(arg))){
    return listutils::typeError(err);
  }

  ListExpr fn = nl->Second(arg);
   if(nl->AtomType(fn)!=TextType){
      return listutils::typeError("file name not constant");
   }
   string fileName = nl->Text2String(fn);

   ListExpr objectList;
   if(! nl->ReadFromFile(fileName, objectList)){
      return listutils::typeError("file not found or "
                             "file does not contain a list");
   }
   // an object file is formatted as
   // (OBJECT <name> <typename> <type> <value> () )
   // the typename is mostly empty the last, empty list is for future extensions

   // check structure
   if(   !nl->HasLength(objectList,6)
      || !listutils::isSymbol(nl->First(objectList),"OBJECT")){
```

```
    return listutils::typeError("file does not contain a "
                                "valid object description");
  }

  ListExpr typeList = nl->Fourth(objectList);

  // check whether this list is a valid type description
  SecondoCatalog* ctl = SecondoSystem::GetCatalog();
  string tname;
  int algId, typeId;
  if(!ctl->LookUpTypeExpr(typeList, tname, algId, typeId)){
    return listutils::typeError("Invalid  type description in file");
  }
  return typeList;

}
```

14.2 Value Mapping

The value mapping is the same as for normal operators.

```
int importObjectVM ( Word * args , Word & result , int message ,
                     Word & local , Supplier s ) {
  FText* fileName = (FText*) args[0].addr;
  if(!fileName->IsDefined()){
    return 0;
  }
  string fn = fileName->GetValue();
  ListExpr objectList;
  if(!nl->ReadFromFile(fn,objectList)){
    return 0;
  }
  if(!nl->HasLength(objectList,6)){
     return 0;
  }
  ListExpr typeList = nl->Fourth(objectList);
  ListExpr instance = nl->Fifth(objectList);
  SecondoCatalog* ctlg = SecondoSystem::GetCatalog();
  string tname;
  int algId, typeId;
  if(!ctlg->LookUpTypeExpr(typeList, tname, algId, typeId)){
    return 0;
  }
  InObject in = am->InObj(algId, typeId);
  int errorPos=0;
  ListExpr errorInfo = listutils::emptyErrorInfo();
  bool correct;

  Word o = in(ctlg->NumericType(typeList), instance,
              errorPos, errorInfo,correct);
  if(!correct){
     return 0;
  }
  // Because the object may be quite large, we replace it instead
  // of manipulating the stored object
  qp->DeleteResultStorage(s);
```

```
    qp->ChangeResultStorage(s, o);
    qp->SetDeleteFunction(s, am->DeleteObj(algId, typeId));
    result = qp->ResultStorage(s);
    return 0;
}
```

14.3 Specification

There are no news within this part.

```
OperatorSpec importObjectSpec (
    "  text -> X " ,
    " importObject(_) " ,
    " Returns the object located in a file named by the argument." ,
    " query importObject('Kinos.obj') "
);
```

14.4 Operator Instance

Also the definition of the operator instance is as usual.

```
Operator importObjectOp (
  "importObject" , // name of the operator
  importObjectSpec.getStr() , // specification
  importObjectVM , // value mapping
  Operator::SimpleSelect , // selection function
  importObjectTM // type mapping
);
```

14.5 More Flexible Variant of Accessing Values in Type Mappings

The operator importObject only accepts a constant text as its input. Sometimes, the argument is build by an expression build from constants and database objects. For example, if there is a database object called *basename* of type text, it would be desirable to import an object using the query :

```
query importObject(basename + 'obj')
```

Because importObject expects that the expression is a constant text, this is not possible. Here, a variant of the *importObject* operator is described supporting this kind of input. We call the operator *importObject2*. Because the value mapping as well as the specification is the same as for *importObject*, we reuse these implementations when defining the operator instance.

The main idea behind this operator is to use the Queryprocessor for the evaluation of the expression building the argument. The *QueryProcessor* provides a function *ExecuteQuery* for this job.

```
ListExpr importObject2TM(ListExpr args){
  string err="text expected";

  if(!nl->HasLength(args,1)){
    return listutils::typeError("expected one argument");
  }
  ListExpr arg = nl->First(args);
  // the list is codes as (<type> <query part>)
  if(!nl->HasLength(arg,2)){
     return listutils::typeError("internal error");
  }

  if(!FText::checkType(nl->First(arg))){
    return listutils::typeError(err);
  }
  // here, accessing the value is changed

   ListExpr expression = nl->Second(arg);

   // we need some variables for feeding the ExecuteQuery function
   Word queryResult;
   string typeString = "";
   string errorString = "";
   bool correct;
   bool evaluable;
   bool defined;
   bool isFunction;
   // use the queryprocessor for executing the expression
   qp->ExecuteQuery(expression, queryResult,
                    typeString, errorString, correct,
                    evaluable, defined, isFunction);
   // check correctness of the expression
   if(!correct || !evaluable || !defined || isFunction){
      assert(queryResult.addr == 0);
      return listutils::typeError("could not extract filename ("+
                                  errorString + ")");
   }
   FText* fn = (FText*) queryResult.addr;
   assert(fn);
   if(!fn->IsDefined()){
      fn->DeleteIfAllowed();
      return listutils::typeError("filename undefined");
   }
   string fileName = fn->GetValue();
   fn->DeleteIfAllowed();

   // frome here, it's the same  as for importObject

   ListExpr objectList;
   if(! nl->ReadFromFile(fileName, objectList)){
      return listutils::typeError("file not found or "
                            "file does not contain a list");
   }
   // an object file is formatted as
   // (OBJECT <name> <typename> <type> <value> () )
   // the typename is mostly empty the last empty list is for future extensions
```

```
  // check structure
  if(   !nl->HasLength(objectList,6)
     || !listutils::isSymbol(nl->First(objectList),"OBJECT")){
    return listutils::typeError("file does not contain a "
                                "valid object description");
  }
  ListExpr typeList = nl->Fourth(objectList);
  // check whether this list is a valid type description
  SecondoCatalog* ctl = SecondoSystem::GetCatalog();
  string tname;
  int algId, typeId;
  if(!ctl->LookUpTypeExpr(typeList, tname, algId, typeId)){
    return listutils::typeError("Invalid  type description in file");
  }
  return typeList;
}
```

14.6 Operator Instance

Here, we use the specification and the value maping for importObject.

```
Operator importObject2Op (
  "importObject2" , // name of the operator
  importObjectSpec.getStr() , // specification
  importObjectVM , // value mapping
  Operator::SimpleSelect , // selection function
  importObject2TM // type mapping
);
```

15 Compact Storing of Attribute Data Types

The default implementation of storing attributes to disc within a relation is to build a byte block from the class and perform a special treatment for the contained flobs. Because an attribute contains some reference counters and other stuff, the storage size is much more than necessary. For example, an 32 bit int value has a storage size of 16 bytes, but needs only 5 bytes (defined + value) for its representation. On the other hand there are small size restricted types, where the actual value varyies in size. For example, a string data type may contain up to 48 characters but the most strings are much shorter.

Secondo provides mechanisms for storing types within relations more efficently than the standard storage procedure. These mechanisms work only for attributes without any FLOB members.

For understanding the mechanism, the tuple representation on disc must be explained. A tuple on disc consists of two parts. The first part is called the core part, the second one is the extension part. Both parts are stored together as a single byte block to disc. The core part of the tuple consists of the core parts of the attributes, written direcly one after each other. The extension part contains the extension parts of the attributes and may be empty.

Obviously, we have to clarify the core and extension part of an attribute. Firstly, we describe the parts if the default mechanism is used. Afterwards, the changes for special storage mechanisms are described.

The core part of an attribute for the default storage consists of the byte block represented by the attribut's class. This is also called root record of the attribute. The extension part of an attribute without any flobs is empty. If the attribute has flobs with a size smaller than a threshold, these flob data build the extension part. The threshold is defined in the file *RelationCommon.cpp* in static tuple variable *Tuple::extensionLimit*.

There are three different methods for storing an attribut in a relation:

- Default: This method corresponds to the storage mechanism above. This mechanism is used for all attribute data types above.
- Core: The storage size is fixed, there is no extension part. The attribute class provides its own serialization method. This istorage method is suitable for small fixed size attributes like number representations.
- Extension: The storage size may vary within a small range. The attribute provides functions storing the core part and the extension part. This may be used for attribute data types like string etc.

15.1 Using Core Storage

The next class uses the core storage mechanism. We describe the implementation of the *ushort* type representing an unsigned integer with 2 bytes length. We use an additional byte for representing the undefined state.

```
class UShort: public Attribute{
  public:
     typedef uint16_t inttype;
     // constructors
     UShort() {}
     UShort(const inttype v): Attribute(true), value(v) {}
     UShort(const bool def): Attribute(def), value(0) {}
     UShort(int dummy): Attribute(false),value(0){}
     UShort(const UShort& rhs): Attribute(rhs), value(rhs.value){}
     // assignment operator
     UShort& operator=(const UShort& src){
        SetDefined(src.IsDefined());
        value = src.value;
        return *this;
     }
     // some class functions
     inttype GetValue() const{
       return value;
     }
     void Set(const bool def, const inttype v){
        SetDefined(def);
        value = v;
     }
     // auxiliary functions
     static const string BasicType(){ return "ushort"; }
     static const bool checkType(const ListExpr e){
       return listutils::isSymbol(e,BasicType());
     }
     // attribute supporting types
```

```
virtual int NumOfFLOBS() const{
   return 0;
}
virtual Flob* GetFLOB(const int i){
   return 0;
}
int Compare(const Attribute* arg) const{
  if(!IsDefined()){
    return arg->IsDefined()?-1:0;
  }
  if(!arg->IsDefined()){
     return 1;
  }
  inttype v = ((UShort*)arg)->GetValue();
  if(value < v) return -1;
  if(value > v) return 1;
  return 0;
}
bool Adjacent(const Attribute* arg) const{
  return false;
}
size_t Sizeof() const{
  return sizeof(*this);
}
size_t HashValue() const{
   if(!IsDefined()){
      return 0xFFFFFFu;
   } else {
      return value;
   }
}
void CopyFrom(const Attribute* arg){
   *this = *((UShort*)arg);
}
Attribute* Clone() const{
  return new UShort(*this);
}
// function supporting generic type constructors
static ListExpr Property(){
   return gentc::GenProperty("-> DATA",
                             BasicType(),
                             "int",
                             "16");
}
static bool CheckKind(ListExpr type, ListExpr& errorInfo){
   return checkType(type);
}
bool ReadFrom(ListExpr instance, const ListExpr typeInfo){
   if(listutils::isSymbolUndefined(instance)){
      SetDefined(false);
       return true;
   }
   if(nl->AtomType(instance)!=IntType){
      return false;
   }
   int v = nl->IntValue(instance);
   if(v< 0 || v>numeric_limits<inttype>::max()){
```

```
            return false;
        }
        Set(true,(inttype)v);
        return true;
    }

    ListExpr ToListExpr(ListExpr typeInfo) const{
        if(!IsDefined()){
            return listutils::getUndefined();
        }
        return nl->IntAtom(value);
    }

    // function for storage management
```

15.1.1 Function *GetStorageType*

Here, the *StorageType* is returned. Possible values are *Default*, *Core*, *Extension*, *Unspecified*, where the last mentioned one makes no sense to use. We want to store the attribute completely within the Core part of the tuple. Thus, the return value is *Core*.

```
inline virtual StorageType GetStorageType() const{
    return Core;
}
```

15.1.2 Function *SerializedSize*

This function returns the number of bytes required on disc for this type.

```
inline virtual size_t SerializedSize() const{
   return sizeof(inttype) + 1;
}
```

15.1.3 Function *Serialize*

This function writes the serialized version of the value into some buffer. We use the first byte for the defined flag and the remaining bytes store the value. The argument *sz* contains the value returned by *SerializedSize*

```
inline virtual void Serialize(char* buffer, size_t sz, size_t offset) const{

    uint8_t b = IsDefined()?1:0;
    memcpy(buffer+offset,&b,1);
    offset+=1;
    memcpy(buffer+offset,&value, sizeof(inttype));
    offset+= sizeof(inttype);
}
```

15.1.4 *Rebuild*

This function reads the value from a buffer. As for the *Serialize* function, *sz* is the size returned by *SerializedSize*. In contrast to *Serialize*, there is no offset. The value is read from the beginning of the buffer.

```
inline virtual void Rebuild(char* buffer, size_t sz){
   size_t offset=0;
   uint8_t b;
   memcpy(&b, buffer+offset, 1);
   offset += 1;
   memcpy(&value, buffer+offset, sizeof(inttype));
}

  private:
    inttype value;
};

GenTC<UShort> ushortTC;
```

15.1.5 Testing the success

There are several operators returning some sizes for attributes. Here, we can use the queries:

```
query [const ushort value 23] memattrsize

query [const ushort value 23] feed transformstream rootattrsize[Elem]
```

and be happy that the first result is greater than the second one.

16 Variable Size Attributes Without FLOBs

The storage mechanism describe in the last section can also be used for storing attribute data types of variable length. Even the usage of pointers is possible. Note that this mechanism is only for small values efficient. For bigger values (more than 512 bytes), FLOBs should be used instead of this mechanism.

We describe the mechanism at the attribute data type *vstring* representing a variable length string. Because the core size of an attribute is always fixed, we have to store the string data within the extension part. On the other hand, the generic Open and Save functions defined in the attribute class only support the default mechanism. For this reason, we have to implement the serialisation (Open and Save) functions by ourself.

```
class VString;
ostream& operator<<(ostream& o, const VString& vs);
```

```
class VString: public Attribute{
  public:
     VString() {}
     VString(const bool def): Attribute(def), value(""){}
     VString(const VString& src):Attribute(src),value(src.value) {}
     VString(const string& s):Attribute(true),value(s){}
     VString& operator=(const VString& src){
        SetDefined(src.IsDefined());
        value = IsDefined()?src.value:"";
        return *this;
     }
     void Set(const bool def, const string& v){
       SetDefined(def);
       value = IsDefined()?v:"";
     }

     string GetValue() const{
         return value;
     }

    // auxiliary functions
    static const string BasicType(){ return "vstring"; }
    static const bool checkType(ListExpr type){
      return listutils::isSymbol(type,BasicType());
    }
    // attribute related functions
    inline int NumOfFLOBs() const{
      return 0;
    }
    inline virtual Flob* GetFLOB(const int i){
       return 0;
    }
    int Compare(const Attribute* arg)const{
      if(!IsDefined()){
         return arg->IsDefined()?-1:0;
      }
      if(!arg->IsDefined()){
         return 1;
      }
      return strcmp(value.c_str(),((VString*)arg)->value.c_str());
    }
    bool Adjacent(const Attribute* arg) const{
      return false;
    }
    size_t Sizeof() const{
       return sizeof(*this);
    }
    size_t HashValue() const{
      if(!IsDefined()){
        return 0;
      }
      size_t res = 0;
      size_t st = value.length()>5u?5:value.length();
      for(size_t i=0;i<st; i++){
          res += i*255 + value[i];
      }
      return res;
```

```
}
void CopyFrom(const Attribute* arg){
   *this = *((VString*) arg);
}
Attribute* Clone() const{
   return new VString(*this);
}
// functions supporting the embedding into secondo
static ListExpr Property(){
    return gentc::GenProperty( " -> DATA",
                               BasicType(),
                               "string or text",
                               "'lang'");
}
static bool CheckKind(ListExpr type, ListExpr& errorInfo){
    return checkType(type);
}



static Word In(const ListExpr typeInfo, const ListExpr le,
            const int errorPos, ListExpr& errorInfo, bool& correct){
    Word res((void*)0);

    if(listutils::isSymbolUndefined(le)){
      res.addr = new VString(false);
      correct = true;
      return res;
    }
    if(nl->AtomType(le)==StringType){
      res.addr = new VString(nl->StringValue(le));
      correct = true;
      return res;
    }
    if(nl->AtomType(le)==TextType){
      res.addr = new VString(nl->Text2String(le));
      correct = true;
      return res;
    }
    correct = false;
    return  res;
}

static ListExpr Out(const ListExpr typeInfo, Word value){
    VString* v = (VString*) value.addr;
    if(!v->IsDefined()){
       return listutils::getUndefined();
    }
    if(v->value.length()<MAX_STRINGSIZE){
       return nl->StringAtom(v->value);
    } else {
       return nl->TextAtom(v->value);
    }
 }

 static Word Create(const ListExpr typeInfo){
   Word res( new VString(false));
```

```
  return res;
}

static void Delete(const ListExpr typeInfo, Word& v){
    delete (VString*) v.addr;
    v.addr = 0;
}

static bool Open( SmiRecord& valueRecord, size_t& offset,
                  const ListExpr typeInfo, Word& value){
   // get Size of the string
   size_t length;
   bool ok = valueRecord.Read(&length,sizeof(size_t),offset);
   if(!ok) { return false; }
   size_t buffersize = sizeof(size_t) + 1 + length;
   char* buffer = new char[buffersize];
   ok = valueRecord.Read(buffer,buffersize,offset);
   offset += buffersize;
   if(!ok){
      delete[] buffer;
      return false;
   }
   VString* v = new VString(false);
   v->Rebuild(buffer, buffersize);
   value.addr = v;
   delete[] buffer;
   return true;
}

static bool Save(SmiRecord& valueRecord, size_t& offset,
                 const ListExpr typeInfo, Word& value){
    VString* v = (VString*) value.addr;
    size_t size = v->SerializedSize();
    char* buffer = new char[size];
    v->Serialize(buffer, size,0);
    bool ok = valueRecord.Write(buffer, size,offset);
    offset += size;
    delete[] buffer;
    return ok;
}

static void Close(const ListExpr typeInfo, Word& w){
    delete (VString*) w.addr;
    w.addr = 0;
}
static Word Clone(const ListExpr typeInfo, const Word& w){
   VString* v = (VString*) w.addr;
   Word res;
   res.addr = new VString(*v);
   return res;
}

static int Size() {
   return 256;
}
```

Because the Serialization does not overwrite internal function pointers as in the default mecha-

nism, here just the argument pointer is returned. Because the Standard constructir of the *string* class initilizes the string to be empty, we cannot use the special variant of *new* here.

```
static void* Cast(void* addr){
    return addr;
}

static bool TypeCheck(ListExpr type, ListExpr& errorInfo){
   return checkType(type);
}
```

Because of the variable length, we have to store the value within the extension part of a tuple.

```
inline virtual StorageType GetStorageType() const{
   return Extension;
}
```

The function *SerializedSize* is implemented as for the *Core* variant.

```
inline virtual size_t SerializedSize() const{
    return sizeof(size_t) + 1 + value.length();
}

inline virtual void Serialize(char* buffer, size_t sz,
                              size_t offset) const{
    size_t length = value.length();
    uint8_t def = IsDefined()?1:0;
    memcpy(buffer+offset, &length , sizeof(size_t));
    offset += sizeof(size_t);
    memcpy(buffer+offset, &def, 1);
    offset += 1;
    memcpy(buffer+offset, value.c_str(), length);
    offset += length;
}

inline virtual void Rebuild(char* buffer, size_t sz){
    size_t length;
    uint8_t def;
    size_t offset = 0;
    memcpy(&length, buffer + offset, sizeof(size_t));
    offset += sizeof(size_t);
    memcpy(&def, buffer+offset, 1);
    offset+=1;
    if(!def){
       value = "";
       SetDefined(false);
    } else {
       this->value = string(buffer+offset,length);
       SetDefined(true);
    }
}
```

GetMemSize

The *GetMemSize* function returns the amount of space required for an object in main memory. The default implementation uses the size of the root block and adds the FlobSizes of FLOBs whose data are not controlled by the FLOBCache. Because in the VString class pointers are present, we have to overwrite this function for returning a correct result.

```
    virtual size_t GetMemSize() {
       return sizeof(*this) + value.length();
    }


  private:
    string value;    // uses pointers internally

};

ostream& operator<<(ostream& o, const VString& vs){
  if(!vs.IsDefined()){
    o << "undef";
  } else {
    o << "'"<<vs.GetValue()<<"'";
  }
  return o;
}
```

16.1 Type constructor instance

```
TypeConstructor VStringTC(
  VString::BasicType(),
  VString::Property, VString::Out, VString::In, 0,0,
  VString::Create, VString::Delete,
  VString::Open, VString::Save, VString::Close, VString::Clone,
  VString::Cast, VString::Size, VString::TypeCheck
);
```

17 Operator text2string

This operator is for testing the vstring implementation. It takes a *text* and converts it into a *vstring*.

```
ListExpr text2vstringTM(ListExpr args){
  if(!nl->HasLength(args,1) || !FText::checkType(nl->First(args))){
    return listutils::typeError("text expected");
  }
  return listutils::basicSymbol<VString>();
}


int text2vstringVM ( Word * args , Word & result , int message ,
                     Word & local , Supplier s ) {
   FText* arg = (FText*) args[0].addr;
   result = qp->ResultStorage(s);
   VString* res = (VString*) result.addr;
   if(!arg->IsDefined()){
      res->SetDefined(0);
      return 0;
   }
   res->Set(true, arg->GetValue());
   return 0;
}
```

```
OperatorSpec text2vstringSpec (
    "  text -> vstring " ,
    " text2vstring(_) " ,
    " Converts a text into a vstring" ,
    " query text2vstring('This is text') "
);

Operator text2vstringOp (
  "text2vstring" , // name of the operator
  text2vstringSpec.getStr() , // specification
  text2vstringVM , // value mapping
  Operator::SimpleSelect , // selection function
  text2vstringTM // type mapping
);
```

18 Definition of the Algebra

In this step, a new algebra – a class derived from the *Algebra* class – is created. Within the constructor of the algebra, we add the type constructors and assign the corresponding kinds to the types. Furthermore, all operators are added to the algebra.

```
class GuideAlgebra : public Algebra {
   public:
     GuideAlgebra() : Algebra() {

      AddTypeConstructor( &SCircleTC );
      SCircleTC.AssociateKind( Kind::SIMPLE() );

      AddTypeConstructor( &ACircleTC );
      ACircleTC.AssociateKind( Kind::DATA() );

      AddTypeConstructor( &GCircleTC );
      GCircleTC.AssociateKind( Kind::DATA() );

      AddTypeConstructor( &IntListTC );
      IntListTC.AssociateKind( Kind::DATA() );

      AddTypeConstructor( &PAVLTC );
      PAVLTC.AssociateKind( Kind::SIMPLE() );


      AddTypeConstructor( &ushortTC);
      ushortTC.AssociateKind(Kind::DATA());

      AddTypeConstructor( &VStringTC);
      VStringTC.AssociateKind(Kind::DATA());


      AddOperator(&perimeterOp);
      AddOperator(&distNOp);
      AddOperator(&countNumberOp);
      AddOperator(&getCharsOp);
      AddOperator(&startsWithSOp);
      AddOperator(&replaceElemOp);
```

```
      AddOperator(&attrIndexOp);
      AddOperator(&createPAVLOp);
      AddOperator(&containsOp);
      AddOperator(&insertOp);

      AddOperator(&importObjectOp);
      importObjectOp.SetUsesArgsInTypeMapping();

      AddOperator(&importObject2Op);
      importObject2Op.SetUsesArgsInTypeMapping();

      AddOperator(&text2vstringOp);
    }
};
```

End of the namespace. The following code cannot be embedded into the algebras's namespace. Thus the namespace should end here.

```
} // end of namespace guide
```

19 Initialization of the Algebra

This piece of code returns a new instance of the algebra.

```
extern "C"
Algebra*
   InitializeGuideAlgebra( NestedList* nlRef,
                           QueryProcessor* qpRef ) {
   return new guide::GuideAlgebra;
}
```

20 Description of the Specification File

Within the *spec* file, the syntax of operators is fixed. The file must be located in the algebra directory and is named *AlgebraName.spec*. The default syntax of an operator is prefix notation. Nevertheless it is recommended, to specify the syntax for all operators.

If an operator occurs in several algebras, the syntax of the operator must be the same. If not, the compilation process of Secondo (*make* in Secondo's main directory) will fail with an appropriate error message.

Lines in the file starting with a '#' are ignored and can be used for comments. For formatting the file, empty lines are also ignored.

The syntax definition of an operator has the format

```
operator <opname> alias <opalias> pattern <syntax>
         [<implicit parameters>]
```

---

with

- opname : Name of the operator
- opalias: Name of the operator as text
- syntax: syntax of the operator
- implicit parameters: use of special types for function arguments

The *implicit parameter* part is optionally and can be used for function arguments only. The *opalias* is the name of the operator as text, e.g. PLUS if the operator is '+'. This has technical reasons.

The syntax can be:

```
 op()           prefix operator without any argument
 op(_)          prefix operator with one argument
 op(_,_)        prefix operator having arbitrary many arguments
 _ infixop _    infix operator
 _ op           postfix operator with one argument
 _ _ op         postfix operator having two arguments
 ...
 _ op [<paralist>] postfix operator having one argument before the
                   operator and some additional arguments within the
                   square brackets
_ _ op [<paralist>] postfix operator having two arguments before the
                    operator and additional arguments within the
                    square brackets
....
```

The parameter list within the square brackets can have the following values.

```
_, _     usual parameter
list     parameter list of arbitrary length
fun      this parameter is a function (together with implicit parameters)
funlist  list of functions
X ; Y    semicolon separated argument lists, X and Y are parameter lists again
```

Implicit parameters can be used for simpler use of function arguments. This requires a fun or a funlist to be part of the parameter list. The syntax is:

```
implicit parameter <namelist> type <typelist>
```

where *namelist* is a list of comma separated names and *typelist* contains the corresponding types. The first name is assigned to the first type, second name to the second type and so on. This requires that both lists must have the same length. Often, the type is given as so-called Type Operator. This is an operator having only the type mapping but no value mapping. Look at the example:

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

The *filter* operator gets a stream of elements together with a function mapping from the elements type to bool. Only elements fulfilling the condition given by this function pass this operator.

By the definition of implicit parameters, a query using the filter operator can be easely written down. Assume a relation with scheme:

```
zipcodes(Name : string, Zipcode : int)
```

a query for finding all elements of this relation having a zipcode smaller than 2000 can be found using:

```
query zipcodes feed filter[.ZipCode < 2000] consume
```

By the implicit parameter part, this query is rewritten automatically to:

```
query zipcodes feed
      filter[ fun (streamelem : STREAMELEM) attr(e,ZipCode) < 2000 ]
      consume
```

The type operator STREAMELEM extracts the tuple type from the arounding types. Thus, without all these mechanisms the query has to be written as

---

```
query zipcodes feed
      filter[ fun( t : tuple([Name : string, Zipcode : int])) attr(t,ZipCode) < 2000 ]
      consume
```

Obviously, the first version using all features is the simplest one from the view of a user.

The operator syntax definitions of the Guide algebra are:

```
# simple prefix operator having one argument
operator perimeter alias PERIMETER pattern op(_)

# infix operator
operator distN alias DISTN pattern _ infixop _

# postfix operator having a parameter in square brackets
operator countNumber alias COUNTNUMBER pattern _ op[_]

#simple prefix operator
operator getChars alias GETCHARS pattern op(_)

# operator having a function with implicit arguments
operator replaceElem alias REPLACEELEM pattern _ op[ fun ]
         implicit parameter streamelem type STREAMELEM

# postfix operator with an parameter
operator startsWithS alias STARTSWITHS pattern _ op [_ ]

# postfix operator with an additional argument
operator attrIndex alias ATTRINDEX pattern _ op [_]

#postfix operator without parameters
operator createPAVL alias CREATEPAVL pattern _ op
```

```
# infix operator
operator contains alias CONTAINS pattern _ infixop _


# postfix operazor with two arguments
operator insert alias INSERT pattern _ _ op


# prefix operator
operator importObject alias IMPORTOBJECT pattern op(_)

# prefix operator
operator importObject2 alias IMPORTOBJECT2 pattern op(_)

# prefix operator
operator text2vstring alias text2vstring pattern op(_)
```

---

21 The Example File

The example file provides at least one query for each operator. If an operator is omitted in this file, the operator is disabled by the secondo framework. The file is used for automatic tests (Selftest). For a manual start of this test, just enter

---

```
Selftest tmp/<AlgebraName>.examples
```

---

within Secondo's *bin* directory. For memory error or memory leak checking, use the command:

```
Selftest  --valgrind tmp/<AlgebraName>.examples
```

---

or

```
Selftest  --valgrindlc tmp/<AlgebraName>.examples
```

---

for more memory leak information.

The format of the file is quite easy. It starts with some general settings followed by examples for the operators. The general settings are:

```
Sequential: No
Database: berlintest
Restore : No
```

If *Sequential* is set to *Yes*, the examples are executed in the order given in the file. If it is set to *No* or this line is omitted, the operators are tested in lexicographical order using the number as secondary sorting criterion.

The *Database* line specifies the database which is used for testing, here *berlintest*. If the test starts and the database is not already part of the system, the database is restored from a file having the same name as the database in the *bin* directory. If also the file is missing, the Selftest will fail.

If *Restore* is set to *Yes*, the database is freshly restored even if the database already exists in the system. This is required, if within the algebra update operators are defined, manipulating the content of the database. This ensures to start from a well defined database state.

After these general settings, for each operator must be given at least one example. The format is:

```
Operator  : <opname>
Number    : <number>
Signature : <signature>
Example   : <query>
Result    : <result>
Tolerance : <tolerance>
```

---

opname is the name of the operator. For the operator may also given an alias using *alias* <*ALIAS*> directly after the operator's name. This may be required for storing results in a file (see below).

number is a counter starting from 1. If Sequential is set to *Yes*, the counter is increased for each example. If Sequential is *No*, the counting starts by 1 for each new operator and is increased if there are more examples for a single operator.

The signature line describes the signature of the operator in a textual way. If for each signature supported by the operator an example is given, this line can be restricted to the signature used in the query. If only one example is given for an entire overloaded operator, the description must handle all possible signatures in a general way.

The *query* part is a query using the operator, constant values and/or objects defined within the used database.

result specifies the result of the example query. If the result has a simple type (int, real, bool, string), it can be given in short form, e.g. 1, TRUE, or "secondo". Otherwise, the result is specified as nested list including the type, e.g. (int 1), (bool TRUE), (string "secondo"). Sometimes, the result depends on the underlying operating system. For example, the number of nodes of an r-tree depends on the page size of the system. To give a result for different operating systems, the result is specified in format:

```
(platform (os1 result1) (os2 result2) ...)
```

---

where os_i is one out of {linux, linux64, win32, mac_osx}.

If the result is too large for writing it directly, the keyword *file* can be used instead of the result. In this case, the result must be stored in a file called
result<*number*>_<*opname*>_<*AlgebraName*> or
result<*number*>_<*opname*>_<*AlgebraName*>_<*operating_system*>
in case of operating system depending results. If the operator's name is not allowed to appear within a file name, the alias of the operator is used within the file name (see above).

The *Tolerance* is an optional value which is used to compare the results. If *Tolerance* is specified, the result list and the computed result of a query are compared using an approximative comparison for double values. The tolerance can be given as an absolute value, e.g. 0.01 if a deviance between the two values should not be greater than 0.01, or as an relative value, e.g. 0.1% is the accepted tolerance is 0.1 percent.

The *GuideAlgebra* contains the following examples:

```
Database: berlintest
Restore: No

Operator : perimeter
Number   : 1
Signature   : scircle -> real
Example  : query perimeter([const scircle value (1 2 3 )])
Result   : 18.85
Tolerance : 0.02

Operator : distN
Number   : 1
Signature   : int x int -> int
Example  : query 1 distN 3
Result   : 2
```

```
Operator : distN
Number   : 2
Signature  : real x real -> real
Example  : query 3.0 distN 1.0
Result   : 2.0

Operator : countNumber
Number   : 1
Signature : stream(int) x int -> int
Example   : query intstream(1,10) countNumber[4]
Result    : 1

Operator : getChars
Number   : 1
Signature: string -> stream(string)
Example  : query getChars("Secondo") count
Result   : 7

Operator  : startsWithS
Number    : 1
Signature : stream(string) x string -> stream(string)
Example   : query intstream(1,1000) replaceElem[ num2string(.) ]  startsWithS["22"] count
Result    : 11


Operator  : replaceElem
Number    : 1
Signature : stream(X) x (X -> Y) -> stream(Y), X,Y in kin DATA
Example   : query intstream(1,3) replaceElem[ fun(i : int) i * 2.0] transformstream sum[Elem]
Result    : 12.0


Operator  : attrIndex
Number    : 1
Signature : stream(tuple(X)) x symbol -> int
Example   : query plz feed attrIndex[Ort]
Result    : 1

Operator  : createPAVL
Number    : 1
Signature : stream(int) -> pavl
Example   : query intstream(1,3) createPAVL
Result    : (pavl (FALSE (2 (1 () ()) (3 () ()))) )


Operator  : contains
Number    : 1
Signature : pavl x int -> bool
Example   : query intstream(1,3) createPAVL contains 4
Result    : FALSE
```

```
Operator  : insert
Number    : 1
Signature : stream(int ) x pavl -> int
Example   : query intstream(1,100) (intstream(50, 150) createPAVL) insert
Result    : 49


Operator  : importObject
Number    : 1
Signature : text -> X
Example   : query importObject('../Data/Objects/Germany/Staedteobj') count
Result    : 59

Operator  : importObject2
Number    : 1
Signature : text -> X
Example   : query importObject2('../Data/Objects/Germany/' + 'Staedteobj') count
Result    : 59


Operator  : text2vstring
Number    : 1
Signature : text -> vstring
Example   : query text2vstring('secondo')
Result    : (vstring "secondo")

Operator  : text2vstring
Number    : 2
Signature : text -> vstring
Example   : query text2vstring('secondo secondo secondo secondo secondo')
Result    : (vstring "secondo secondo secondo secondo secondo")
```

---

22 Advanced Tests

The Selftest provides only simple tests. Only queries are allowed within an *examples* file, hence only objects within the stored database or constant values can be used. Secondo provides a more powerful test suite for testing an algebra modul, the *TestRunner*. A TestRunner file consists of three parts: the *setup* part, followed by the *test* part and the final *teardown* part. Within the setup part, a database can be created, restored or just opened, new objects can be stored and so on. If one of the commands in the setup part fails, the complete test fails. In the testcase part, commands and their results are specified. Besides a concrete result, the result can also indicate the success or failure of a command. Wheras the success is required for let commands

and similar, the failure case can be used to check the system for wrong inputs, e.g. checking type mappings.

An example TestRunner file including descriptions of the file format is the *example.test* file located in Secondo's *bin* directory.

For using a *TestRunner* file, enter:

```
TestRunner -i <filename>
```

within Secondo's *bin* directory. As usual, if valgrind is part of your system, the following commands can be used for finding memory errors and leaks:

```
TestRunner --valgrind -i <filename>
TestRunner --valgrindlc -i <filename>
```