Secondo Programmer's Guide

Version 2, October 5, 2004

Ralf Hartmut Güting, Victor Teixeira de Almeida, Dirk Ansorge,

Thomas Behr, Markus Spiekermann

Praktische Informatik IV, Fernuniversität Hagen

D-58084 Hagen, Germany

Table of Contents

1 Algebra Module Implementation

In SECONDO, data types and operations are provided by algebra modules. Such modules are initialized in SECONDO by the algebra manager which makes the data types and operations of algebras available in the query processor. Figure 2 in Section 1 of the *Secondo User Manual* gives a good impression of SECONDO's structure and where algebras are linked into the system. Theoretically, the number of algebras linked into SECONDO is unlimited. However, the performance of the system may be slightly slowed down if a very high number of algebras is registered.

Naturally, an algebra implementor first has to write the C++-Code providing the data types and operations. Any such algebra may be linked into SECONDO if some additional functions are implemented, e.g. the `In`- and `Out`-functions, which convert the "internal" data type representation to the nested list representation used in SECONDO. When the implementation is done, the algebra has to be registered in some places in the system to work properly.

In the following sections, two examples are given about how algebras are implemented and embedded in SECONDO. First, the simple `PointRectangleAlgebra` is described providing two new data types and two operations. Second, an algebra for handling streams of `int` values is presented. For both examples it is explained how the compiled code is linked together with other SECONDO modules.

1.1 Example 1: PointRectangleAlgebra

1.1.1 Writing a Simple Algebra for SECONDO

The `PointRectangleAlgebra` is a simple algebra which was implemented just for the purpose of having a small example algebra for SECONDO. The implementation of this class is part of the standard SECONDO distribution and is located in the directory `Algebras`. All algebras must be located in this directory. However, not all algebras, which are present here, have to be inevitably embedded in the system. The file `PointRectangleAlgebra.cpp` is viewed best by using `PDView`, another tool on the SECONDO-CD. This way the included documentation can be read formatted nicely. In the following, important and interesting parts of the code are shown and explained.

Every new algebra must be a subclass of the class `Algebra`. In `Algebra.cpp` the definitions for the construction of operators, type constructors and algebras can be found. It is strongly recommended to have a closer look to this example algebra before starting to implement an own algebra. The algebra implementation is discussed as implemented, from the beginning to the end.

Preliminaries

First, we need some `include` directives:

```
#include "Algebra.h"
#include "NestedList.h"
#include "QueryProcessor.h"
#include "StandardTypes.h"
```

`Algebra.h` is included, since the new algebra must be a subclass of it. All of the data available in SECONDO has a nested list represention. Therefore, conversion functions have to be written for this algebra, too, and `NestedList.h` is needed for this purpose. The result of an operation is passed directly to the query processor. An instance of `QueryProcessor` serves for this. SECONDO provides some standard data types such as `CcBool`, which is needed as the result type of the implemented operations. So `StandardTypes.h` is included.

References to instances of the `NestedList` and the `QueryProcessor` classes are needed later but are already defined here.

```
static NestedList* nl;
static QueryProcessor* qp;
```

Data Structure: Class XPoint

Now, the new data type `XPoint` is defined, followed by the implementation of its operations.

```
class XPoint
{
 public:
  XPoint( int x, int y );
  ~XPoint();
  int     GetX();
  int     GetY();
  void    SetX( int x );
  void    SetY( int y );
  XPoint* Clone();
 private:
  int x;
  int y;
};

//implementation of the operations
//...
```

Nested List Representation/Conversion

As presented above, an `XPoint` value has two coordinates, represented by `int` values. The nested list representation can be chosen freely, but for standard geometric data types as well as for the purpose of reuseability in other algebras it should be a reasonable representation like

```
(x y)
```

The required functions for the conversion of the internal representation (i.e. instance of class `XPoint`) to the nested list representation and vice versa are called `In`- and `Out` functions. Some kind of “naming convention” is used here; therefore these functions are called `InXPoint` and `OutXPoint` for class `XPoint`.

```
static ListExpr
OutXPoint( ListExpr typeInfo, Word value )
{
  XPoint* point;
  point = (XPoint*)(value.addr);
  return nl->TwoElemList(nl->IntAtom(point->GetX()),
          nl->IntAtom(point->GetY()));
}

static Word
InXPoint( const ListExpr typeInfo, const ListExpr instance,
       const int errorPos, ListExpr& errorInfo, bool& correct )
{
  XPoint* newpoint;

  if ( nl->ListLength( instance ) == 2 )
  {
    ListExpr First = nl->First(instance);
    ListExpr Second = nl->Second(instance);

    if ( nl->IsAtom(First) && nl->AtomType(First) == IntType
      && nl->IsAtom(Second) && nl->AtomType(Second) == IntType )
    {
      correct = true;
      newpoint = new XPoint(nl->IntValue(First), nl->IntValue(Second));
      return SetWord(newpoint);
    }
  }
  correct = false;
  return SetWord(Address(0));
}
```

Nested lists are used in `OutXPoint` as follows: A new list with two elements is constructed with `Two-ElemList`, which is a function of class `NestedList`. As arguments a nested list containing two `int` values is passed and inserted into the list. An element of a nested list may be either a nested list (this is why they are called “nested” lists) or an atom. In this case atom types are needed. They are constructed using `IntAtom`, again a function of class `NestedList`. The `int` values are derived from the `XPoint` value with usage of `GetX`- and `GetY`-functions of class `XPoint`. The use of `NestedList.cpp` is straight-forward. Just have a look at `NestedList.h` in the `include` directory.

`InXPoint` is probably the more interesting function, since here a new `XPoint` instance is created. Both `int` values are extracted from the nested list and passed to the `XPoint` type constructor. Finally, the newly created `XPoint` instance is returned. Note, that in the `In`-function it is necessary to check carefully and completely whether the passed list has the correct structure. Such lists can be written by users directly (using a text editor) and may have errors.

Description of the Signature of the Type Constructors

At the user-interface the command `list type constructors` lists all type constructors of all currently linked algebra modules. The information listed is generated by the algebra module itself, to be more precise it is generated by the `Property`-functions:

```
static ListExpr
XPointProperty()
{
  return (nl->TwoElemList(
              nl->FiveElemList(
                  nl->StringAtom("Signature"),
                  nl->StringAtom("Example Type List"),
                  nl->StringAtom("List Rep"),
                  nl->StringAtom("Example List"),
                  nl->StringAtom("Remarks")),
              nl->FiveElemList(
                  nl->StringAtom("-> DATA"),
                  nl->StringAtom("xpoint"),
                  nl->StringAtom("(<x> <y>)"),
                  nl->StringAtom("(-3 15)"),
                  nl->StringAtom("x- and y-coordinates must be "
                         "of type int."))));
}
```

Here, only explanatory textual data has to be entered. Note that the maximum length of a `String` is 48.

Persistent Storage and Related Generic Functions

A SECONDO object belongs to a database and needs to be stored persistently. Hence it needs a representation on disk, in files or records of the underlying storage manager.

To be used in query processing, an object must be *opened*. That is, in addition to the disk representation some main memory representation must be created that makes it possible to access the value. Hence an object has two states:

- *closed*: only the disk representation exists
- *opened*: disk and memory representations exist.

There are six generic functions for each type constructor that allow one to create and delete a value (or SECONDO object) of the type, open and close it, save it and clone it. These operations return a value in a particular state and are also applicable to a value in some state. The state diagram in Figure 1 shows how operations manipulate object states.

The operations have the following meaning:

- `create`: create a value in open state (disk and memory part)
- `delete`: delete the value, i.e., remove disk part and memory part
- `open`: given a disk representation, create the memory part

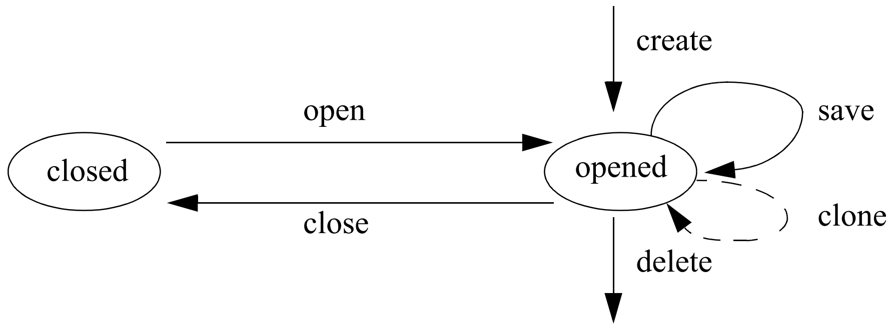

Figure 1: Generic functions and object states

- `close`: release the memory part
- `save`: propagate changes from the memory part to the disk part
- `clone`: make a copy of the object (disk part and memory part)

To make life easier for the implementor of a simple data type like `int` or `xpoint`, there exists a default mechanism for persistent storage. Here the text form of the nested list representation for the type is stored in a record on disk. Hence to store an object, its `Out` function is called; the resulting list is converted to text form and written to the record. To open an object, the text representation is read from the record and converted to a nested list for which then the `In` function is called to create the value.

This means that for a type constructor the functions `create`, `delete`, `close`, and `clone` *must* be implemented. The remaining two functions open and save *may* be implemented. If they are not implemented, the default persistent storage mechanism is used.

To make things yet a bit more complex, there is a variant of the default mechanism. Observe that the `In` function needs to perform a complete check of the argument list to see whether it has a correct structure and whether the value described by it is correct. For example, for the ***region*** data type in the spatial algebra expensive tests need to be made. To avoid this, it is possible to define an alternative nested list structure just to be used internally for storage. In this case, the data structure (e.g. for a ***region***) can be directly transformed into an appropriate list structure describing the elements of this data structure, and it is easy to recreate the data structure from this list representation.

To use an alternative list structure, the implementor of a data type needs to provide two functions called `SaveToList` and `RestoreFromList`.

Hence for persistent storage, there are three alternatives:

- If `open` and `save` are implemented, a specific persistent implementation is used.
- Otherwise, if `SaveToList` and `RestoreFromList` are implemented, then the default mechanism is used but with a specific list structure.
- Otherwise, the default mechanism is used with the external list representation, using `In` and `Out` functions.

In general, the `open` and `save` functions are not implemented for simple objects; instead, the default storage mechanism is used. Therefore, they are not implemented here.

The functions themselves are self-explanatory:

```
static Word
CreateXPoint( const ListExpr typeInfo )
{
  return (SetWord( new XPoint( 0, 0 ) ));
}

static void
DeleteXPoint( Word& w )
{
  delete (XPoint *)w.addr;
  w.addr = 0;
}

static void
CloseXPoint( Word& w )
{
  delete (XPoint *)w.addr;
  w.addr = 0;
}

static Word
CloneXPoint( const Word& w )
{
  return SetWord( ((XPoint *)w.addr)->Clone() );
}
```

The `SizeOf`-function must be implemented to provide information about the object's size.

```
int
SizeOfXPoint()
{
  return sizeof(XPoint);
}
```

Kind Checking

When entering a type expression into a user interface, the query processor first does a kind checking. This means that the structure of the argument passed to each type constructor is checked. If it is not correct, the input is not accepted. Actually, this checking is done by so-called kind checking functions implemented in the algebra. At this point we have to specify the kind checking function for the `XPoint` constructor. Since it has no arguments, this is trivial.

```
static bool
CheckXPoint( ListExpr type, ListExpr& errorInfo )
{
  return (nl->IsEqual( type, "xpoint" ));
}
```

In contrast to this example, writing kind checking functions may get much more difficult for other types such as relations in the `RelationAlgebra`.

Defining Type Constructors

Now, that all required functions are defined, the `TypeConstructor` object for the new data type `XPoint` can be defined. This is not much more than passing a list of previously defined functions to the constructor of class `TypeConstructor`. Some additional values, which are not explained here, are for future extensions of the SECONDO system. A zero is passed for all functions that are not implemented, e.g. `open` and `save`.

```
TypeConstructor xpoint(
  "xpoint",                      //name
  XPointProperty,                //property function describing signature
  OutXPoint, InXPoint,           //Out and In functions
  0, 0,                          //SaveToList and RestoreFromList functions
  CreateXPoint, DeleteXPoint,      //object creation and deletion
  0, 0, CloseXPoint, CloneXPoint, //object open, save, close, and clone
  DummyCast,                       //cast function
  SizeOfXPoint,                    //sizeof function
  CheckXPoint,                     //kind checking function
  0,                               //predef. pers. function for model
  TypeConstructor::DummyInModel,
  TypeConstructor::DummyOutModel,
  TypeConstructor::DummyValueToModel,
  TypeConstructor::DummyValueListToModel );
```

Data Structure: Class Rectangle

Up to now the implementation of functions for the data type `XPoint` was presented. Similar code must be written for the second data type `XRectangle` (including `In`- and `Out`-functions, signature description, transition functions etc.). This code is omitted here.

Creating Operators

Now that both data types are available, the type mapping functions are defined. A type mapping function checks, whether its argument types are correct. If not, the input is not accepted. The result type is a list expression for the result type of the symbol `typeerror` (if the arguments are not correct). Again, the code should be self-explanatory.

```
static ListExpr
RectRectBool( ListExpr args )
{
  ListExpr arg1, arg2;
  if ( nl->ListLength(args) == 2 )
  {
    arg1 = nl->First(args);
    arg2 = nl->Second(args);
   if ( nl->IsEqual(arg1, "xrectangle") && nl->IsEqual(arg2, "xrectangle"))
    return nl->SymbolAtom("bool");
  }
  return nl->SymbolAtom("typeerror");
}

static ListExpr
XPointRectBool( ListExpr args )
{
  ListExpr arg1, arg2;
  if ( nl->ListLength(args) == 2 )
  {
    arg1 = nl->First(args);
    arg2 = nl->Second(args);
    if ( nl->IsEqual(arg1, "xpoint") && nl->IsEqual(arg2, "xrectangle") )
    return nl->SymbolAtom("bool");
  }
  return nl->SymbolAtom("typeerror");
}
```

Selection Functions

Functions may be overloaded. Selection functions are used to select one of several evaluation functions for an overloaded operator, based on the types of the arguments. In this example we don't have overloaded operators and therefore just have to return 0.

```
static int
simpleSelect (ListExpr args ) { return 0; }
```

An example from the `StandardAlgebraC++` shows how the correct operations are selected if overloaded operators exist. For each overloaded operator, an array containing the function pointers of several implementations is defined. Note, that the functions in these arrays are value mapping functions, which are desribed in the following subsection.

```
ValueMapping ccplusmap[] = { CcPlus_ii, CcPlus_ir, CcPlus_ri, CcPlus_rr };
ValueMapping ccminusmap[] = { CcMinus_ii, CcMinus_ir, CcMinus_ri,
                                CcMinus_rr };
ValueMapping ccproductmap[] = { CcProduct_ii, CcProduct_ir, CcProduct_ri,
CcProduct_rr };
ValueMapping ccdivisionmap[] = { CcDivision_ii, CcDivision_ir,
CcDivision_ri, CcDivision_rr };
```

Here, `CcPlus_ii` is the +-operator for two `int` values etc. A selection function returns the correct index for the `ccplusmap`.

```
static int
CcMathSelectCompute( ListExpr args )
{
  ListExpr arg1 = nl->First( args );
  ListExpr arg2 = nl->Second( args );
  if ( TypeOfSymbol( arg1 ) == ccint && TypeOfSymbol( arg2 ) == ccint )
    return (0);
  if ( TypeOfSymbol( arg1 ) == ccint && TypeOfSymbol( arg2 ) == ccreal )
    return (1);
  if ( TypeOfSymbol( arg1 ) == ccreal && TypeOfSymbol( arg2 ) == ccint )
    return (2);
  if ( TypeOfSymbol( arg1 ) == ccreal && TypeOfSymbol( arg2 ) == ccreal )
    return (3);
  return (-1); // This point should never be reached
}
```

In `StandardAlgebraC++` the selection function `CcMathSelectCompute` is used for arithmetic operations +, -, *, /.

Value Mapping Functions

For any operation available in the algebra a value mapping function must be defined. Inside of these functions the internal functions (e.g. `intersects`) are applied to the passed arguments. The resulting value then is written directly to an object provided by the query processor. In this example algebra we have two operations:

```
static int
intersectsFun (Word* args, Word& result, int message, Word& local, Supplier
s)
{
  XRectangle* r1;
  XRectangle* r2;
  r1 = ((XRectangle*)args[0].addr);
  r2 = ((XRectangle*)args[1].addr);

  result = qp->ResultStorage(s);    //query processor has provided
                                    //a CcBool instance to take the result
  ((CcBool*)result.addr)->Set(true, r1->intersects(*r2));
                                    //the first argument says the boolean
                                    //value is defined, the second is the
                                    //CcBool value)
  return 0;
}

static int
insideFun (Word* args, Word& result, int message, Word& local, Supplier s)
{
  XPoint* p;
  XRectangle* r;
  p = ((XPoint*)args[0].addr);
  r = ((XRectangle*)args[1].addr);

  result = qp->ResultStorage(s);    //query processor has provided
                                    //a CcBool instance to take the result
```

```
    bool res = ( p->GetX() >= r->GetXLeft() && p->GetX() <= r->GetXRight()
    && p->GetY() >= r->GetYBottom() && p->GetY() <= r->GetYTop() );

    ((CcBool*)result.addr)->Set(true, res);
                                      //the first argument says the boolean
                                      //value is defined, the second is the
                                      //CcBool value)
    return 0;
  }
```

Definition of Operators

Similar to the description of the definition of the type constructors, the operators are described. The signature and the meaning of the operators are explained here.

```
const string intersectsSpec  =
                      "( ( \"Signature\" \"Syntax\" \"Meaning\" "
                      "\"Example\" ) "
                      "( <text>(xrectangle xrectangle) -> bool</text--->"
                      "<text>_ intersects _</text--->"
                      "<text>Intersection predicate for two"
                      " xrectangles.</text--->"
                      "<text>r1 intersects r2</text--->"
                      ") )";
const string insideSpec  =
                      "( ( \"Signature\" \"Syntax\" \"Meaning\" "
                      "\"Example\" ) "
                      "( <text>(xpoint xrectangle) -> bool</text--->"
                      "<text>_ inside _</text--->"
                      "<text>Inside predicate.</text--->"
                      "<text>p inside r</text--->"
                      ") )";
```

The description is given as a character string containing a nested list. In the examples, there are many strings which are in C++ all concatenated into a single one. The list has the form

```
( (<heading 1> ... <heading k>) (<entry 1> ... <entry k>) )
```

Headings are string atoms and entries are text atoms. Quotes for the string atom have to be escaped within another string, hence one needs to write `\"Signature\"` , for example. Standard headings for operator descriptions are Signature, Syntax, Meaning, and Example. This is followed by four text atoms including the description itself.

Finally, instances of the class `Operator` are constructed for each operator.

```
Operator intersects (
  "intersects",         //name
  intersectsSpec,       //specification
  intersectsFun,        //value mapping
  Operator::DummyModel, //dummy model mapping, defined in Algebra.h
  simpleSelect,         //trivial selection function
  RectRectBool          //type mapping
);
```

```
Operator inside (
  "inside",             //name
  insideSpec,           //specification
  insideFun,            //value mapping
  Operator::DummyModel, //dummy model mapping, defined in Algebra.h
  simpleSelect,         //trivial selection function
  XPointRectBool        //type mapping
);
```

In case of an overloaded operator (as in `StandardAlgebraC++`) a different constructor for instances of an `Operator` is used where the value mapping array is also passed as an argument (see `Algebra.h`).

Creating the Algebra

The algebra itself is basically created by declaring a class derived from class `Algebra` and calling add-functions for all type constructors and operators in the constructor of this class. By calling these functions the constructors and operators are registered in the algebra manager.

```
class PointRectangleAlgebra : public Algebra
{
 public:
  PointRectangleAlgebra() : Algebra()
  {
    AddTypeConstructor( &xpoint );
    AddTypeConstructor( &xrectangle );

    xpoint.AssociateKind("SIMPLE"); //this means that xpoint and xrectangle
    xrectangle.AssociateKind("SIMPLE"); //can be used in places where types
                                        //of kind SIMPLE are expected
    AddOperator( &intersects );
    AddOperator( &inside );
  }
  ~PointRectangleAlgebra() {};
};
```

The last line defines the instance of the algebra.

```
PointRectangleAlgebra pointRectangleAlgebra;
```

Initialization

Finally, the algebra must be equipped with an initialization function. The algebra manager has a reference to this function if this algebra is included in the list of required algebras, thus forcing the linker to include this module.

The algebra manager invokes this function to get a reference to the instance of the algebra class and to provide references to the global nested list container (used to store constructor, type, operator and object information) and to the query processor.

The function has a C interface to make it possible to load the algebra dynamically at runtime.

```
extern "C"
Algebra*
InitializePointRectangleAlgebra( NestedList* nlRef, QueryProcessor* qpRef )
{
  nl = nlRef;
  qp = qpRef;
  return (&pointRectangleAlgebra);
}
```

1.1.2 Linking the PointRectangleAlgebra to SECONDO

To link the algebra to the SECONDO system it has to be registered in the algebra manager. Additionally, it has to be entered in one configuration file and `makefile`.

The first thing to do is to make a new directory in the `Algebras` directory of the SECONDO system. The new directory name should be the same as the algebra name omitting the term "Algebra" (`PointRectangle` in this case). Afterwards the algebra file has to be copied to the new directory. Now, two additional files have to be created:

1. The `.spec`-file, which provides important information for the parser about the algebra's operators.
2. The `make` file for the algebra directory including information about which files have to be compiled during the compilation process of the system.

The `.spec` file for the `PointRectangleAlgebra` looks like this:

```
operator intersects alias INTERSECTS pattern _ infixop _
operator inside alias INSIDE pattern _ infixop _
```

As you can see, both available operators are listed. The structure of such a specification is:

```
operator <name> alias <ALIAS> pattern <pattern>
```

The `ALIAS` is the name of the token for the operator needed for the lexical analysis. As a naming convention the original name is kept but written in capitals. Then, the `pattern` for the operator is given. The symbol _ denotes the places of arguments and `infixop` shows the position of the operator. Therefore, the operator `intersects` would be used as `a intersects b` in a query. Other examples for operator specifications can be found in the complete `spec`-file which is the concatenation of all algebra `.spec`-files. This file is located in the `Algebras` directory.

To create the `makefile` the easiest way is to copy a `makefile` from another algebra module and adapt it to the new algebra. The only thing to do here is to change the `MODNAME` at the right place (which is easy to find).

```
...
MODNAME=PointRectangleAlgebra
LIBNAME=lib$(MODNAME)
...
```

Next, the file `makefile.algebras` has to be changed. This file contains two entries for every algebra. The first defines the directory name and the second the name of the algebra module like in the example below :

```
...
ALGEBRA_DIRS += PointRectangle
ALGEBRAS     += PointRectangleAlgebra

ALGEBRA_DIRS += BTree
ALGEBRAS     += BTreeAlgebra
...
```

The last step is to register the algebra in the algebra manager. To do this it has to be added to the list of algebras in `AlgebraList.i` in the `Algebras/Management` directory. Here, the algebra must be entered together with its name, a unique algebra number (just choose an unused number) and a level on which the algebra works (which is `executable` for the `PointRectangleAlgebra`).

```
...
ALGEBRA_INCLUDE(1,StandardAlgebra,Hybrid)
ALGEBRA_INCLUDE(2,FunctionAlgebra,Executable)
ALGEBRA_INCLUDE(3,RelationAlgebra,Executable)
ALGEBRA_INCLUDE(4,PointRectangleAlgebra,Executable)
ALGEBRA_EXCLUDE(5,StreamExampleAlgebra,Executable)
...
```

Now, all configuration is done for the `PointRectangleAlgebra`. Execute the `make` file in the SECONDO home directory to link the new algebra. After the process has finished, start SECONDO and type `list algebra PointRectangleAlgebra` to see the new operators and type constructors.

Note: The easiest way to make your algebra available is to change `AlgebraList.i.cfg` and `makefile.algebras` and then invoke `make alg=auto`.

1.2 Example 2: StreamExampleAlgebra

1.2.1 Writing an Algebra Using Streams

The `StreamExampleAlgebra` is a small example demonstrating streams of objects. In this case operators for the construction of streams of `int` values are provided together with some operators which have such streams as arguments. This example helps to understand much more sophisticated algebras using streams such as the `RelationAlgebra`.

The structure of this algebra is pretty much the same as the structure of the previously described `PointRectangleAlgebra` (and any other SECONDO algebra). Obviously, the same set of additional functions is needed for the implementation of the new `StreamExampleAlgebra`. But, in contrast to the `PointRectangleAlgebra` this algebra doesn't have any constructors, since a new stream is constructed using an operator, not a constructor. The algebra provides the following operators:

- `int × int → stream(int)` **intstream**
  Creates a stream of integers containing all integers from the first up to the second argument. If the second argument is smaller than the first, the stream will be empty.
- `stream(int) → int` **count**
  Returns the number of elements in an integer stream.
- `stream(int) → stream(int)` **printintstream**
  Prints out all elements of the stream. Returns the argument stream unchanged.
- `stream(int) × (int → bool) → stream(int)` **filter**
  Filters the elements of an integer stream by a predicate.

Again, we strongly recommend to have a close look at the implementation of the algebra. Here, we only describe the new concepts and the exciting parts of the new algebra.

The algebra has no constructors, hence no function concerning constructors (e.g. constructor implementation, kind checking functions...) has to be written.

Type Mapping Functions

The first thing to do is to implement the type mapping functions for the four operators. As an example for this the mapping for **intstream** will serve:

```
static ListExpr
intstreamType( ListExpr args ){
ListExpr arg1, arg2;
if ( nl->ListLength(args) == 2 )
  {
    arg1 = nl->First(args);
    arg2 = nl->Second(args);
    if ( nl->IsEqual(arg1, "int") && nl->IsEqual(arg2, "int") )
      return nl->TwoElemList(nl->SymbolAtom("stream"),
                     nl->SymbolAtom("int"));
  }
  return nl->SymbolAtom("typeerror");
}
```

Note that a stream is treated like an atom in the nested list representation. Nothing more of this piece of code should be new. The type mapping for other operators is quite similar.

Selection Functions

No overloaded operators are provided by the algebra. Hence, a `simpleSelect` function is implemented equal to the one in the `PointRectangleAlgebra`.

Value Mapping Functions

To be able to understand how the value mapping functions work we have to take a closer look at stream operators in general. Stream operators manipulate streams of objects. They may consume one or more input streams, produce an output stream or both. For a given stream operator α, let us call the operator receiving its output stream its successor and the operators from which it receives its input streams it predecessors. Now, stream operators work as follows: Operator α is sent a `request` message from its successor to receive a stream object. Operator α in turn sends a `request` to its predecessors. The predecessors either deliver the object (sending a `yield` message) or don't have objects any more (sending a `cancel` message). Figure 2 shows the protocol dealing with streams.

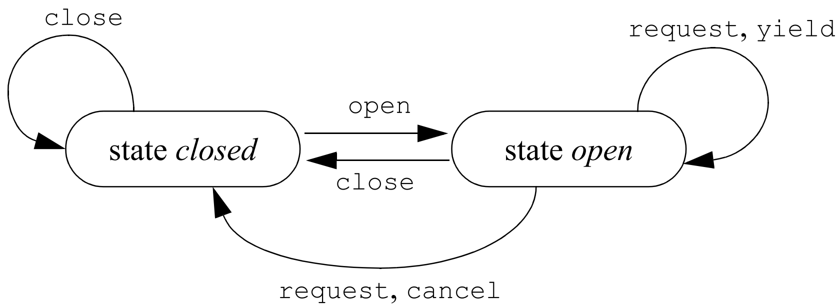

Figure 2: State Diagram for Streams

The first message sent to an operator producing a stream must be `open`. This converts the stream from the state *closed* to the state *open*. Afterwards, either `request` or `close` messages can be sent to the stream. Sending `request` means, that the successor would like to receive an object. The response can be a `yield` message (giving the object to the successor, remaining in state *open*) or a `cancel` message (no further objects are available, switching to state *closed*). The successor may send a `close` message, which means that it does not wish to receive any further objects even though they may be available. This also transforms the stream into state *closed*.

When trying to simulate stream operators by algebra functions, it can be observed that a function need not relinquish control (terminate) when it sends a message to a predecessor. This can be treated pretty much like calling a parameter function. However, the function needs to terminate when it sends a `yield` or `cancel` message to the successor. This makes it necessary to write the function in such a way that it has some local memory and that each time when it is called, it just delivers one object to the successor.

In the following example for a value mapping function for `intstream` the possible messages are encoded by constants:

```
CONST OPEN = 1; REQUEST = 2; CLOSE = 3; YIELD = 4; CANCEL = 5;
```

The messages are passed as parameter and also used as return type. The parameter `local` is used to store local variables that must be maintained between calls to the stream. More information about stream operators can be found in [GFB+97].

```
static int
intstreamFun
  (Word* args, Word& result, int message, Word& local, Supplier s)
{
```

```
  struct Range {int current, last;}* range;

  CcInt* i1;
  CcInt* i2;
  CcInt* elem;

  Word arg0, arg1;

  switch( message )
  {
    case OPEN:

      qp->Request(args[0].addr, arg0);
      qp->Request(args[1].addr, arg1);

      i1 = ((CcInt*)arg0.addr);
      i2 = ((CcInt*)arg1.addr);

      range = new Range;
      range->current = i1->GetIntval();
      range->last =  i2->GetIntval();

      local.addr = range;

      return 0;

    case REQUEST:

      range = ((Range*) local.addr);

      if ( range->current <= range->last )
      {
        elem = new CcInt(true, range->current++);
        result.addr = elem;
        return YIELD;
      }
      else return CANCEL;

    case CLOSE:

      range = ((Range*) local.addr);
      delete range;
      return 0;
  }
  /* should not happen */
  return -1;
}
```

As we can see, three different messages, namely `OPEN`, `REQUEST` and `CLOSE`, are handled in the function. Note that for any operator that produces a stream its arguments are not evaluated automatically. To get the argument value, the value mapping function needs to use `qp->Request` to ask the query processor for evaluation explicitly. This is done in the section for `OPEN`. After that a value for `range` is set and stored in the local variable. In the `REQUEST` section `range` is read and a new `int` value is computed if the current value for `range` is smaller than the last stream value. In this case, `YIELD` is returned. Otherwise the result is `CANCEL`. Finally, in the `CLOSE` section, the stream is closed.

The value mapping function for `count` shows how a stream is consumed:

```
while ( qp->Received(args[0].addr) )
  {
    count++;
    delete((CcInt*) elem.addr);              //consume the stream objects
    qp->Request(args[0].addr, elem);
  }
```

In `filterFun` a parameter function is used to let through only selected stream objects:

```
Word elem, funresult;
ArgVectorPointer funargs;

...

case REQUEST:

  funargs = qp->Argument(args[1].addr);        //Get the argument vector for
                                               //the parameter function.
  qp->Request(args[0].addr, elem);
  while ( qp->Received(args[0].addr) )
  {
    (*funargs)[0] = elem;                      //Supply the argument for the
                                               //parameter function.
    qp->Request(args[1].addr, funresult);      //Ask the parameter function
                                               //to be evaluated.
    if ( ((CcBool*) funresult.addr)->GetBoolval() )
      {
        result = elem;
        return YIELD;
      }
    qp->Request(args[0].addr, elem);
  }
  return CANCEL;
```

Definition of Operators

In this part of the algebra implementation, some constants are defined which are used to explain the signature and the meaning of the operators. Note, that this is only a textual description. If changing the implementation, don’t forget to change the description. This is a common source of errors!

Furthermore, now that all specifications definitions are done, the operators are defined here.

Creating the Algebra

The algebra is created using the following, very small piece of code:

```
class StreamExampleAlgebra : public Algebra
{
 public:
  StreamExampleAlgebra() : Algebra()
  {
    AddOperator( &intstream );
    AddOperator( &cppcount );
    AddOperator( &printintstream );
    AddOperator( &filter );
  }
  ~StreamExampleAlgebra() {};
};
```

In contrast to the `PointRectangleAlgebra`, the `Add`-commands for constructors are missing, since no constructors are provided.

Initialization

The algebra is initialized in the same way as the `PointRectangleAlgebra`.

1.2.2 Linking the StreamExampleAlgebra to SECONDO

Linking new algebras to the SECONDO system is always the same. Nothing different than in the example before is done. The following points are on the check-list:

- create a new algebra directory and copy your code to this directory
- write `.spec` and `make` for the algebra (place them in algebra directory)
- adapt `makefile.algebras` in algebra directory
- register algebra in manager by adding it in file `Algebra/Management/AlgebraList.i`

2 Extending the Relational Algebra

At this point, you should be familiar with the SECONDO Relational Algebra as a user, which means that you know how to create and query relations.

You should also know about creating algebras in detail, i.e., creating type constructors, operators, etc. because more sophisticated concepts about type constructors and operators will be handled in this section.

It is also important that you understand the concepts in the Stream Algebra, which are used in almost all operators in the Relational Algebra.

2.1 The Relational Algebra Implementation

The Relational Algebra provides two type constructors: *rel* and *tuple*. The structural part of the relational model can be described by the following signature:

kinds IDENT, DATA, TUPLE, REL

type constructors

	→	DATA	*int*, *real*, *string*, *bool*
$(\text{IDENT} \times \text{DATA})^+$	→	TUPLE	*tuple*
TUPLE	→	REL	*rel*

Therefore a tuple is a list of one or more pairs <identifier, attribute type>. A relation is built from such a tuple type.

There are two types of the Relational Algebra that can co-exist in the SECONDO System, namely Persistent Relational Algebra (PRA) and Main Memory Relational Algebra (MMRA). In the PRA, relations are files (`SmiRecordFile` class) containing variable length records (`SmiRecord` class), each record storing a tuple. For more detail on files and records, see Section 5. In the MMRA, relations are handled in memory, and a relation has a compact table (see class `CTable`) of tuples. The whole relation is read into memory in the In-function, and a list representation is used to read and write a relation into disk. Since this process can be very slow, and the same relation can be used lots of times, they are cached and not destroyed in the `Close` type constructor function. They are only destroyed if there is no space available in the cache.

A define “RELALG_PERSISTENT” (see Algebras/Relation-C++/makefile) switches between the two representations. If it is defined, then the PRA is used, otherwise the MMRA is used.

Another important structure that is used by the Relational Algebra is the Database Array (`DBArray` class). DBArrays are used to implement complex attribute types, such as *region* for example. DBArrays also have different implementations within the PRA and the MMRA. While in the MMRA, it is simply a memory array, in the PRA it has a complex structure.

To explain its structure, we need first to explain the concept of FLOBs, which stands for Faked Large OBjects. There is a tradeoff between storing large objects in tuples together with the other attributes and in a separate record. Not always does a large object use a large amount of storage. As an example, imagine that the *region* type constructor has a large object to store its array of segments. Imagine then that we have in the system a relation that stores rectangles as regions. Rectangles need to store only four segments and then, the storage needed for rectangles is not large. Therefore, it is preferable to store the rectangles together with the other attributes of the relation. To solve this problem, the concept of a FLOB was created. It is an abstraction of a large object (LOB) that decides where to store the objects. If the size of the large object is smaller than a specified threshold, then the “large” object is stored together with the other attributes of the tuple, and otherwise it is stored in a separate record.

This concept makes sense only for the PRA. In the MMRA the FLOBs are simply a sequence of bytes of variable size. In the PRA, the FLOBs are stored together (in the same record) with the tuples if their sizes are small and in a separate file for large objects (LOBs). Therefore, a relation in the PRA contains two files: one containing records for the tuples and another containing records for the LOBs.

Database arrays are constructed on top of FLOBs. The difference between FLOBs and DBArrays is that a FLOB is a sequence of bytes (in disk or in memory) without structure, and a DBArray is a structured array implemented as a C++ template. For more detailed information about FLOBs and DBArrays, see Section 3.

The most important operators in the relational algebra are described below:

- **feed**: produces a stream of tuples from a relation.
- **consume**: the contrary of **feed**, i.e., produces a relation from a stream of tuples.
- **rename**: changes only the type, not the value of a stream by appending the characters supplied as argument to each attribute name.
- **filter**: receives a stream of tuples and passes along only tuples for which the parameter function evaluates to true.
- **attr**: retrieves an attribute value from a tuple. The dot “.” notation is used inside some operators, like **filter** for example.
- **project**: implements the relational projection operation on streams.
- **product**: implements the relational cartesian product operation on streams.
- **count**: counts the number of tuples in a stream or in a relation.

It is important to note that most of the operators of the relational algebra run on streams of tuples instead of directly on relations. Exceptions are the **feed** operator, that produces the streams, and the **count** operator, that is allowed to count the number of tuples also directly on relations.

2.2 Value Mapping Functions

We will, in this section, take a closer look into the **feed** and **consume** operators' value mapping functions. The **feed** operator's value mapping function is a good example of the stream algebra concepts, whereas with the **consume** operator we can show an important concept about free and non-free tuples.

Feed Operator Value Mapping Function

Let us now take a closer look on the value mapping function of the **feed** operator, for a review of the stream algebra concepts.

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


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

      local = SetWord(rit);
      return 0;

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

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

When the message is an `OPEN`, the **feed** operator requests from the query processor the argument relation, initializes the iterator `rit`, and stores this iterator in a special variable called `local`. This `local` variable is used to keep the state of the operator, i.e., it is a way to simulate the storage of local variables as static. Then, in the `REQUEST` message, the **feed** operator gets the iterator back from the `local` variable, retrieves the next tuple from it, and puts it into the `result` variable. If there are no more tuples available in the iterator, then `CANCEL` is returned, otherwise `YIELD` is returned. This `result` variable together with the return of the function (`YIELD` or `CANCEL`) will be used by the operator which sent the `REQUEST` message to the **feed** operator. Finally, in the `CLOSE` message, the iterator is closed and 0 is returned, which means success.

Consume Operator Value Mapping Function

Let us now take a closer look into the **consume** operator's value mapping function:

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

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

  qp->Open(args[0].addr);
  qp->Request(args[0].addr, actual);
  while (qp->Received(args[0].addr))
  {
    Tuple* tuple = ((Tuple*)actual.addr)->CloneIfNecessary();
    rel->AppendTuple(tuple);
    if( tuple != actual.addr )
      ((Tuple*)actual.addr)->DeleteIfAllowed();
    tuple->Delete();

    qp->Request(args[0].addr, actual);
  }

  result = SetWord((void*) rel);

  qp->Close(args[0].addr);

  return 0;
}
```

As mentioned before, the **consume** operator receives tuples from a stream and builds a relation containing these tuples. Actually the relation creation is not a task of the **consume** operator, but of the query processor. The query processor previously creates storage at the query tree construction time for the type constructor's result type returned in the type mapping function. The function `ResultStorage` of the query processor returns a pointer (as a `Word`) to this created space.

There are special cases where the **consume** operator can be called inside a loop - in the **loopjoinrel** operator for example - and the storage is created only once by the query processor. Therefore, it is necessary to empty the relation retrieved from the query processor before proceeding.

The function then sends the stream messages OPEN and REQUEST, to retrieve the first tuple from the stream argument. If (and while) the tuple is received, it gets the tuple from the actual variable written by the REQUEST message. The **consume** operator clones the tuple, appends it to the result relation, and asks for the next one from the stream argument. After retrieving all tuples from the stream argument and appending them to the result relation, it closes the stream, and returns 0, meaning success.

We did not mention intentionally the tuple's Delete and DeleteIfAllowed functions in the explanation of the **consume** operator's value mapping function. These are functions that belong to an important concept of the MMRA that will be discussed now in more detail. In queries, in general, there is a specified sequence that is followed: the relation is transformed into a stream by the **feed** operator; the tuples coming from streams are passed along to one or more operators like **rename**, **filter**, and **product**; and then the resulting stream of tuples is transformed back into a relation by the **consume** operator, or the tuples are counted using the **count** operator. In order to illustrate the idea (and the need) of *free* and *non-free* tuples, we will use two relations, namely employee and dept, and an example query:

```
create employee: rel(tuple([ename: string, empnr: int, deptnr: int]));
update employee :=
  [const rel(tuple([ename: string, empnr: int, deptnr: int]))
   value (("Smith" 12 3)("Myers" 13 2)("Bush" 11 1)
          ("Jones" 14 2)("Smith" 16 1)("Langdon" 9 3)
          ("Lambert" 4 3)("Callahan" 1 2)("Myers" 17 2)
          ("Simpson" 8 3))];

create dept: rel(tuple([leader: string, deptnr: int]));

update dept :=
  [const rel(tuple([leader: string, deptnr: int]))
   value (("Smith" 3)("Myers" 2)("Bush" 1))];
```

and the query is:

```
query
  employee feed {a}
  dept feed
  product
  filter[.deptnr_a = .deptnr]
  project[ename_a, empnr_a, deptnr_a, leader]
  consume
```

As a comment, the usage of "{a}" is a shorthand for the operation "rename[a]".

The tuples are passed from one operator to another as pointers (C++ pointers). Initially, all tuples belong to relations, but in the middle of a query, new tuples can be created and, to avoid memory leaks, some tuples must be deleted. Using this example, the **product** operator creates new tuples containg all attributes from the tuples of the relation employee (after being renamed) and the relation dept and pass them to the **filter** operator. The **project** operator retrieves these tuples (only those that satisfy the filter property) and creates different ones with only some of the attributes. At this moment, the tuples created by the **product** operator should be deleted, because they will be no longer needed.[1] We can conclude by this example that the **project** operator should always delete the tuples that it receives. This is not true! Imagine the query below:

```
query employee feed project[ename] consume
```

If we, in the **project** operator, delete the tuples that it receives, we will delete the tuples that are still referenced by the `employee` relation.

To correctly delete and create tuples, we have created the concept of free and non-free tuples, implemented as a flag inside the `Tuple` class called `free`. Free tuples are ready to be deleted, and non-free tuples are the contrary. Tuples pointed to by relations are always non-free tuples, in a way that they are never deleted by any operator. The tuples created by the internal operators are always free tuples. The implementation is complemented by the following functions in the `Tuple` class:

- `IsFree`: checks if the tuple is free or not
- `SetFree`: sets the free flag
- `DeleteIfAllowed`: calls the delete of the tuple if it is a free tuple and does nothing otherwise.
- `CloneIfNecessary`: the tuple is cloned only if it is not free. If it is a free tuple, a simple pointer to it is returned. The clone function always creates non-free tuples.

One should see in the **project** operator value mapping function that it contains a `DeleteIfAllowed` call to delete tuples only if it is allowed, i.e., if they are free. In this way, it will delete the tuples from the first query example, created in the **product** operator, but it will not delete the tuples from the second query example, coming from the `employee` relation.

Going back to the **consume** operator value mapping function explanation, every tuple received from the stream argument is cloned (using the function `CloneIfNecessary`) which creates a non-free tuple. After that, the tuple can be deleted if allowed (using the function `DeleteIfAllowed`).

For the PRA, this concept of free and non-free tuples is somehow different, because the tuples are persistent and do not need to be kept in memory in some cases. For that, there is still one line of the code not commented yet, which calls the function `Delete`. This code is necessary for the PRA. In the MMRA, after appending a tuple to a relation, nothing else must be done, but in the PRA, the tuple must be deleted from memory, since it was written into disk. For that, the `Delete` function is used, which deletes tuples for the PRA and does nothing for the MMRA.

2.3 Type Mapping Functions

Up to now, in the algebra implementation tasks, the type mapping functions of all operators were very simple. A type mapping function takes a nested list as an argument. Its contents are type descriptions of an operator's input parameters. A nested list describing the output type of the operator is returned. The feed operator's type mapping function, for example, can be described as

1. In fact, also the tuples that did not pass the **filter** operator should be deleted.

```
((rel x)) -> (stream x)
```

which means that it receives a relation of tuples as an argument and returns the tuples as a stream.

In the relational algebra we have some type mapping functions that are quite complex and use the special keyword `APPEND` in the result type. We will show the need and the effects of the `APPEND` keyword using the **attr** and **project** type mapping functions as examples.

Attr Type Mapping Function

The **attr** operator takes a tuple and retrieves an attribute value from it. The type mapping function should be:

```
((tuple ((x1 t1)...(xn tn))) xi)    -> ti)
```

where `xi` is an attribute name, and `ti` is its data type, both indexed by `i`. The type mapping function of the **attr** operator should be

```
ListExpr AttrTypeMap(ListExpr args)
{
  ListExpr first, second, attrtype;
  string  attrname;
  int j;
  if(nl->ListLength(args) == 2)
  {
    first = nl->First(args);
    second  = nl->Second(args);

    if((nl->ListLength(first) == 2  ) &&
        (TypeOfRelAlgSymbol(nl->First(first)) == tuple)  &&
        (nl->IsAtom(second)) &&
        (nl->AtomType(second) == SymbolType))
    {
      attrname = nl->SymbolValue(second);
      j = FindAttribute(nl->Second(first), attrname, attrtype, nl);
      if (j)
        return attrtype;
    }
  }
  ErrorReporter::ReportError("Incorrect input for operator attr.");
  return nl->SymbolAtom("typeerror");
}
```

where the variable `first` contains the list `(tuple ((x1 t1)...(xn tn)))`and the `second` contains the attribute name `xi` that we are interested in retrieving. The function `FindAttribute` receives a list of pairs of the form `((x1 t1)...(xn tn))`, an attribute name and a data type that will be filled as a result. It then determines, whether the attribute name occurs as one of the attributes in this list. If so, the index in the list (beginning from 1) is returned and the corresponding datatype is put in the attribute data type argument. Otherwise 0 is returned.

Consider now the value mapping function for the **attr** operator. It can get a tuple and an attribute name as arguments. However, to access the attribute value it needs the numeric index of the attribute.

But the value mapping function does not know about the tuple type, and therefore it would be impossible to determine the attribute index. Moreover, it would be inefficient to compute the index once for every tuple processed in a stream. Since we calculated the index in the type mapping function, it would be nice if another argument could be added and used in the value mapping function. This will be done using the `APPEND` keyword. The technique used is that the type mapping function returns not just a result type, but a list of the form

```
(APPEND (<newarg1> ... <newargn>) <resulttype>)
```

`APPEND` tells the query processor to add the elements of the following list `(<newarg1> ... <newargn>)` to the argument list of the operator as if they had been written in the query. Using this approach, we can pass the attribute index as another argument to the value mapping function of the **attr** operator as it is shown below.

```
((tuple ((x1 t1)...(xn tn))) xi)    -> (APPEND (i) ti)
```

This resulting type mapping function will pass the tuple `t`, the attribute name `xi` and the attribute index `i` to the value mapping function as arguments, and set the attribute type `ti` to be the result type of the operator. The type mapping function could be rewritten like this:

```
((tuple ((x1 t1)...(xn tn))) xi i)    -> ti)
```

The *real*[1] **attr** operator type mapping function is then:

```
ListExpr AttrTypeMap(ListExpr args)
{
  ListExpr first, second, attrtype;
  string  attrname;
  int j;
  if(nl->ListLength(args) == 2)
  {
    first = nl->First(args);
    second  = nl->Second(args);
    if((nl->ListLength(first) == 2  ) &&
        (TypeOfRelAlgSymbol(nl->First(first)) == tuple)  &&
        (nl->IsAtom(second)) &&
        (nl->AtomType(second) == SymbolType))
    {
      attrname = nl->SymbolValue(second);
      j = findattr(nl->Second(first), attrname, attrtype, nl);
      if (j)
        return
          nl->ThreeElemList(nl->SymbolAtom("APPEND"),
                            nl->OneElemList(nl->IntAtom(j)), attrtype);
    }
  }
  ErrorReporter::ReportError("Incorrect input for operator attr.");
  return nl->SymbolAtom("typeerror");
}
```

The main difference is that instead of returning only the attribute type, the function now returns a three element list containing first the keyword `APPEND`, then a list containing the attribute index, and

1. This is still not the real type mapping function, which contains more detailed type checking and error reporting.

finally the attribute type, which will be the result type of the **attr** operator. The value mapping function is then:

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

  tupleptr = (Tuple*)args[0].addr;
  index = ((CcInt*)args[2].addr)->GetIntval();
  result = SetWord(tupleptr->GetAttribute(index - 1));
  return 0;
}
```

The value mapping function takes the arguments from the vector `args`. The first argument in position 0 is the tuple, the second is the attribute name which is not used, and the third in position 2 is the attribute index passed inside the `APPEND` command. The function then returns the attribute value at this position pointed to by the attribute index.

Project Type Mapping Function

Following the same idea presented above for the **attr** operator, the **project** operator also uses the `APPEND` command. The **project** operator's type mapping function acts like the description below.

```
((stream (tuple ((x1 T1) ... (xn Tn)))) (ai1 ... aik))->
    (APPEND
      (k (i1 ... ik))
      (stream (tuple ((ai1 Ti1) ... (aik Tik)))))
```

which means that it receives a stream of tuples with tuple description `((x1 T1) ... (xn Tn))` and a list of attribute names `(ai1 ... aik)`. The type mapping function will return not only the result type, which is a stream of tuples containing only the attributes in the argument set `(ai1 ... aik)`, i.e., a stream of tuples with description `((ai1 Ti1) ... (aik Tik))`. It uses the `APPEND` command to append a list of attribute indexes `(i1 ... ik)`, and the number of attributes `k` contained in the set. The **project** operator's type mapping function is shown below[1]:

```
ListExpr ProjectTypeMap(ListExpr args)
{
  bool firstcall;
  int noAttrs, j;
  ListExpr first, second, first2, attrtype, newAttrList, lastNewAttrList,
           lastNumberList, numberList, outlist;
  string attrname;

  firstcall = true;
  if (nl->ListLength(args) == 2)
  {
    first = nl->First(args);
    second = nl->Second(args);
```

1. As for the **attr** operator, the more complicated type checking and error reporting are omitted.

```
if ((nl->ListLength(first) == 2) &&
    (TypeOfRelAlgSymbol(nl->First(first)) == stream) &&
    (nl->ListLength(nl->Second(first)) == 2) &&
    (TypeOfRelAlgSymbol(nl->First(nl->Second(first))) == tuple) &&
    (!nl->IsAtom(second)) && (nl->ListLength(second) > 0))
{
  noAttrs = nl->ListLength(second);

  while (!(nl->IsEmpty(second)))
  {
    first2 = nl->First(second);
    second = nl->Rest(second);

    if (nl->AtomType(first2) == SymbolType)
      attrname = nl->SymbolValue(first2);
    else
    {
      ErrorReporter::ReportError("Incorrect input for project.");
      return nl->SymbolAtom("typeerror");
    }
    j = findattr(nl->Second(nl->Second(first)), attrname, attrtype, nl);
    if (j)
    {
      if (firstcall)
      {
        firstcall = false;
        newAttrList =
          nl->OneElemList(nl->TwoElemList(first2, attrtype));
        lastNewAttrList = newAttrList;
        numberList = nl->OneElemList(nl->IntAtom(j));
        lastNumberList = numberList;
      }
      else
      {
        lastNewAttrList =
          nl->Append(lastNewAttrList,
                     nl->TwoElemList(first2, attrtype));
        lastNumberList =
          nl->Append(lastNumberList, nl->IntAtom(j));
      }
    }
    else
    {
      ErrorReporter::
        ReportError("Incorrect input for operator project.");
      return nl->SymbolAtom("typeerror");
    }
  }
```

```
      // Check whether all new attribute names are distinct
      // - not yet implemented
      outlist =
        nl->ThreeElemList(
          nl->SymbolAtom("APPEND"),
          nl->TwoElemList(nl->IntAtom(noAttrs), numberList),
          nl->TwoElemList(nl->SymbolAtom("stream"),
                          nl->TwoElemList(nl->SymbolAtom("tuple"),
                                          newAttrList)));
      return outlist;
    }
  }
  ErrorReporter::ReportError("Incorrect input for operator project.");
  return nl->SymbolAtom("typeerror");
}
```

This function starts setting the `first` and `second` variables with the lists of the tuple representation and the set of attribute names desired in the projection, respectively. The number of resulting attributes is taken from the length of the attribute names list, and a variable `first2` is used to iterate inside the set of attribute names. The `findattr` function is used again to retrieve the attribute index given an attribute name and a list of these attributes indexes called `numberList` is constructed. A list containing the pairs <attribute name, attribute type> is also constructed using the `newAttrList` variable. The return of the **project** operator's type mapping function is a list of three elements, the first containing the `APPEND` command, the second containing the information that will be appended as arguments passed to the value mapping function, i.e., the set of attributes indexes and the size of this set, and the third containing the result type of the **project** operator, which is a stream of tuples containing the pairs in the `newAttrList` variable.

3 DBArray - An Abstraction to Manage Data of Widely Varying Size

3.1 Overview

The `DBArray` class provides an abstract mechanism supporting instances of data types of varying size. It implements an array, whose slot size is fixed, however, the number of slots may grow dynamically. `DBArray` is implemented as a template class and provides the following interface:

Creation/Removal	**Access**	**Other**
`DBArray`	`Append`	`Size`
`~DBArray`	`Get`	`Resize`
`Destroy`	`Put`	`Clear`
		`Sort`

The class is derived from the class `FLOB` with the purpose of hiding the complexity of managing large objects. The `FLOB` (Faked Large Object) decides whether an object has to be loaded or stored on disk and how it is distributed over different record files. `FLOBs` were invented to improve the storage process of tuples potentially containing large objects into records of a storage management system. Based on a threshold value, small objects are stored within tuple records, whereas large objects are swapped out into separate records; for details refer to [DG98]. Hence, the design of the `DBArray`, `FLOB` and the tuple representation within the relational algebra was coordinated to support a simple integration of new data types into the relational data model.

3.2 Example

As an example for the usage of class DBArray, the implementation of the Polygon Algebra [Poly02] can be studied. It uses a class `Polygon` having a private member of type `DBArray`:

```
class Polygon {
  ...
  void Append( Vertex& v );
  ...
  private:
    int noVertices;
    int maxVertices;
    DBArray<Vertex> vertices;
    PolygonState state;
  }
```

```
void Polygon::Append( Vertex& v )
{
  assert( state == partial );
  vertices->Put( noVertices++, v );
  if( noVertices > maxVertices )
    maxVertices = noVertices;
}
```

These are only some code fragments, but they demonstrate the usage of `DBArray`. Since `DBArray` is a template class it is type safe and instantiated with parameter type `Vertex`, a simple struct representing (x, y) coordinates. The assertion `state == partial` is just a helpful instrument to locate errors during the development, a polygon may be in state `partial` or `completed`.

The algebra's `In`-function will create a new class `Polygon` and iterate over the nested list, which is passed to it by the update command. During the iteration process the `Append` function is used to fill the `DBArray` with vertices. This is demonstrated in the example below.

```
Word
InPolygon( const ListExpr typeInfo, const ListExpr instance,
           const int errorPos, ListExpr& errorInfo, bool& correct )
{
  Polygon* polygon;

  polygon = new Polygon( 0 );

  ListExpr first;
  ListExpr rest = instance;
  while( !nl->IsEmpty( rest ) )
  {
    first = nl->First( rest );
    rest = nl->Rest( rest );

    if( nl->ListLength( first ) == 2 &&
        nl->IsAtom( nl->First( first ) ) &&
          nl->AtomType( nl->First( first ) ) == IntType &&
        nl->IsAtom( nl->Second( first ) ) &&
          nl->AtomType( nl->Second( first ) ) == IntType )
    {
      Vertex v( nl->IntValue( nl->First( first ) ),
        nl->IntValue( nl->Second( first ) ) );
      polygon->Append( v );
    }
    else
    {
      correct = false;
      return SetWord( Address(0) );
    }
  }
  polygon->Complete();
  correct = true;
  return SetWord( polygon );
}
```

The function Clear deletes the current content of a FLOB, but the instance can be reused. Usage of Destroy completely removes all underlying records. The latter is typically done in a data type's delete function.

```
void DeletePolygon(Word& w)
{
  Polygon* polygon = (Polygon*)w.addr;

  polygon->Destroy();
  delete polygon;
}
```

Here polygon->Destroy calls vertices->Destroy.

3.3 Acessing FLOBs Directly

The DBArray class provides a nice and clean abstraction mechanism to support arrays of varying size, containing elements of fixed size. But, if the data type is not well organized in arrays, one can use directly an object of class FLOB. This would be important for a type constructor which needs to store long and variable sized texts, for example. Texts can be viewed as an array of characters, but it would be better to store them directly into a FLOB to avoid lots of calls to Put and Get functions. The FLOB class provides an interface similar to the DBArray class, but the Put and Get functions are different. Their interface is shown below:

```
void Put( int offset, int length, const void *source );
void Get( int offset, int length, void *target );
```

They read (write) into source (target), from an offset until length in bytes. Let us see an example of the BinaryFileAlgebra, which provides the type constructor ***binfile***. This algebra stores in a SECONDO object the bytes sequence of a file. The storage will be provided by a member variable of type FLOB in the BinaryFile class sketched below:

```
#include "Base64.h"

class BinaryFile : public StandardAttribute
{
  public:
  ...

    void Encode( string& textBytes );
    void Decode( string& textBytes );
    bool SaveToFile( char *fileName );

  private:

    FLOB binData;
    bool canDelete;
};
```

Since SECONDO needs the ability of textual representation of data, binary data must be encoded in printable characters. Base 64 [BF93] is a widely used encoding format for this purpose. The Encode and `Decode` functions convert binary to textual representation and vice versa. The examples below demonstrate access to a `FLOB` objects, i.e., how to use the `Put` and `Get` functions.

```
void BinaryFile::Encode( string& textBytes )
{
  Base64 b;
  char *bytes = (char *)malloc( binData.Size() );
  binData.Get( 0, binData.Size(), bytes );
  b.encode( bytes, binData.Size(), textBytes );
  free( bytes );
}
```

Both functions use the `Base64` class that provides functions for encoding binary data and decoding Base 64 data. The `Encode` function first allocates some bytes for reading and then calls the function `Get` to read in the bytes from the `FLOB`. These bytes are in binary format and they are passed to the `encode` function of the `Base64` class creating a string with characters of the Base 64 alphabet.

```
void BinaryFile::Decode( string& textBytes )
{
  Base64 b;
  int sizeDecoded = b.sizeDecoded( textBytes.size() );
  char *bytes = (char *)malloc( sizeDecoded );

  int result = b.decode( textBytes, bytes );

  assert( result <= sizeDecoded );

  binData.Resize( result );
  binData.Put( 0, result, bytes );
  free( bytes );
}
```

The `Decode` function does the contrary, it decodes a Base64 text string into a block of binary bytes and stores it in the FLOB using the function `Put`. Note that before copying data into the FLOB its capacity must be adjusted with the `Resize` function.

3.4 Interaction with the Relational Algebra

The requirements for data types to be used as relation attributes are discussed in detail in the next section, but when DBArrays or FLOBs are used, two functions inherited from class `TupleElement` must be implemented. An example of these functions for the *binfile* type constructor is shown below.

```
int BinaryFile::NumOfFLOBs()
{
  return 1;
}

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

As data types can have more than one FLOB, it is necessary to implement a mechanism to have access to them. In this way, the function `NumOfFLOBs` must return how many FLOBs the data type has, and the function `GetFLOB` must return the one identified by the requested index `i`.

4 Making Data Types Available as Attribute Types for Relations

When data types shall be used as attributes of a relation object, every C++ class implementing such a type has to implement a special set of functions, which are used by operators of the relational algebra. These functions add some useful properties like comparability or the ability to compute hash values and are declared in the following hierachy of classes:

`TupleElement` → `Attribute` → `StandardAttribute` → `IndexableStandardAttribute`.

To get this additional functionality, a set of functions inherited from class `StandardAttribute` must be implemented.

Moreover, to support building indexes on complex attributes, functions inherited from `Indexable-StandardAttribute` must be implemented in order to guarantee interoperability with the BTree Algebra [BTree02]. Those data types also need to belong to the kind `INDEXABLE`.

The functions of each class are explained in more detail in the tables below:

Class `TupleElement`	
Function	**Description**
`NumOfFLOBs`	Returns the number of `FLOBs` (`DBArrays`) used in the class. Default implementation returns zero.
`GetFLOB`	Returns a pointer to a `FLOB` object.
`Open`	May be used as standard method for opening a persistent object. Can be useful in the implementation of type constructors.
`Save`	The complementary function to `Open`. Makes an object persistent.
`Initialize`	Not used!
`Print`	Can be implemented to print out useful information for debugging.
`operator<<`	Can be implemented to write the class into an `ostream` object.

The `open` and `save` functions should not be overwritten. They define useful standard implementations for creating or restoring persistent versions of an object with `FLOB` members, hiding the complexity of reading and writing data from memory into records. If the data type does not contain members of type `FLOB,` no functions of this class need to be implemented.

Class `Attribute`	
Function	**Description**
`Compare`	Defines a total order. This is, for example, used by the **sortby** operator of the relational algebra.
`Adjacent`	Decides whether the current instance (`this`) is adjacent to an instance of the same type passed as argument. Examples: 2 and 3 are adjacent integers; “abc“ and “abd“ are adjacent strings. This is used by `range` algebra operations.
`Clone`	This is needed by several operators of the relational algebra.
`IsDefined`	This function is used to indicate if the object represents a valid value or not, for example, a division by zero results into an integer object with status ’not defined’.
`SetDefined`	Used to set the objects ’defined’ status.

Class `StandardAttribute`	
Function	**Description**
`HashValue`	This is a function which maps values to integers used by hash-functions. The **hashjoin** operator of the relational algebra needs this information.
`CopyFrom`	Works like a copy constructor. It copies the value of a referenced attribute into this object. This is used by several operators of the relational algebra.

Class `IndexableStandardAttribute`	
Function	**Description**
`WriteTo`	Converts the value of an object into an uniqe (order preserving) key value.
`SizeOfChars`	Length of the key value.
`ReadFrom`	Restores an object value from its key.

Additionally, a function returning the object’s size is needed. Usually, an implementation uses the preprocessor macro `sizeof(<ClassName>)`. It is used by the tuple manager, a module which can bring tuples from disk into memory representation and vice versa, and has to be defined in the data type’s `TypeConstructor` object (see Section 1).

Note: The data structure of any data type intended to be suitable as an attribute type is strictly required to have the following structure: It consists of a single block of memory which may contain a fixed number of FLOBs or DBArrays. Hence all attributes of such a class must themselves have fixed size (so the compiler embeds their memory blocks into the block of the class instance). In other words, the only types that can be used as class members except for FLOBs and DBArrays are scalar types like `int`, `float`, `char[]`, etc. It is not possible to use pointer structures or classes from libraries, e.g. a binary tree or a string, as type of member variables. If dynamic data structures like trees are needed they have to embedded into DBArrays, with array indices serving as pointers.

The reason for this restriction is that the mechanism for building tuple data structures with the automatic placement of FLOBs is based on this assumption. In addition, all "pointers" are automatically stable regardless of how values are placed in memory.

At first glance, an algebra implementor may not understand the need for all of these functions. However, the relational algebra makes use of all of them. So, the implementation of these functions is essential.

Refer to the `DateTimeAlgebra` [Date04] as an example for an algebra whose types are made available as attribute types in relations. The example below describes some parts of the C++ class `DateTime`.

```
class DateTime : public IndexableStandardAttribute  {

...

/*

The next functions are needed for the DateTime class to act as
an attribute of a relation.

*/
     int Compare(Attribute* arg);
     bool Adjacent(Attribute*);
     int Sizeof();
     bool IsDefined() const;
     void SetDefined( bool defined );
     size_t HashValue();
     void CopyFrom(StandardAttribute* arg);
     DateTime* Clone();
     void WriteTo( char *dest ) const;
     void ReadFrom( const char *src );
     SmiSize SizeOfChars() const;

...
}
```

This class uses no FLOBs, hence no functions of class `TupleElement` are implemented. The `PolygonAlgebra` [Poly02] already mentioned in Section 3.2 does have FLOBs. In the implementation of its `open` and `save` function one can see how the standard `open` and `save` methods of class `TupleElem` are reused.

```
bool
OpenPolygon( SmiRecord& valueRecord,
             const ListExpr typeInfo,
             Word& value )
{
  Polygon *p = new Polygon( 0 );
  p->Open( valueRecord, typeInfo );
  value = SetWord( p );
  return true;
}

bool
SavePolygon( SmiRecord& valueRecord,
             const ListExpr typeInfo,
             Word& value )
{
  Polygon *p = (Polygon *)value.addr;
  p->Save( valueRecord, typeInfo );
  return true;
}
```

5 SMI - The Storage Management Interface

The SMI is a general interface for reading and writing data to disk. Although this sounds simple, implementing concepts for locking, buffering, and transaction management is a highly complex task. Therefore, SECONDO does not have its own concepts for this purpose but uses already existing database libraries. Hence, the SMI provides a collection of classes which are used as a general interface within SECONDO to access the API of other database libraries. Currently, code for two implementations of the SMI is present; the first one is based on the Open Source project Berkeley-DB and the second one uses the Oracle-DBMS. The latest SECONDO sources can only be compiled with the Berkeley-DB version of the SMI, since only this version is maintained in the further development. However, the interface was designed to separate the code of the SECONDO core system and the algebra modules from interfaces of libraries which support storage management facilities.

5.1 Retrieving and Updating Records

In this introduction an overview about the SMI classes and their interdependencies is presented. The SMI uses the two concepts *records* and *files*. In a record a sequence of bytes can be stored. It has a unique ID and each file can hold many records. Basically, the SMI offers operations on files and records. An overview about all SMI classes is presented below:

Classes	**Description**
`SmiEnvironment`	This class provides static member functions for the startup and initialization of the storage management environment.
`SmiFile`	Base class, a container for records.
`SmiRecordFile`	Records are selected by their IDs.
`SmiKeyedFile`	Records are accessed by a key value.
`SmiFileIterator`	Base class, supports scanning of files.
`SmiRecordFileIterator`	Iterator for files with access via record-IDs.
`SmiKeyedFileIterator`	Iterator for files with access via keys.
`SmiRecord`	A handle for processing records. It can be used to access all or partial data of records.
`SmiKey`	A generalization for different types of keys, such as integer, string, etc.
`PrefetchingIterator`	Efficient read-only iteration reducing I/O activity.

More technical information about functions and their signatures is described in the file `SecondoSMI.h`. The following code examples demonstrate the usage of the `SmiRecordFile` and `SmiRecord` classes.

```
bool makefixed = true;
string filename = (makeFixed) ? "testfile_fix" : "testfile_var";
SmiSize reclen = (makeFixed) ? 20 : 0;
SmiRecordFile rf( makeFixed, reclen );
if ( rf.Open( filename ) )
{
  cout << "RecordFile successfully created/opened: "
      << rf.GetFileId() << endl;
  cout << "RecordFile name   =" << rf.GetName() << endl;
  cout << "RecordFile context=" << rf.GetContext() << endl;
  cout << "(Returncodes: 1 = ok, 0 = error )" << endl;
  SmiRecord r;
  SmiRecordId rid, rid1, rid2;
  rf.AppendRecord( rid1, r );
  r.Write( "Emilio", 7 );
  rf.AppendRecord( rid2, r );
  r.Write( "Juan", 5 );
  char buffer[30];
  rf.SelectRecord( rid1, r, SmiFile::Update );
  r.Read( buffer, 20 );
  cout << "buffer = " << buffer << endl;
  cout << "Write " << r.Write( " Carlos ", 8, 4 );
  cout << "Read " << r.Read( buffer, 20 ) << endl;
  cout << "buffer = " << buffer << endl;
  r.Truncate( 3 );
  int len = r.Read( buffer, 20 );
  cout << "Read " << len << endl;
  buffer[len] = '\0';
  cout << "buffer = " << buffer << endl;
}
```

A `RecordFile` object can either contain records of fixed length or of variable length. This is controlled by a boolean parameter passed to the constructor. Note that a new record first has to be appended to the `RecordFile`; afterwards a value can be assigned using the `write` operation. The `write` and `read` operations on records may have up to three parameters. The first parameter defines a storage buffer for the data which is transfered into memory or to the record on disk. The second parameter holds the number of bytes to transfer. Finally, the (optional) third parameter defines an offset relative to the starting position of the record on disk.

5.2 The SMI Environment

At the startup process of SECONDO, an instance of class `SmiEnvironment` is created. Then if, during the work with SECONDO, a database is opened, `SmiFile` objects can be used. Whenever new objects in a SECONDO database are created or destroyed using algebra operations, information about the involved `SmiFiles` and the objects types and values has to be maintained in the database catalog. Hence the catalog does many SMI-Operations. Moreover, the query processor opens, closes, creates and deletes objects.

Most data types have quite simple representations and can be stored in files using default persistence mechanisms, but more complex data types, e.g. relations, have to organize data in their own files and thus use SMI operations.

In order to get familiar with the SMI it is recommended to create small test programs independent from the main SECONDO system. A framework for the startup and shutdown of the storage manager is shown below:

```
SmiError rc;
bool ok;

string configFile = "SecondoConfig.ini"
rc = SmiEnvironment::StartUp( SmiEnvironment::MultiUser,
                              configFile, cerr );
cout << "StartUp rc=" << rc << endl;
if ( rc == 1 )
{
    string dbname="test";
    ok = SmiEnvironment::CreateDatabase( dbname );
    if ( ok ) {
      cout << "CreateDatabase ok." << endl;
    } else {
      cout << "CreateDatabase failed." << endl;
    }
    if ( ok = SmiEnvironment::OpenDatabase( dbname ) ) {
      cout << "OpenDatabase ok." << endl;
    } else {
      cout << "OpenDatabase failed." << endl;
    }
  }
  if ( ok )
  {
    cout << "Begin Transaction: "
   << SmiEnvironment::BeginTransaction() << endl;

    /* SMI code */

    cout << "Commit: "
   << SmiEnvironment::CommitTransaction() << endl;

    if ( SmiEnvironment::CloseDatabase() ) {
      cout << "CloseDatabase ok." << endl;
    } else {
      cout << "CloseDatabase failed." << endl;
    }
    if ( SmiEnvironment::EraseDatabase( dbname ) ) {
      cout << "EraseDatabase ok." << endl;
    } else {
      cout << "EraseDatabase failed." << endl;
    }
  }
}
rc = SmiEnvironment::ShutDown();
cout << "ShutDown rc=" << rc << endl;
```

For building an executable program which offers a runtime environment for working with `SmiFiles` this code has to be linked together with the SECONDO SMI library and Berkeley-DB library. Please refer to the file `./Tests/makefile` which contains rules for linking against these libraries.

6 Extending the Optimizer

In this section we describe how simple extensions to the optimizer can be done. We first give an overview of how the optimizer works in Section 6.1. We then explain in Section 6.2 how various extensions of the underlying SECONDO system lead to extensions of the optimizer, and how these can be programmed.

The optimizer is written in PROLOG. For programming extensions, some basic knowledge of PROLOG is required, but also sufficient.

6.1 How the Optimizer Works

6.1.1 Overview

The current version of the optimizer is capable of handling *conjunctive queries*, formulated in a relational environment. That is, it takes a set of relations together with a set of selection or join predicates over these relations and produces a query plan that can be executed by (the current relational system implemented in) SECONDO.

The selection of the query plan is based on cost estimates which in turn are based on given selectivities of predicates. Selectivities of predicates are maintained in a table (a set of PROLOG facts). If the selectivity of a predicate is not available from that table, then an interaction with the SECONDO system takes place to determine the selectivity. More specifically, the selectivity is determined by sending a selection or join query on small *samples* of the involved relations to SECONDO which returns the cardinality of the result.

The optimizer also implements a simple SQL-like language for entering queries. The notation is pretty much like SQL except that the lists occurring (lists of attributes, relations, predicates) are written in PROLOG notation. Also note that the where-clause is a list of predicates rather than an arbitrary boolean expression and hence allows one to formulate conjunctive queries only.

Observe that in contrast to the rest of SECONDO, the optimizer is not data model independent. In particular, the queries that can be formulated in the SQL-like language are limited by the structure of SQL and also the fact that we assume a relational model. On the other hand, the core capability of the optimizer to derive efficient plans for conjunctive queries is needed in any kind of data model.

The optimizer in its up-to-date version (the one running in SECONDO) is described completely in [Güt02], the document containing the source code of the optimizer. That document is available from the SECONDO system by changing (in a shell) to the `Optimizer` directory and saying

```
make
pdview optimizer
pdview optimizer
```

After the second call of `pdview` also the table of contents has been generated and included. If any questions remain open in the sequel, refer to that document. A somewhat detailed description of optimization in SECONDO can also be found in [GBA+04].

6.1.2 Optimization Algorithm

The optimizer employs an as far as we know novel optimization algorithm which is based on *shortest path search in a predicate order graph*. This technique is remarkably simple to implement, yet efficient.

A predicate order graph (POG) is the graph whose nodes represent sets of evaluated predicates and whose edges represent predicates, containing all possible orders of predicates. Such a graph for three predicates *p*, *q*, and *r* is shown in Figure 3.

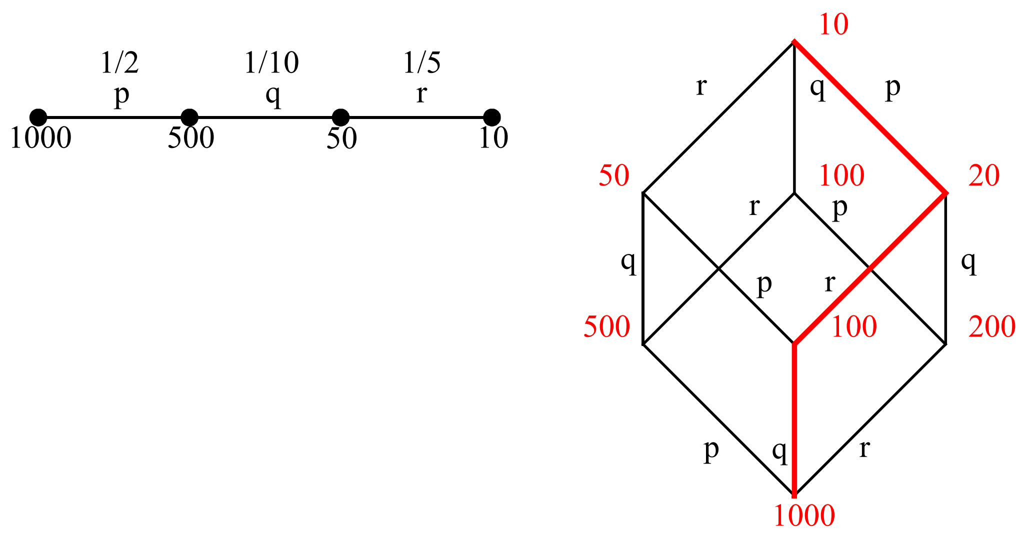

Figure 3: A predicate order graph for 3 predicates

Here the bottom node has no predicate evaluated and the top node has all predicates evaluated. The example illustrates, more precisely, possible sequences of selections on an argument relation of size 1000. If selectivities of predicates are given (for *p* it is 1/2, for *q* 1/10, and for *r* 1/5), then we can annotate the POG with sizes of intermediate results as shown, assuming that all predicates are independent (not *correlated*). This means that the selectivity of a predicate is the same regardless of the order of evaluation, which of course is not always true.

If we can further compute for each edge of the POG possible evaluation methods, adding a new “executable” edge for each method, and mark the edge with estimated costs for this method, then finding a shortest path through the POG corresponds to finding the cheapest query plan. Figure 4 shows an example of a POG annotated with evaluation methods.

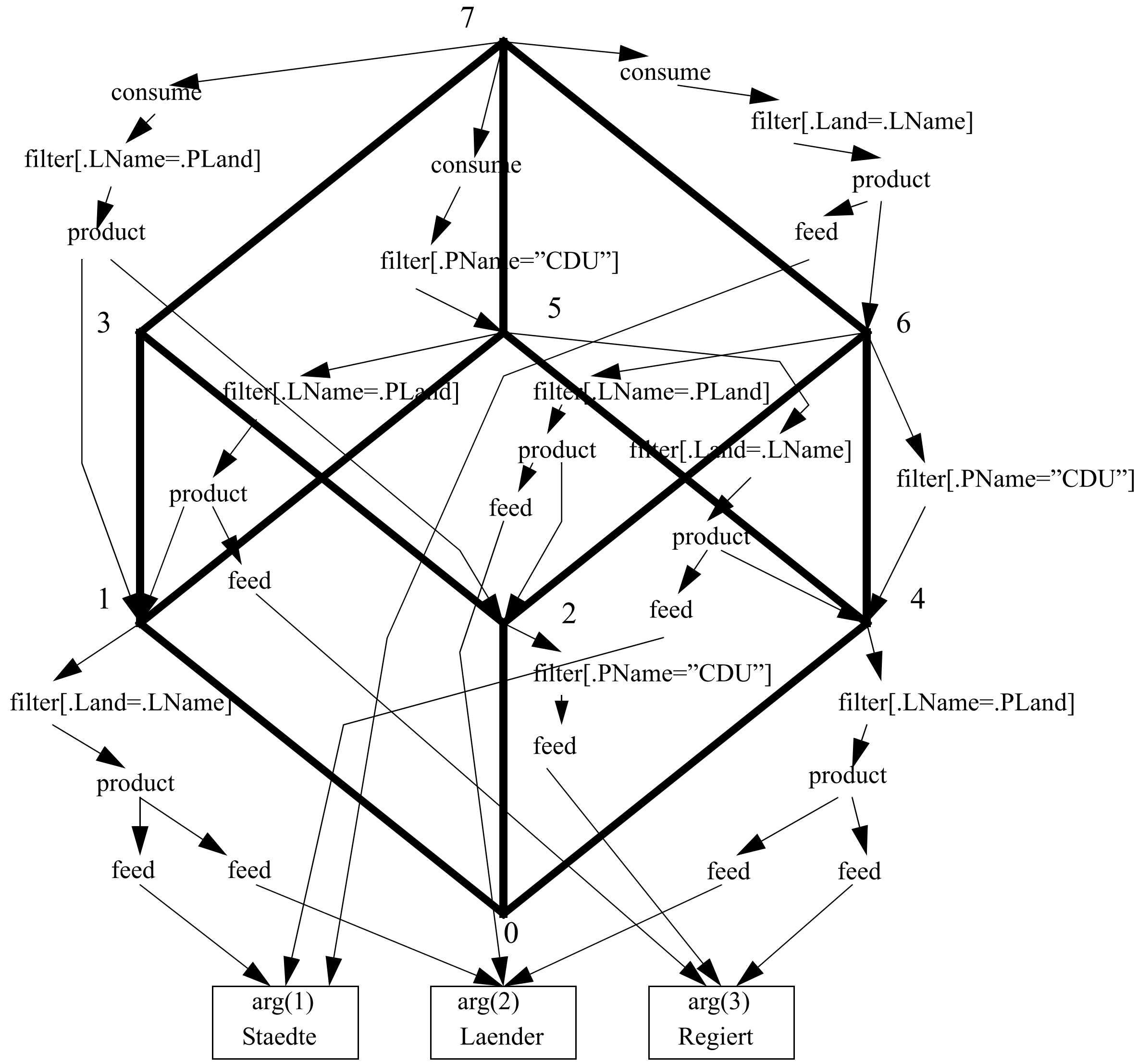

Figure 4: A POG annotated with evaluation methods

In this example, there is only a single method associated with each edge. In general, however, there will be several methods. The example represents the query:

```
select *
from Staedte, Laender, Regiert
where Land = LName and PName = ’CDU’ and LName = PLand
```

for relation schemas

```
Staedte(SName, Bev, Land)
Laender(LName, LBev)
Regiert(PName, PLand)
```

Hence the optimization algorithm proceeds in the following steps (the section numbers indicated refer to [Güt02]):

1. For given relations and predicates, construct the predicate order graph and store it as a set of facts in memory (Sections 2 through 4).
2. For each edge, construct corresponding executable edges, called *plan edges*. This is controlled by optimization rules describing how selections or joins can be translated (Sections 5 and 6).
3. Based on sizes of arguments and selectivities, compute the sizes of all intermediate results. Also annotate edges of the POG with selectivities (Section 7).
4. For each plan edge, compute its cost and store it in memory (as a set of facts). This is based on sizes of arguments and the selectivity associated with the edge and on a cost function (predicate) written for each operator that may occur in a query plan (Section 8).
5. The algorithm for finding shortest paths by Dijkstra is employed to find a shortest path through the graph of plan edges annotated with costs, called *cost edges*. This path is transformed into a SECONDO query plan and returned (Section 9).
6. Finally, a simple subset of SQL in a PROLOG notation is implemented. So it is possible to enter queries in this language. The optimizer determines from it the lists of relations and predicates in the form needed for constructing the POG, and then invokes step 1 (Section 11).

6.2 Programming Extensions

The following kinds of extensions to the optimizer arise when the SECONDO system is extended by new algebras for attribute types, new query processing methods (operators in the relational algebra), or new types of indexes:

1. New algebras for attribute types:
   - Write a display function for a new type constructor to show corresponding values.
   - Define the syntax of a new operator to be used within SQL and in SECONDO.
   - Write optimization rules to perform selections and joins involving a new operator, including rules to use appropriate indexes.
   - Define a cost function or constant for using the operator. [1]
2. New query processing operators in the relational algebra:
   - Define the operator syntax to be used in SECONDO.
   - Write optimization rules using the new operator.
   - Write a cost function (predicate) for the new operator.
3. New types of indexes:
   - Define the operator syntax to be used in SECONDO for search operations on the index.
   - Write optimization rules using the index.
   - Write cost functions for access operations.

---

1. In the current version of the optimizer this is not yet done for operations on attribute types. All such operations are assumed to have cost 1. Obviously this is a simplification and in particular wrong for data types of variable size, e.g. *region*. It should be changed in the future.

One can see that the same issues arise for various kinds of SECONDO extensions. In the following subsections we cover:

- Writing a display function for a type constructor
- Defining operator syntax for SQL
- Defining operator syntax for SECONDO
- Writing optimization rules
- Writing cost functions

6.2.1 Writing a Display Predicate for a Type Constructor

This is a relatively easy extension. Moreover, it is not mandatory. If the optimizer does not know about a type constructor, it displays the value as a nested list (as in the other user interfaces).

In PROLOG, "functions" are implemented as predicates, hence we actually need to write a display predicate. More precisely, for the existing predicate `display` we need to write a new rule. The predicate is

```
display(Type, Value) :-
```

Display the `Value` according to its `Type` description.

The predicate is used to display the list coming back from calling SECONDO. For the result of a query, this list has two elements (<type expression>, <value expression>) which are lists themselves, now converted to the PROLOG form. These are used to instantiate the `Type` and `Value` variables. The structure of the `Type` list is used to control the displaying of values in the `Value` list. The rule to display `int` values is very simple:

```
display(int, N) :-
  !,
  write(N).
```

The following is the rule for displaying a relation:

```
display([rel, [tuple, Attrs]], Tuples) :-
  !,
  nl,
  max_attr_length(Attrs, AttrLength),
  displayTuples(Attrs, Tuples, AttrLength).
```

It determines the maximal length of attribute names from the list of attributes `Attr` in the type description and then calls `displayTuples` to display the value list.

```
displayTuples(_, [], _).

displayTuples(Attrs, [Tuple | Rest], AttrLength) :-
  displayTuple(Attrs, Tuple, AttrLength),
  nl,
  displayTuples(Attrs, Rest, AttrLength).
```

This processes the list, calling `displayTuple` for each tuple.

```
displayTuple([], _, _).

displayTuple([[Name, Type] | Attrs], [Value | Values], AttrNameLength) :-
  atom_length(Name, NLength),
  PadLength is AttrNameLength - NLength,
  write_spaces(PadLength),
  write(Name),
  write(' : '),
  display(Type, Value),
  nl,
  displayTuple(Attrs, Values, AttrNameLength).
```

Finally, `displayTuple` for each attribute writes the attribute name and calls `display` again for writing the attribute value, controlled by the type of that attribute.

For example, there is no rule to display the spatial type `point`, so let us add one. The list representation of a point value is `[<x-coord>, <y-coord>]`. We want to display a list `[17.5, 20.0]` as

```
[point x = 17.5, y = 20.0]
```

Hence we write a rule:

```
display(point, [X, Y]) :-
  !,
  write('[point x = '),
  write(X),
  write(', y = '),
  write(Y),
  write(']').
```

The predicate `display` is defined in the file `auxiliary.pl`. There, we insert the new rule at an appropriate place before the last rule which captures the case that no other matching rule can be found.

6.2.2 Defining Operator Syntax for SQL

The SECONDO operators that can be used directly within the SQL language are those working on attribute types such as +, <, `mod`, `inside`, `starts`, `distance`, ... In order to not get confused we require that such operators are written with the same syntax in SQL and SECONDO. In this subsection we discuss what needs to be done so that the operator syntax is acceptable to PROLOG within the SQL (term) notation. We also need to tell the optimizer, how to translate such an operator to SECONDO syntax. This is covered in the next subsection.

Some of these operators, for example, +, -, *, /, <, >, are also in PROLOG defined as operators with a specific syntax, so you can write them in this syntax within a PROLOG term without further specification. The operators above are all written in infix syntax; so this is possible also within an SQL where-clause.

For operators that are not yet defined in PROLOG, there are two cases:

- Any operator can be written in prefix syntax, for example

```
length(x), distance(x, y), translate(x, y, z)
```

This is just the standard PROLOG term notation, so it is fine with PROLOG to write such terms.

- If an operator is to be used in infix syntax, we have to tell PROLOG about it by adding an entry to the file `opsyntax.pl`. For example, to tell that `touches` is an infix operator, we write:

```
:- op(800, xfx, touches).
```

The other arguments besides `touches` determine operator priority and syntax as well as associativity. For our use, please leave the other arguments unchanged.

6.2.3 Defining Operator Syntax for SECONDO

Within the optimizer, query language expressions (terms) are written in prefix notation, as usual in PROLOG. This happens, for example, in optimization rules. Hence, instead of the SECONDO notation

```
x y product filter[cond] consume
```

a term of the form

```
consume(filter(product(x, y), cond))
```

is manipulated. For any operator, the optimizer must know how to translate it into SECONDO notation. This holds for query processing operators (`feed`, `filter`, ...) not visible at the SQL level as well as for operators on attribute types such as <, `touches`, `length`, `distance`. There are three different ways how operator syntax can be defined, at increasing levels of complexity.:

(1) By *default*. For operators with 1, 2, or 3 arguments that are not treated explicitly, there is a default syntax, namely:

- 1 or 3 arguments: prefix syntax
- 2 arguments: infix syntax

This means, the operators `length`, `touches`, and `translate` with 1, 2, and 3 arguments, respectively, are automatically written as:

```
length(x), x touches y, translate(x, y, z)
```

(2) By a *syntax specification* via predicate `secondoOp` in the file `opsyntax.pl`. This predicate is defined as:

```
secondoOp(Op, Syntax, NoArgs) :-
```

`Op` is a SECONDO operator written in `Syntax`, with `NoArgs` arguments.

Here are some example specifications:

```
secondoOp(distance, prefix, 2).

secondoOp(feed, postfix, 1).
secondoOp(consume, postfix, 1).
```

```
secondoOp(count, postfix, 1).
secondoOp(product, postfix, 2).
secondoOp(filter, postfixbrackets, 2).
secondoOp(loopjoin, postfixbrackets, 2).
secondoOp(exactmatch, postfixbrackets, 3).
```

Not all possible cases are implemented. What can be used currently, is:

- `postfix`, 1 or 2 arguments: corresponds to _ # and _ _ #
- `postfixbrackets`, 2 or 3 arguments, of which the last one is put into the brackets: corresponds to patterns _ # [_] or _ _ # [_]
- `prefix`, 2 arguments: # (_, _)

Observe that `prefix`, either 1 or 3 arguments, and `infix`, 2 arguments, do not need a specification, as they are covered by the default rules mentioned above.

(3) For all other forms, a `plan_to_atom` rule has to be *programmed explicitly.* Such translation rules can be found in Section 5.1.3 of the optimizer source code [Güt02]. The predicate is defined as

```
plan_to_atom(X, Y) :-
```

`Y` is the SECONDO expression corresponding to term `X`.

Although not necessary, we could write an explicit rule to translate the product operator:

```
plan_to_atom(product(X, Y), Result) :-
  plan_to_atom(X, XAtom),
  plan_to_atom(Y, YAtom),
  concat_atom([XAtom, YAtom, 'product '], '', Result),
  !.
```

Actually, these rules are not hard to understand: the general idea is to recursively translate the arguments by calls to `plan_to_atom` and then to concatenate the resulting strings together with the operator and any needed parentheses or brackets in the right order into the result string.

An example, where an explicit rule is needed, is the translation of the `sortmergejoin`, which has 4 arguments:

```
plan_to_atom(sortmergejoin(X, Y, A, B), Result) :-
  plan_to_atom(X, XAtom),
  plan_to_atom(Y, YAtom),
  plan_to_atom(A, AAtom),
  plan_to_atom(B, BAtom),
  concat_atom([XAtom, YAtom, 'sortmergejoin[',
    AAtom, ', ', BAtom, '] '], '', Result),
  !.
```

In general, very little needs to be done for user level operators as they are mostly covered by defaults, and the specification of query processing operators is also in most cases quite simple via the `secondoOp` predicate.

6.2.4 Writing Optimization Rules

Optimization rules are used to translate selection or join predicates associated with edges of the predicate order graph into corresponding SECONDO expressions. This happens in step 2 of the optimization algorithm described in Section 6.1.2. Before we can formulate translation rules, we need to understand the in detail the representation of predicates.

Consider the query

```
select *
from staedte as s, plz as p
where s:sname = p:ort and p:plz > 40000
```

on the optimizer example database `opt` discussed also in the SECONDO User Manual. The optimizer source code [Güt02] contains a rule:

```
example5 :- pog(
  [rel(staedte, s, u), rel(plz, p, l)],
  [pr(attr(s:sName, 1, u) = attr(p:ort, 2, u), rel(staedte, s, u),
    rel(plz, p, l)),
   pr(attr(p:pLZ, 1, u) > 40000, rel(plz, p, l))],
  _, _).
```

This rule corresponds to the query; it says that in order to fulfill the goal `example5`, the predicate order graph should be constructed via calling predicate `pog`. That predicate has four arguments of which only the first two are of interest now. The first argument is a list of relations, the second a list of predicates. A relation is represented as a term, for example,

```
rel(staedte, s, u)
```

which says that the relation name is `staedte`, it has an associated variable `s`, and should in SECONDO be written in upper case (`u`), hence as `Staedte`. A predicate is represented as a term

```
pr(Pred, Rel)
pr(Pred, Rel1, Rel2)
```

of which the first is a selection and the second a join predicate. Hence

```
pr(attr(s:sName, 1, u) = attr(p:ort, 2, u), rel(staedte, s, u),
  rel(plz, p, l))
```

is a join predicate on the two relations `Staedte` and `plz`. An attribute of a relation is represented as a term, for example,

```
attr(s:sName, 1, u)
```

which says that the attribute name is `sName`, it is an attribute of the first of the two relations mentioned in the predicate (`1`), and it should in SECONDO be written in upper case (`u`), hence as `SName`.

Therefore, when you type (after starting the optimizer and opening database `opt`)

```
example5.
```

the predicate order graph for our example query will be constructed. PROLOG replies just `yes`. You can look at the predicate order graph by writing `writeNodes` and `writeEdges`, which lists the nodes and edges of the constructed graph, as follows:

```
16 ?- writeNodes.
Node: 0
Preds: []
Partition: [arp(arg(2), [rel(plz, p, l)], []), arp(arg(1), [rel(staedte, s,
u)], [])]

Node: 1
Preds: [pr(attr(s:sName, 1, u)=attr(p:ort, 2, u), rel(staedte, s, u),
rel(plz, p, l))]
Partition: [arp(res(1), [rel(staedte, s, u), rel(plz, p, l)], [attr(s:sName,
1, u)=attr(p:ort, 2, u)])]

Node: 2
Preds: [pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l))]
Partition: [arp(res(2), [rel(plz, p, l)], [attr(p:pLZ, 1, u)>40000]),
arp(arg(1), [rel(staedte, s, u)], [])]

Node: 3
Preds: [pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l)), pr(attr(s:sName, 1,
u)=attr(p:ort, 2, u), rel(staedte, s, u), rel(plz, p, l))]
Partition: [arp(res(3), [rel(staedte, s, u), rel(plz, p, l)], [attr(p:pLZ,
1, u)>40000, attr(s:sName, 1, u)=attr(p:ort, 2, u)])]

Yes
17 ?-
```

The information about a node contains the node number which encodes in a way explained in [Güt02] which predicates have already been evaluated; it also contains explicitly the list of predicates that have been evaluated, and some more technical information (`Partition`) describing which of the relations involved have already been connected by join predicates. You can observe that in node 0 no predicate has been evaluated and in node 3 both predicates have been evaluated.

```
17 ?- writeEdges.
Source: 0
Target: 1
Term: join(arg(1), arg(2), pr(attr(s:sName, 1, u)=attr(p:ort, 2, u),
rel(staedte, s, u), rel(plz, p, l)))
Result: 1

Source: 0
Target: 2
Term: select(arg(2), pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l)))
Result: 2

Source: 1
Target: 3
Term: select(res(1), pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l)))
Result: 3

Source: 2
Target: 3
```

```
Term: join(arg(1), res(2), pr(attr(s:sName, 1, u)=attr(p:ort, 2, u),
rel(staedte, s, u), rel(plz, p, l)))
Result: 3

Yes
18 ?-
```

The information about edges contains the numbers of the source and target node of the POG, and the selection or join predicate associated with this edge. The `Result` field has the number of the node to which the result of evaluating the selection or join is associated; this is normally, but not always (see [Güt02]) the same as the target node of the edge.

Note that the construction of the predicate order graph does not at all depend on the representation of relations or attributes; it does only need to know by a representation `pr(x, a, b)` that this is a join predicate on relations `a` and `b`. For example, you can type

```
pog([a, b, c, d], [pr(x, a), pr(y, b), pr(z, a, b), pr(w, b, c), pr(v, c,
d)], _, _).
```

and the system will construct a POG for the given four relations with two selection and three join predicates. Try it!

After the somewhat lengthy introduction to this subsection we know what the predicates look like that should be transformed by optimization rules into query plans. For example, they can be

```
select(arg(2), pr(attr(p:pLZ, 1, u)>40000, rel(plz, p, l)))

join(arg(1), res(2), pr(attr(s:sName, 1, u)=attr(p:ort, 2, u),
  rel(staedte, s, u), rel(plz, p, l)))
```

In these terms, `arg(N)` refers to the argument number `N` in the construction of the POG, hence one of the original relations, and `res(M)` refers to the intermediate result associated with the node `M` of the POG. Note that an intermediate result is assumed and required to be a stream of tuples so that optimization rules can use stream operators for evaluating predicates on them.

Optimization rules are given in Section 5.2 of the optimizer [Güt02]. Let us consider some example rules.

Translating the Arguments

```
res(N) => res(N).

arg(N) => feed(rel(Name, *, Case)) :-
  argument(N, rel(Name, *, Case)), !.

arg(N) => rename(feed(rel(Name, Var, Case)), Var) :-
  argument(N, rel(Name, Var, Case)).
```

These rules describe how the arguments of a selection or join predicate should be translated. Translation is defined by the predicate `=>` of arity 2 which has been defined as an infix operator in PROLOG. One might also have called the predicate `translate` and then written, for example

```
translate(res(N), res(N)).
```

However, the form with the “arrow” looks more intuitive; the arrow can be read as “translates into”. The rules above say that a term of the form res(N) is not changed by translation. For `arg(N)`, which is a stored relation, we look up its name and then apply a `feed` and possibly an additional `rename` to convert it into a stream of tuples.

PROLOG is a wonderful environment for testing and debugging. Assuming that we have executed `example5`, we can directly see how things translate. For example, consider

```
18 ?- arg(1) => X.

X = rename(feed(rel(staedte, s, u)), s) ;

No
19 ?-
```

Hence, we can just pass a term as an argument to the => predicate and a variable for the result and see the translation. We can further check how the translated term is converted into SECONDO notation:

```
21 ?- plan_to_atom(rename(feed(rel(staedte, s, u)), s), X).

X = 'Staedte  feed {s} '

Yes
22 ?-
```

Translating Selection Predicates

Here is a rule to translate a selection predicate:

```
select(Arg, pr(Pred, _)) => filter(ArgS, Pred) :-
  Arg => ArgS.
```

It says that a selection on argument `Arg` can be translated into filtering the stream `ArgS` if `Arg` translates into `ArgS`. A selection can also be translated into an `exactmatch` operation on a B-tree under certain conditions. This is specified by the following three rules:

```
select(arg(N), Y) => X :-
  indexselect(arg(N), Y) => X.

indexselect(arg(N), pr(attr(AttrName, Arg, Case) = Y, Rel)) => X :-
  indexselect(arg(N), pr(Y = attr(AttrName, Arg, Case), Rel)) => X.

indexselect(arg(N), pr(Y = attr(AttrName, Arg, AttrCase), _)) =>
  exactmatch(IndexName, rel(Name, *, Case), Y)
  :-
  argument(N, rel(Name, *, Case)),
  !,
  hasIndex(rel(Name, *, Case), attr(AttrName, Arg, AttrCase), IndexName).
```

The first rule says that translation of a selection on a stored relation can be reduced to a translation of a corresponding index selection (`indexselect(Arg, Pred)` is just a new term introduced by this rule). The second rule reverses the order of arguments for an = predicate in order to find the value for searching the index always on the left hand side. The third rule is the most interesting one. It says that index selection with an equality predicate on a stored relation `arg(N)` can be translated into an `exactmatch` operation after looking up the relation and checking that is has an index called `IndexName` on the relevant attribute. At the moment it is still implicit that this index must be a B-tree.

Translating Join Predicates

Finally, let us consider some rules for translating join predicates.

```
join(Arg1, Arg2, pr(Pred, _, _)) => filter(product(Arg1S, Arg2S), Pred) :-
  Arg1 => Arg1S,
  Arg2 => Arg2S.
```

This means that every join can be translated into filtering the Cartesian product of the streams corresponding to the arguments.

```
join(Arg1, Arg2, pr(X=Y, R1, R2)) => JoinPlan :-
  X = attr(_, _, _),
  Y = attr(_, _, _), !,
  Arg1 => Arg1S,
  Arg2 => Arg2S,
  join00(Arg1S, Arg2S, pr(X=Y, R1, R2)) => JoinPlan.

join00(Arg1S, Arg2S, pr(X = Y, _, _)) => sortmergejoin(Arg1S, Arg2S,
  attrname(Attr1), attrname(Attr2))   :-
  isOfFirst(Attr1, X, Y),
  isOfSecond(Attr2, X, Y).

join00(Arg1S, Arg2S, pr(X = Y, _, _)) => hashjoin(Arg1S, Arg2S,
  attrname(Attr1), attrname(Attr2), 997)   :-
  isOfFirst(Attr1, X, Y),
  isOfSecond(Attr2, X, Y).
```

These rules specify ways for translating an equality predicate. The first rule checks whether on both sides of the = predicate there are just attribute names (rather than more complex expressions).[1] In that case, after translating the arguments into streams, the problem is reduced to translating a corresponding term which has a `join00` functor. The second rule specifies translation of the `join00` term into a `sortmergejoin`, the third into a `hashjoin`, using a fixed number of 997 buckets.

The latter two rules use auxiliary predicates `isOfFirst` and `isOfSecond` to get the name of the attribute (among `X` and `Y`) that refers to the first and the second argument, respectively. Note also that the attribute names passed to `sortmergejoin` or `hashjoin` are given as, for example `attrname(Attr1)` rather than `Attr1` directly. The reason is that a normal attribute name of the form `attr(sName, 1, u)` is converted later into the SECONDO "." notation, hence into `.SName` whereas `attrname(attr(sName, 1, u))` is converted into `SName`.

1. There are other rules corresponding to the first one that allow one to translate to a hash join or a sortmergejoin, even if one or both of the arguments to the equality predicate are expressions. The idea is to first apply an `extend` operation which adds the value of the expression as a new attribute, then performing the join using the new attribute, and finally to remove the added attribute again by applying a `remove` operator.

Using Parameter Functions in Translations

In all the rules we have seen so far, it was possible to use the implicit notation for parameter functions in SECONDO. Recall that the implicit form of a `filter` predicate is written as

```
... filter[.Bev > 500000]
```

whereas the explicit complete form would be

```
... filter[fun(t:TUPLE) attr(t, Bev) > 500000]
```

However, sometimes it is necessary to write the full form in SECONDO. For example, in the `loop-join` operator's parameter function we may need to refer to an attribute of the outer relation explicitly. A join as in our example query

```
select *
from staedte as s, plz as p
where s:sname = p:ort
```

could be formulated as a `loopjoin`:

```
query Staedte feed {s} loopjoin[fun(var1:TUPLE) plz feed {p} fil-
ter[attr(var1, SName_s) = .Ort_p]] consume
```

Although currently there are no translation rules yet in the optimizer using the explicit form, there is support for using it. The PROLOG term representing a parameter function has the form:

```
fun([param(Var1, Type1), ..., param(VarN, TypeN)], Expr)
```

Furthermore, when writing rules involving such functions, variable names need to be generated automatically. There is a predicate available

```
newVariable(Var)
```

which returns on every call a new variable name, namely `var1`, `var2`, `var3`, ... For the type operators such as `TUPLE` that one needs to use in explicit parameter functions, a number of conversions to SECONDO are defined by:

```
type_to_atom(tuple, 'TUPLE').
type_to_atom(tuple2, 'TUPLE2').
type_to_atom(group, 'GROUP').
```

The `attr` operator of SECONDO is available under the name `attribute` (in PROLOG) in order to avoid confusion with the `attr(_, _, _)` notation for attribute names. Of course, it is converted back to `attr` in the `plan_to_atom` rule.

Hence a PROLOG term corresponding to

```
Staedte feed filter[fun(t:TUPLE) attr(t, Bev) > 500000]
```

is

```
filter(feed(rel(staedte, *, u)),
  fun([param(t, tuple)], attribute(t, attrname(attr(bev, 0, u))) > 500000))
```

Showing Translations

One can see the translations that the optimizer generates for all edges of a given POG by the goal `writePlanEdges`. For the `example5` above we get:

```
17 ?- example5.

Yes
18 ?- writePlanEdges.
Source: 0
Target: 1
Plan: Staedte  feed {s}  plz  feed {p}  product  filter[(.SName_s = .Ort_p)]
Result: 1

Source: 0
Target: 1
Plan: Staedte  feed {s}  loopjoin[plz_Ort plz  exactmatch[.SName_s] {p} ]
Result: 1

Source: 0
Target: 1
Plan: Staedte  feed {s} plz  feed {p} sortmergejoin[SName_s, Ort_p]
Result: 1

Source: 0
Target: 1
Plan: Staedte  feed {s} plz  feed {p} hashjoin[SName_s, Ort_p, 997]
Result: 1

Source: 0
Target: 2
Plan: plz  feed {p}  filter[(.PLZ_p > 40000)]
Result: 2

Source: 1
Target: 3
Plan: res(1)  filter[(.PLZ_p > 40000)]
Result: 3

Source: 2
Target: 3
Plan: Staedte  feed {s}  res(2)  product  filter[(.SName_s = .Ort_p)]
Result: 3

Source: 2
Target: 3
Plan: Staedte  feed {s} res(2) sortmergejoin[SName_s, Ort_p]
Result: 3

Source: 2
Target: 3
Plan: Staedte  feed {s} res(2) hashjoin[SName_s, Ort_p, 997]
Result: 3

Yes
19 ?-
```

6.2.5 Writing Cost Functions

The next step in the optimization algorithm described in Section 6.1.2, step 3, is to compute the sizes of all intermediate results and associate the selectivities of predicates with all edges. No extensions are needed in this step. We can call for an execution of this step by the goal `assignSizes` (`delete-Sizes` to remove them again) and see the result by `writeSizes`. Continuing with `example5`, we have:

```
20 ?- assignSizes.

Yes
21 ?- writeSizes.
Node: 1
Size: 7419.81

Node: 2
Size: 22696.9

Node: 3
Size: 4080.89

Source: 0
Target: 1
Selectivity: 0.0031

Source: 0
Target: 2
Selectivity: 0.55

Source: 1
Target: 3
Selectivity: 0.55

Source: 2
Target: 3
Selectivity: 0.0031

Yes
22 ?-
```

The next step 4 is to assign costs to all generated plan edges. This step can be called explicitly by the goal `createCostEdges` (removal by `deleteCostEdges`), and plan edges annotated with costs can be listed by `writeCostEdges`. For `example5`, we have now:

```
29 ?- createCostEdges.

Yes
30 ?- writeCostEdges.
Source: 0
Target: 1
Plan: Staedte  feed {s}  plz  feed {p}  product  filter[(.SName_s = .Ort_p)]
Result: 1
Size: 7419.81
Cost: 8.23362e+006
```

```
Source: 0
Target: 1
Plan: Staedte  feed {s}  loopjoin[plz_Ort plz  exactmatch[.SName_s] {p} ]
Result: 1
Size: 7419.81
Cost: 75027

Source: 0
Target: 1
Plan: Staedte  feed {s} plz  feed {p} sortmergejoin[SName_s, Ort_p]
Result: 1
Size: 7419.81
Cost: 157724

Source: 0
Target: 1
Plan: Staedte  feed {s} plz  feed {p} hashjoin[SName_s, Ort_p, 997]
Result: 1
Size: 7419.81
Cost: 92569.4

Source: 0
Target: 2
Plan: plz  feed {p}  filter[(.PLZ_p > 40000)]
Result: 2
Size: 22696.9
Cost: 89962.1

Source: 1
Target: 3
Plan: res(1)  filter[(.PLZ_p > 40000)]
Result: 3
Size: 4080.89
Cost: 12465.3

Source: 2
Target: 3
Plan: Staedte  feed {s}  res(2)  product  filter[(.SName_s = .Ort_p)]
Result: 3
Size: 4080.89
Cost: 3.8703e+006

Source: 2
Target: 3
Plan: Staedte  feed {s} res(2) sortmergejoin[SName_s, Ort_p]
Result: 3
Size: 4080.89
Cost: 71374

Source: 2
Target: 3
Plan: Staedte  feed {s} res(2) hashjoin[SName_s, Ort_p, 997]
Result: 3
Size: 4080.89
Cost: 40289.9
```

```
Yes
31 ?-
```

Here we can see that each plan edge has been annotated with the expected size of the result at the result (usually target) node of that edge. This is the same size as in the listing for `writeSizes`; it has just been copied to the plan edges. More important is the computation of the cost for each plan edge, which is listed in the `Cost` field.

The cost for an edge is computed by a predicate `cost` defined in Section 8.1 of the optimizer [Güt02]. The predicate is:

```
cost(Term, Sel, Size, Cost) :-
```

> The cost of an executable `Term` representing a predicate with selectivity `Sel` is `Cost` and the size of the result is `Size`. Here `Term` and `Sel` have to be instantiated, and `Size` and `Cost` are returned.

The predicate `cost` is called by the predicate `createCostEdges` which passes to it a term (resulting from translation rules and associated with a plan edge) and the selectivity associated with that edge. The predicate is then evaluated by recursively descending into the term. For each operator applied to some arguments, the cost is determined by first computing the cost of producing the arguments and the size of each argument and then computing the cost and result size for evaluating this operator.

Somewhere in the term is an operator that actually realizes the predicate associated with the edge. For example, this could be the `filter` or the `sortmergejoin` operator. This operator uses the selectivity `Sel` passed to it to determine the size of its result.

We can see the existing plan edges after constructing the POG by writing:

```
17 ?- planEdge(Source, Target, Term, Result).
```

One of the solutions listed (for `example5`) is

```
Source = 2
Target = 3
Term = filter(product(rename(feed(rel(staedte, s, u)), s), res(2)),
attr(s:sName, 1, u)=attr(p:ort, 2, u))
Result = 3 ;
```

For each operator or argument occurring in a term there must be a rule describing how to get the result size and cost. Let us consider some of these rules.

```
cost(rel(Rel, _, _), _, Size, 0) :-
  card(Rel, Size).

cost(res(N), _, Size, 0) :-
  resultSize(N, Size).
```

These rules determine the size and cost of arguments. In both cases the cost is 0 (there is no computation involved yet) and the size is looked up. For a stored relation it is found via a fact `card(Rel, Size)` stored in the file `database.pl` (see the SECONDO User Manual); for an intermediate result it was computed by `assignSizes` and can be looked up via predicate `resultSize`. The `Sel` argument passed is not used.

```
cost(feed(X), Sel, S, C) :-
  cost(X, Sel, S, C1),
  feedTC(A),
  C is C1 + A * S.
```

This is the rule for the `feed` operator. It first determines size `S` and cost `C1` for the argument `X`. The size of the result is for `feed` the same as the size of the argument (relation). The cost is determined by using a “feed tuple constant” that describes the cost for evaluating feed on one tuple. Such constants are determined experimentally and stored in a file `operators.pl` (see Appendix C of the optimizer [Güt02]). There we find an entry

```
feedTC(0.4).
```

The cost for evaluating feed is therefore `C1` (we know that is 0 by the rule above) plus 0.4 times the number of tuples of the argument relation.

```
cost(rename(X, _), Sel, S, C) :-
  cost(X, Sel, S, C1),
  renameTC(A),
  C is C1 + A * S.
```

The rule for `rename` is quite similar. The operator just passes the tuple that it receives to the next operator; the `rename` tuple constant happens to be 0.1. The cost is added to the cost of the argument.

```
cost(product(X, Y), _, S, C) :-
  cost(X, 1, SizeX, CostX),
  cost(Y, 1, SizeY, CostY),
  productTC(A, B),
  S is SizeX * SizeY,
  C is CostX + CostY * SizeY * B + S * A.
```

The `product` operator first collects its second argument stream `Y` into a buffer. It then processes the first argument stream `X`, combining each of its tuples with each tuple in the buffer. The result size is, as everyone knows, the product of the sizes of the arguments. The cost is the sum of the costs for producing the arguments, reading argument `Y` into the buffer (per tuple cost given by constant `B`), and producing product tuples (per tuple cost `A`).

```
cost(filter(X, _), Sel, S, C) :-
  cost(X, 1, SizeX, CostX),
  filterTC(A),
  S is SizeX * Sel,
  C is CostX + A * SizeX.
```

The `filter` operator determines the cost and size for its argument. Note that it passes a selectivity 1 to the argument cost evaluation, because the selectivity is actually “used” by this operator. The result size for `filter` is determined by applying the `Sel` factor.

For the moment cost estimation is still rather simplistic, but one can see the principles. For example, in the `filter` operator we assume a constant cost for evaluating a predicate regardless of what the predicate is. In the rule above the predicate is not considered. A refinement would be to also model the cost for all operators that can occur in predicates, and then to model the cost for predicate evaluation precisely. In addition one would need for variable size attribute data types statistics about their average size. For example, for a relation with an attribute of type `region`, one should have a predicate (similar to `card`) stating the average number of edges for `region` values in that relation. Alternatively, similar to the current selectivity determination, such statistics could be retrieved by a query to SECONDO and then be stored for further use.

Cost estimation needs to be extended when a new operator is added that is used in translations of predicates. From the algorithm implementing the operator one should understand how the sizes of arguments determine the cost and write a corresponding rule. The relevant factors for the per tuple cost need to be determined in experiments; they should be set relative to the other existing factors in the file `operators.pl`. Of course, the relationship between these factors and the actual running times depend on the machine where the experiments are run.

7 Integrating New Types into User-Interfaces

7.1 Introduction

SECONDO can be extended with new algebras. Therefore, a user-interface should be able to integrate new display functions for the new data types defined in the newly introduced algebras. In the following sections we show how to extend Javagui to able to display the new data types. Afterwards, the appropriate extension for `SecondoTTY` is described.

7.2 Extending the Javagui

Data is exchanged between SECONDO and Javagui via TCP. In general, a main part of this data are SECONDO objects, which are encoded in nested list format. Nested lists of SECONDO objects are sent to the viewer in format `(<type> <value>)`.

There are two ways to integrate a new data type into Javagui: to write a new viewer or to extend the Hoese-viewer. Both ways have advantages and also disadvantages. When extending the Hoese-viewer, the developer only needs to implement how to draw the new object. The functionalities provided by the Hoese-viewer (zoom etc.) can be used without writing additional code. Unfortunately, the Hoese-viewer can not display all kinds of SECONDO objects. For instance, the developer can not define the drawing-order for multiple objects, and it is not possible to display three dimensional objects. Besides, since the Hoese-viewer has only a small area for displaying text, it is not suitable for objects with a big textual representation. Therefore, in a lot of cases it is more reasonable to write a new viewer. Since the Hoese-viewer uses only one display function for a SECONDO object, it is not possible to have alternative representations for objects. By writing a new viewer, the developer can decide how, when and where an object is drawn.

7.2.1 Writing a New Viewer

Every viewer must be part of the `viewer` package. If more than one class is needed to implement a viewer, only the main class should be in the `viewer` package. All other classes should be placed in a new package. Although Javagui consists of a lot of Java classes, the developer only needs to know six of them, namely `SecondoViewer`, `SecondoObject`, `ListExpr`, `MenuVector`, `ID` and `IDManager`. Furthermore, the developer should know the standard Java classes (especially the Java Swing components). The following sections describe these six classes. Before a new viewer is integrated the `makefile` in the `viewer` directory should be changed. The `makefile` in the `Javagui/viewer` directory has to be extended by adding the name of the viewer to the variable `VIEWER_CLASSES` and by adding the names of the packages to the `VIEWER_DIRS` variable.

The ID Class

This class is used to distinguish different SECONDO objects, even though these objects may have the same value. For instance in a viewer, this class can be used to check whether an object is already displayed. To do this, the `equals` method of the class is used.

The IDManager Class

A viewer can create new SECONDO objects. For instance the Hoese-viewer can load object values from files and create SECONDO objects from them. Each SECONDO object must have an own `id.` To get an unused `id`, use the `getNextID` method of this class.

The MenuVector Class

Each viewer can extend the main menu of Javagui with its own entries. The `MenuVector` class is used for this purpose. Normally, a viewer developer should only use the `addMenu(JMenu)` method to extend the menu. The created `MenuVector` is the return value of the `getMenuVector` method (see Table 1).

The ListExpr Class

All objects resulting from requests to SECONDO are in nested list format. The `ListExpr` class is the Java representation of such lists. A viewer should extract the desired information from the nested list. To display a SECONDO object, the viewer needs to derive the desired data from a nested list and create Java objects from it.

The SecondoObject Class

An instance of the `SecondoObject` class consists of an `id`, a `name`, a `value`, and some methods to access this data. The `id` is used for internal identification, whereas the `name` identifies an object for the user of Javagui. The `value` is the nested list representation of this object.

The SecondoViewer Class

Every viewer is a subclass of `SecondoViewer`. All abstract methods have to be implemented. Table 1 describes all abstract methods in this class.

`String getName()`	Gets the name of this viewer. This name is displayed in the `Viewers` menu of Javagui.
`boolean addObject(SecondoObject o)`	Adds a SECONDO object. In this method the viewer must analyse the value of `o` (using `o`'s `toListExpr` method) and display it.
`removeObject (SecondoObject o)`	Removes `o` from this viewer.
`void removeAll()`	Removes all objects from this viewer.
`boolean canDisplay(SecondoObject o)`	Normally, a viewer can not display all objects resulting from requests to SECONDO. This method returns `true` if this viewer is able to display the given object.
`boolean isDisplayed(SecondoObject o)`	Returns `true` if `o` is contained in this viewer. Note, that the result of this function can be `true` while `o` is not visible, e.g. in the `StandardViewer` only the selected of many objects is visible.
`boolean selectObject(SecondoObject O)`	Selects `o` in this viewer. How an objects is selected can be specified by the viewer implementor.
`MenuVector getMenuVector()`	Returns the menu extension for this viewer. If no menu extension exists, `null` is returned.
`double getDisplayQuality(SecondoObject SO)`	This method is not abstract. This means, that a viewer implementor may but does not have to implement this method. The result of this method must be in range [0,1]. 0 means, that the viewer can't display this object. 1 means, that this is the best viewer to display the given object. This feature is used when a viewer is selected (see SECONDO User Manual).

Table 1: Methods of SecondoViewer

In addition, this class has a member variable `DEBUG_MODE` of type `boolean`. If this variable has the value `true`, all exceptions caught should be printed out with the help of the `printStackTrace` method of the `Exception` class.

Example

In this example a viewer is described, which can display results of inquiries to SECONDO.

List Format

All results of such inquiries to SECONDO are nested lists with two elements. The first element is a symbol atom containing the value inquiry. The second element is again a list with two elements. The first element of this list is a symbol atom describing the kind of the inquiry. Possible values are databases, types, objects, constructors, operators, algebras or algebra. The structure of the second element depends on this value.

The lists for databases and algebras have the same structure. They consist of symbol atoms describing the names of databases and algebras. Example lists are:

- (inquiry (databases (GEO OPT EUROPE)))
- (inquiry (algebras (StandardAlgebra RelationAlgebra SpatialAlgebra)))

The list for constructors consists of lists of three elements. Each of these lists consists of a symbol describing the constructor name, a list with property names and a list with property values.

(inquiry (constructors (<constructor$_1$>...<constructor$_n$>))) where
constructor$_i$:= (name <property names> <property values>)

The lists for property names and for property values should have the same length. The element at position x in the value list is the value for the name at the same position in the list of names. All elements in these lists are atomic.

The list structure for operators is the same as for constructors. Only the keyword is different.

The value list for algebra has two elements. The first element is a symbol atom describing the name of the requested algebra. The second element is a list of two elements describing the type constructors and the operators of this algebra. The format of these lists is the same as in the lists above.

The structure for types is the following:

- (inquiry (types (TYPES <type$_1$> ... <type$_n$>)))

Each <type$_i$> is in format:

- (TYPE <name> <value>)

where TYPE is a keyword, <name> is a symbol and <value> describes the type. Since the possible types depend on the currently used algebras, <value> has not a fixed structure.

The lists for objects are similiar to the lists for types:

- (inquiry (objects (OBJECTS <object$_1$> ... <object$_n$>)))

where <object$_i$> is a list built as follow:

- (OBJECT <name> <typename> <type>)

where `OBJECT` is a keyword, `<name>` a symbol containing the name of the object, `<typename>` is a list containing the user defined name of the type as a symbol or an empty list if no name exists, and `<type>` is a list describing the type.

Building the Viewer

The name for this viewer is "InquiryViewer". All objects are formatted with help of `html` code. The layout of this viewer is very simple. At the top, there is an option bar for selecting an object. Located at the bottom, there is a field and a button supporting searching within the text. In the remaining area the formatted textual representation of the selected object is shown. The package of this viewer is `viewer`. Since graphical elements are used, a few imports are needed. In addition, the access to `ListExpr` and `SecondoObject` is required. (See the source code in Appendix A.)

The viewer has three components to manage `SecondoObject`s: a `ComboBox` containing the names of `SecondoObject`s, a `Vector` containing the `SecondoObject`s and another `Vector` containing the `html` code for `SecondoObject`s. The connection between the different representations of an object is given by the indices of the objects in these containers. To extend the main menu of Javagui the viewer has a `MenuVector MV`. For displaying objects a `JEditorPane` is used.

The `getName` method just returns the string "InquiryViewer". The `isDisplayed` method checks whether the given object is contained in the vector `SecondoObjects` or not. The method `removeObject` removes a given object from all of its representations. All representations can be emptied with the `removeAll` method. After an object is selected from a combobox, the position of this object in the `SecondoObjects` vector is determined, and then the object can be displayed. The `MenuVector` built in the constructor is returned using the `getMenuVector` method. The code for this methods and the code for html formatting is given in Appendix A. The remaining methods are described here in detail.

The `canDisplay` method checks whether this viewer can display a given `SecondoObject`. This is done by checking the list format described above. The first element of the list must be a symbol atom with the content "inquiry". The second element has to be a list of two elements. The first element of this list must be again a symbol atom. The value of this symbol must be an element of the set {"databases", "constructors", "operators", "algebras", "algebra", "types", "objects"}.

```
public boolean canDisplay(SecondoObject o){
  ListExpr LE = o.toListExpr(); // get the nested list of o
  if(LE.listLength()!=2) // the length must be two
    return false;
  // the first element must be an symbol atom with content "inquiry"
  if(LE.first().atomType()!=ListExpr.SYMBOL_ATOM ||
     !LE.first().symbolValue().equals("inquiry"))
    return false;
  ListExpr VL = LE.second();
  // the length of the second element must again be two
  if(VL.listLength()!=2)
    return false;
  ListExpr SubTypeList = VL.first();
  // the first element of this list must be a symbol atom
```

```
  if(SubTypeList.atomType()!=ListExpr.SYMBOL_ATOM)
    return false;
  String SubType = SubTypeList.symbolValue();
  // check for supported "sub types"
  // the used constants just contain the appropriate String
  if(SubType.equals(DATABASES) || SubType.equals(CONSTRUCTORS) ||
      SubType.equals(OPERATORS) || SubType.equals(ALGEBRA) ||
      SubType.equals(ALGEBRAS) || SubType.equals(OBJECTS) ||
      SubType.equals(TYPES))
    return true;
  return false;
}
```

Because this viewer is very good for displaying objects resulting from inquiries, the `getDisplayQuality` method is overwritten.

```
public double getDisplayQuality(SecondoObject SO){
    if(canDisplay(SO))
       return 0.9;
    else
       return 0;
 }
```

The constructor of this class builds the graphical components and initializes the objects managing the SECONDO objects. Besides, the menu is extended.

```
public InquiryViewer(){
  // add the components
  setLayout(new BorderLayout());
  add(BorderLayout.NORTH,ComboBox);
  add(BorderLayout.CENTER,ScrollPane);
  HTMLArea.setContentType("text/html");
  HTMLArea.setEditable(false);
  ScrollPane.setViewportView(HTMLArea);

  // build the panel for search within the text
  JPanel BottomPanel = new JPanel();
  BottomPanel.add(CaseSensitive);
  BottomPanel.add(SearchField);
  BottomPanel.add(SearchButton);
  add(BottomPanel,BorderLayout.SOUTH);
  CaseSensitive.setSelected(true);

  // register functions to the components
  ComboBox.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent evt){
    showObject();
  }});

  SearchField.addKeyListener(new KeyAdapter(){
  public void keyPressed(KeyEvent evt){
    if( evt.getKeyCode() == KeyEvent.VK_ENTER )
      searchText();
  }});
```

```
    SearchButton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        searchText();
    }});

    // build the MenuExtension
    JMenu SettingsMenu = new JMenu("Settings");
    JMenu HeaderColorMenu = new JMenu("header color");
    JMenu CellColorMenu = new JMenu("cell color");
    SettingsMenu.add(HeaderColorMenu);
    SettingsMenu.add(CellColorMenu);
    ActionListener HeaderColorChanger = new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        JMenuItem S = (JMenuItem) evt.getSource();
        HeaderColor = S.getText().trim();
        reformat();
      }
    };
    ActionListener CellColorChanger = new ActionListener(){
      public void actionPerformed(ActionEvent evt){
        JMenuItem S = (JMenuItem) evt.getSource();
        CellColor = S.getText().trim();
        reformat();
      }
    };
    // some colors for the menuextension
    HeaderColorMenu.add("white").addActionListener(HeaderColorChanger);
    HeaderColorMenu.add("silver").addActionListener(HeaderColorChanger);
    CellColorMenu.add("yellow").addActionListener(CellColorChanger);
    CellColorMenu.add("aqua").addActionListener(CellColorChanger);

    MV.addMenu(SettingsMenu);
    }
```

The `addObject` method constructs the html-code for a new object and appends this object to all components managing `SecondoObject` representations.

```
  public boolean addObject(SecondoObject o){
    // check if object is correct
    if(!canDisplay(o))
      return false;
    // already displayed => only select
    if (isDisplayed(o))
      selectObject(o);
    else{
      // build the text and append object to all representations
      ListExpr VL = o.toListExpr().second();
      ObjectTexts.add(getHTMLCode(VL));
      ComboBox.addItem(o.getName());
      SecondoObjects.add(o);
      try{
        // ensure to diplay the new object
        ComboBox.setSelectedIndex(ComboBox.getItemCount()-1);
        showObject();
      }
      catch(Exception e){
```

```
        if(DEBUG_MODE)
          e.printStackTrace();
      }
    }
    return true;
  }
```

Don't forget to extend the makefile in the Javagui/viewer directory.

7.2.2 Extension of the Hoese-Viewer

This viewer can display textual, graphical and temporal objects, but it can not display composite types. The embedded relation type is the only composite type that can be displayed. This means that in the list representation of a SECONDO object (<type> <value>), <type> must be a single symbol atom. To add a new object type to the Hoese-viewer, a new class in the package viewer.hoese.algebras must be implemented. The name of this class must be Dspl<type>, where <type> is the symbol value of <type> in the list above. If another class-name is chosen, the viewer can't find this class. If no new packages are included, the makefiles don't need to be changed. Depending on the kind of the new object type (textual, graphical or temporal), a class implementing different interfaces should be developed. For a simple extension, existing adapter classes can be used.

Creating Textual Objects

In the class for a new textual object, the DsplBase interface should be implemented. There is an adapter class DsplGeneric, which implements all methods from the DsplBase interface by default. The easiest way to add a new textual type is to extend this class. Then only the init method has to be overwritten. For a formatted output within relations, the additional Interface DsplSimple containing another init method has to be implemented.

Example: Inserting Rational Numbers

In this example, rational numbers will be displayed. The nested list structure for such objects looks as follows:

```
(rational (<sign> <intpart> <numDecimal> / <denomDecimal>))
```

where

- <sign> can be either the symbol "+" (positive number), "-" (negative number) or nothing (interpreted as positive)
- <intpart>, <numDecimal> and <denomDecimal> are non negative integer values with <numDecimal> < <denomDecimal>

The output format should be `rat: <sign> <numDecimal> / <demonDecimal>`, where the `<intpart>` from the nested list is integrated in this representation. This class is named `Dsplrational`.

The display class is shown below:

```
package viewer.hoese.algebras;

import sj.lang.ListExpr;
import viewer.hoese.*;

public class Dsplrational extends DsplGeneric implements DsplSimple{
  /* returns a string representing this rational or "ERROR" if
   *the format of the list is wrong
   */
  private String getValueString(ListExpr value){
    int len = value.listLength();
    if(len!=4 && len !=5)
      return "ERROR";
    String result="";
    if(value.listLength()==5){ // with sign
      ListExpr SignList = value.first();
      if(SignList.atomType()!=ListExpr.SYMBOL_ATOM)
        return "ERROR";
      String sign = SignList.symbolValue();
      if(sign.equals("-")) // ignore other values
        result += sign +" ";
      value = value.rest(); // skip the signum
    }
    // check the types
    if( value.first().atomType()!=ListExpr.INT_ATOM ||
      value.second().atomType()!=ListExpr.INT_ATOM ||
      value.fourth().atomType()!=ListExpr.INT_ATOM)
      return "ERROR";
    int intPart = value.first().intValue();
    int numDecimal = value.second().intValue();
    int denomDecimal = value.fourth().intValue();
    result += ""+(denomDecimal*intPart+numDecimal) + " / " + denomDecimal;
    return result;
  }

  public void init(ListExpr type, ListExpr value, QueryResult qr){
    qr.addEntry("rat : " + getValueString(value));
  }
  public void init (ListExpr type,int typewidth,ListExpr value,
                    int valuewidth, QueryResult qr){
    String T = new String(type.symbolValue());
    String V = getValueString(value);
    T=extendString(T,typewidth);
```

```
    V=extendString(V,valuewidth);
    qr.addEntry(T + " : " + V);
    return;
  }
}
```

Inserting a Graphical Object

In a new display class for a graphical object the interface `DsplGraph` has to be implemented. The adapter class is named `DisplayGraph`. If a Java `shape` can be created, only the `init` method has to be overwritten and the variable `RenderObject` must be set. If it is not possible to create a Java `shape`, the methods `init`, `draw` and `getBounds` have to be overwritten. Optionally, the `toString` method can be overwritten to have a special representation in the text area of this viewer.

Example: Inserting a Rectangle Type

The nested list format for a Rectangle is given in the following form:

(rect (<x_1> <y_1> <x_2> <y_2>))

where x_i and y_i are numeric values. Because in Java a class `Rectangle2D.Double` implementing the `shape` interface already exists, only the `init` method has to be overwritten. Here, the implementation of the class `Dsplrect` is shown:

```
package  viewer.hoese.algebras;

import  java.awt.geom.*;
import  java.awt.*;
import  sj.lang.ListExpr;
import  java.util.*;
import  viewer.*;
import  viewer.hoese.*;


/**
  * The displayclass for rectangles
  */
public class Dsplrect extends DisplayGraph {
  /** The internal datatype representation */
  Rectangle2D.Double rect;
  /**
    * Scans the numeric representation of a rectangle
    */
  private void ScanValue (ListExpr v) {
    if (v.listLength() != 4) {
      System.err.println("Error: No correct rectangle expression:"+
                         "4 elements needed");
      err = true;
      return;
    }
    Double X1 = LEUtils.readNumeric(v.first());
    Double Y1 = LEUtils.readNumeric(v.second());
```

```
    Double X2 = LEUtils.readNumeric(v.third());
    Double Y2 = LEUtils.readNumeric(v.fourth());
    if(X1==null || X2==null || Y1==null | Y2==null){
      System.err.println("Error: no correct rectangle"+
                         " expression (not a numeric)");
      err =true;
      return;
    }
    double x1 = X1.doubleValue();
    double x2 = X2.doubleValue();
    double y1 = Y1.doubleValue();
    double y2 = Y2.doubleValue();
    double x = Math.min(x1,x2);
    double w = Math.abs(x2-x1);
    double y = Math.min(y1,y2);
    double h = Math.abs(y2-y1);
    rect = new Rectangle2D.Double(x,y,w,h);
  }

  /**
    * Init. the Dsplrect instance.
    */
  public void init (ListExpr type, ListExpr value, QueryResult qr) {
    AttrName = type.symbolValue();
    ScanValue(value);
    if (err) {
      System.out.println("Error in ListExpr :parsing aborted");
      qr.addEntry(new String("(" + AttrName + ": GA(rectangle))"));
      return;
    }
    else
      qr.addEntry(this);
    RenderObject=rect;
  }
}
```

Including a Graphical Temporal Type

In a display class for a graphical temporal type the `Timed` interface and the `DsplGraph` interface have to be implemented. To do that, only the `DisplayTimeGraph` class has to be extended.

Example: Including a Moving Point

A moving point is a point which changes the position over the time. The point is defined in disjoint intervals.

The list representation for a moving point is:

(mpoint (<unit$_1$>...<unit$_n$>))

In an unit, the point moves from one position to another of a line segment. The two positions may be the same. In this case the point is staying at this position during this time interval. An unit has the following structure:

```
unit := ( <interval> (x1 y1 x2 y2)),
```

where

```
<interval> := (<start> <end> <leftclosed><rightclosed>)
```

`<leftclosed>` and `<rightclosed>` are boolean atoms, describing whether the interval contains the appropriate endpoints.

`<start>` and `<end>` are of type `instant`, which is defined as string in format:

```
year-month-day[-hour:minute[:second[.millisecond]]]
```

where the square brackets denote optional values.

`(x1,y1)` defines the location of the moving point at the beginning of the time interval. `(x2,y2)` is the position of the moving point at the end of the time interval. The moving point moves linearly from `(x1,y1)` to `(x2,y2)` during the given time interval.

In the `init` method the list is converted to an internal representation of a moving point. The bounding box is computed as the minimal rectangle containing all endpoints from the included units.

The `getRenderObject` checks whether the moving point is defined at a given time instant, which means that a unit whose interval contains the given time instant exists. If no unit is found, `getRenderObject` returns `null`. Otherwise the position of this moving point is computed. Around this position a rectangle or a circle is constructed to display this point. The complete source code is given in Appendix B. The class in the appendix additionally supports an old nested list format for moving points.

7.3 Writing New Display Functions for SecondoTTY and SecondoTTYCS

The text based user-interfaces of SECONDO (SecondoTTYBDB and SecondoTTYCS) can display objects in a formatted manner. In order to do this, display functions have to be defined. If no display function exists for a type, the result will be printed in a nested list format. In this section we describe how to write and register new display functions.

To define a display function for a new type, two files, `DisplayTTY.h` (located in the `include` directory) and `DisplayTTY.cpp` (in the `UserInterfaces` directory) have to be changed. In `DisplayTTY.h` you have to add an entry for the new display function with the following format:

```
void DisplayTTY::DisplayType( ListExpr type, ListExpr numType, ListExpr value )
```

where:

- `type`: contains the type in the familiar manner (e.g. `(int)` or `(rel (tuple( ...)))`)

- `numType`: contains also the type description, but here the types are encoded by algebra number and number of the used type constructor in this algebra. (e.g. `(4 3)` or `((2 0)((3 1)(...)))`)
- `value`: contains the value list

7.3.1 Display Functions for Simple Types

Writing a display function for a simple (non-composite) type is very easy. Only the value list has to be analyzed and then written to the standard-output in the desired format.

Example: Display Function for the Point Type

```
void
DisplayTTY::DisplayPoint( ListExpr type, ListExpr numType, ListExpr value)
{
  if(nl->ListLength(value)!=2)
    cout << "Incorrect Data Format";
  else{
    bool err;
    double x = getNumeric(nl->First(value),err);
    if(err){
      cout << "Incorrect Data Format";
      return;
    }
    double y = getNumeric(nl->Second(value),err);
    if(err){
      cout << "Incorrect Data Format";
      return;
    }
    cout << "point: (" << x << "," << y << ")";
  }
}
```

The `getNumeric` function has been defined in `DisplayTTY`. It returns the `double` value of a list containing an integer, a real, or a rational number. If the list does not contain a value of these types, the `err` parameter will be set to `true`. A short output should not end with a `newline`, because this can lead to conflicts when formatting composite types which contain this simple type.

7.3.2 Display Functions for Composite Types

For a composite type (e.g. relation or array), the display function is defined by recursively calling `CallDisplayFunction` for the embedded types. The function `CallDisplayFunction` has four parameters. The last three parameters (`type`, `numType` and `value`) correspond to the parameters in display functions. The first parameter (`idPair`) is used to find the correct display function. For a simple type, `numType` and `idPair` are equal, but for a composite type, `idPair` only contains the "main type" of the `numType` list.

Example: Display Function for an Array Type

The type description of an array is given as `(array <arraytype>)`, where `<arraytype>` can be simple or composite. To find the main type of this list, `<arraytype>` is traversed using depth-first-search until a integer value is found (this integer represents the algebra-number of the main type). For every element of the value list, `CallDisplayFunction` is invoked.

```
void
DisplayTTY::DisplayArray( ListExpr type, ListExpr numType, ListExpr value)
{
  if(nl->ListLength(value)==0)
    cout << "an empty array";
  else{
    ListExpr AType = nl->Second(type);
    ListExpr ANumType = nl->Second(numType);
    // find the idpair
    ListExpr idpair = ANumType;
    while(nl->AtomType(nl->First(idpair))!=IntType)
      idpair = nl->First(idpair);
    int No = 1;
    cout << "*************** BEGIN ARRAY ***************" << endl;
    while( !nl->IsEmpty(value)){
      cout << "--------------- Field No: " << No++ << " ---------------";
      cout << endl;
      CallDisplayFunction(idpair,AType,ANumType,nl->First(value));
      cout << endl;
      value = nl->Rest(value);
    }
    cout << "***************  END ARRAY  ***************";
  }
}
```

7.3.3 Register Display Functions

To register a new display function, just invoke `InsertDisplayFunction` in the `initialize` function of `DisplayTTY`. The calls for the above described functions are as follows:

```
InsertDisplayFunction( "array", &DisplayArray);
InsertDisplayFunction( "point", &DisplayPoint);
```

References

[Alm03] V. T. de Almeida, Object State Diagram in the Secondo System. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Documents”, file “ObjectStateDiagram.pdf”, Sept. 2003.

[BF93] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. Chapter 5.2: Base64 Content-Transfer-Encoding. Also known as RFC 1521, published Online, e.g. http://www.freesoft.org/CIE/RFC/1521/index.htm. September 1993.

[BTree02] Algebra ModuleBTreeAlgebra. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Algebras/BTree”, file “BTreeAlgebra.cpp”, since Dec. 2002.

[Date04] Algebra Module DateTimeAlgebra. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Algebras/DateTime”, file “DateAlgebra.cpp”, since April 2004.

[DG98] S. Dieker and R.H. Güting, Efficient Handling of Tuples with Embedded Large Objects. *Data & Knowledge Engineering 32 (2000)*, 247-269.

[GBA+04] Güting, R.H., T. Behr, V.T. de Almeida, Z. Ding, F. Hoffmann, and M. Spiekermann, SECONDO: An Extensible DBMS Architecture and Prototype. Fernuniversität Hagen, Informatik-Report 313, 2004.

[GFB+97] R.H. Güting, C. Freundorfer, L. Becker, S. Dieker, H. Schenk: Secondo/QP: Implementation of a Generic Query Processor. 10th Int. Conf. on Database and Expert System Applications (DEXA'99), LNCS 1677, Springer Verlag, 66-87, 1999.

[Güt02] R.H. Güting, A Query Optimizer for Secondo. Description and PROLOG Source Code. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Optimizer”, file “optimizer”, from 2002 on.

[Poly02] Algebra Module PolygonAlgebra. Fernuniversität Hagen, Praktische Informatik IV, Secondo System, Directory “Algebras/Polygon”, file “PolygonAlgebra.cpp”, since Sept. 2002.

A The Source for the InquiryViewer

```
package viewer;

import javax.swing.*;
import javax.swing.text.*;
import java.util.Vector;
import java.awt.*;
import java.awt.event.*;
import gui.SecondoObject;
import sj.lang.*;

public class InquiryViewer extends SecondoViewer{

 // define supported subtypes
 private static final String DATABASES = "databases";
 private static final String CONSTRUCTORS="constructors";
 private static final String OPERATORS = "operators";
 private static final String ALGEBRAS = "algebras";
 private static final String ALGEBRA = "algebra";
 private static final String TYPES = "types";
 private static final String OBJECTS ="objects";

 private JScrollPane ScrollPane = new JScrollPane();
 private JEditorPane  HTMLArea = new JEditorPane();
 private JComboBox ComboBox = new JComboBox();
 private Vector ObjectTexts = new Vector(10,5);
 private Vector SecondoObjects = new Vector(10,5);
 private SecondoObject CurrentObject=null;

 private String HeaderColor = "silver";
 private String CellColor ="white";
 private MenuVector MV = new MenuVector();
 private JTextField SearchField = new JTextField(20);
 private JButton SearchButton = new JButton("Search");
 private int LastSearchPos =0;
 private JCheckBox CaseSensitive = new JCheckBox("Case Sensitive");


/* create a new InquiryViewer */
 public InquiryViewer(){
   setLayout(new BorderLayout());
   add(BorderLayout.NORTH,ComboBox);
   add(BorderLayout.CENTER,ScrollPane);
   HTMLArea.setContentType("text/html");
   HTMLArea.setEditable(false);
   ScrollPane.setViewportView(HTMLArea);

   JPanel BottomPanel = new JPanel();
   BottomPanel.add(CaseSensitive);
   BottomPanel.add(SearchField);
   BottomPanel.add(SearchButton);
   add(BottomPanel,BorderLayout.SOUTH);
   CaseSensitive.setSelected(true);
```

```
  ComboBox.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent evt){
       showObject();
    }});

   SearchField.addKeyListener(new KeyAdapter(){
       public void keyPressed(KeyEvent evt){
         if( evt.getKeyCode() == KeyEvent.VK_ENTER )
             searchText();
   }});

   SearchButton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent evt){
          searchText();
      }});


   JMenu SettingsMenu = new JMenu("Settings");
   JMenu HeaderColorMenu = new JMenu("header color");
   JMenu CellColorMenu = new JMenu("cell color");
   SettingsMenu.add(HeaderColorMenu);
   SettingsMenu.add(CellColorMenu);

   ActionListener HeaderColorChanger = new ActionListener(){
     public void actionPerformed(ActionEvent evt){
         JMenuItem S = (JMenuItem) evt.getSource();
         HeaderColor = S.getText().trim();
         reformat();
     }
   };
   ActionListener CellColorChanger = new ActionListener(){
      public void actionPerformed(ActionEvent evt){
         JMenuItem S = (JMenuItem) evt.getSource();
         CellColor = S.getText().trim();
         reformat();
      }
   };

   HeaderColorMenu.add("white").addActionListener(HeaderColorChanger);
   HeaderColorMenu.add("silver").addActionListener(HeaderColorChanger);
   HeaderColorMenu.add("gray").addActionListener(HeaderColorChanger);
   HeaderColorMenu.add("aqua").addActionListener(HeaderColorChanger);
   HeaderColorMenu.add("blue").addActionListener(HeaderColorChanger);
   HeaderColorMenu.add("black").addActionListener(HeaderColorChanger);
   CellColorMenu.add("white").addActionListener(CellColorChanger);
   CellColorMenu.add("yellow").addActionListener(CellColorChanger);
   CellColorMenu.add("aqua").addActionListener(CellColorChanger);
   CellColorMenu.add("lime").addActionListener(CellColorChanger);
   CellColorMenu.add("silver").addActionListener(CellColorChanger);

   MV.addMenu(SettingsMenu);
}

/** returns the html formatted string representation for an atomic list */
private String getStringValue(ListExpr atom){
  int at = atom.atomType();
```

```
    String res = "";
    switch(at){
       case ListExpr.NO_ATOM : return "";
       case ListExpr.INT_ATOM : return ""+atom.intValue();
       case ListExpr.BOOL_ATOM : return atom.boolValue()?"TRUE":"FALSE";
       case ListExpr.REAL_ATOM : return ""+atom.realValue();
       case ListExpr.STRING_ATOM: res = atom.stringValue();break;
       case ListExpr.TEXT_ATOM: res = atom.textValue();break;
       case ListExpr.SYMBOL_ATOM: res = atom.symbolValue();break;
       default : return "";
    }

    res = replaceAll("&",res,"&amp");
    res = replaceAll("<",res,"&lt;");
    res = replaceAll(">",res,"&gt;");
    return res;
  }


  /** include for using older Java-versions */
  private static String replaceAll(String what, String where, String ByWhat){
    StringBuffer res = new StringBuffer();
    int lastpos = 0;
    int len = what.length();
    int index = where.indexOf(what,lastpos);
    while(index>=0){
        if(index>0)
           res.append(where.substring(lastpos,index));
        res.append(ByWhat);
        lastpos = index+len;
        index = where.indexOf(what,lastpos);
    }
    res.append(where.substring(lastpos));
    return res.toString();
  }


/** searchs the text in the textfield in the document and
  * marks its if found
  */
private void searchText(){
  String Text = SearchField.getText();
  if(Text.length()==0){
    MessageBox.showMessage("no text to search");
    return;
  }
  try{
     Document Doc = HTMLArea.getDocument();
     String DocText = Doc.getText(0,Doc.getLength());
     if(!CaseSensitive.isSelected()){
        DocText = DocText.toUpperCase();
        Text = Text.toUpperCase();
     }
     int pos = DocText.indexOf(Text,LastSearchPos);
     if(pos<0){
        MessageBox.showMessage("end of text is reached");
```

```
          LastSearchPos=0;
          return;
       }
       pos = pos;
       int i1 = pos;
       int i2 = pos+Text.length();
       LastSearchPos = pos+1;
       HTMLArea.setCaretPosition(i1);
       HTMLArea.moveCaretPosition(i2);
       HTMLArea.getCaret().setSelectionVisible(true);
   } catch(Exception e){
     if(DEBUG_MODE)
        e.printStackTrace();
        MessageBox.showMessage("error in searching text");
   }
}

/** returns the html string for a single entry for
  * type constructors or operators
  */
 private String formatEntry(ListExpr LE){
    if(LE.listLength()!=3){
      System.err.println("InquiryViewer : error in list (listLength() # 3");
      return "";
    }
    ListExpr Name = LE.first();
    ListExpr Properties = LE.second();
    ListExpr Values = LE.third();
    if(Properties.listLength()!= Values.listLength()){
       System.err.println("InquiryViewer : Warning: lists "+
       "have different lengths ("+Name.symbolValue()+")");
    }

    String res =" <tr><td class=\"opname\" colspan=\"2\">" +
     Name.symbolValue() + "</td></tr>\n";
    while( !Properties.isEmpty() & ! Values.isEmpty()){
      res = res + "   <tr><td class=\"prop\">" +
             getStringValue(Properties.first())+"</td>" +
             "<td class=\"value\">" +
             getStringValue(Values.first())+"</td></tr>\n";
      Properties = Properties.rest();
      Values = Values.rest();
    }

    // handle non empty lists
    // if the lists are correct this never should occur
    while( !Properties.isEmpty()){
      res = res + "   <tr><td class=\"prop\">" +
             getStringValue(Properties.first())+"</td>" +
             "<td>   </td></tr>\n";
      Properties = Properties.rest();
    }
    while(!Values.isEmpty()){
      res = res + "   <tr><td> </td>" +
           "<td class=\"value\">" +
           getStringValue(Values.first())+"</td></tr>\n";
```

```
    Values = Values.rest();
   }

   return res;
 }

/** create the html head for text representation
  * including the used style sheet
  */
 private String getHTMLHead(){
    StringBuffer res = new StringBuffer();
    res.append("<html>\n");
    res.append("<head>\n");
    res.append("<title> inquiry viewer </title>\n");
    res.append("<style type=\"text/css\">\n");
    res.append("<!--\n");
    res.append("td.opname { background-color:"+HeaderColor+
        "; font-family:monospace;"+
        "font-weight:bold; "+
        "color:green; font-size:x-large;}\n");
    res.append("td.prop  {background-color:"+CellColor+
        "; font-family:monospace; font-weight:bold; color:blue}\n");
    res.append("td.value {background-color:"+CellColor+
        "; font-family:monospace; color:black;}\n");
    res.append("-->\n");
    res.append("</style>\n");
    res.append("</head>\n");
    return res.toString();
  }


 /** get the html formatted html Code for type contructors
   */
 private String getHTMLCode_Constructors(ListExpr ValueList){
    StringBuffer res = new StringBuffer();
    if(ValueList.isEmpty())
       return "no type constructors are defined <br>";
    res.append("<table border=\"2\">\n");
     while(!ValueList.isEmpty() ){
       res.append(formatEntry(ValueList.first()));
       ValueList = ValueList.rest();
    }
    res.append("</table>\n");
    return res.toString();
 }

 /** returns the html-code for  operators */
 private String getHTMLCode_Operators(ListExpr ValueList){
   if(ValueList.isEmpty())
       return "no operators are defined <br>";
   // the format is the same like for constructors
   return getHTMLCode_Constructors(ValueList);
 }


 /** returns the html for an Algebra List */
```

```
private String getHTMLCode_Databases(ListExpr Value){
  // the valuelist for algebras is just a list containing
  // symbols representing the database names
  if(Value.isEmpty())
     return "no database exists <br>";
  StringBuffer res = new StringBuffer();
  res.append("<ul>\n");
  while (!Value.isEmpty()){
    res.append("<li> "+Value.first().symbolValue() + " </li>");
    Value = Value.rest();
  }
  res.append("</ul>");
  return res.toString();
}

/** returns the html code for objects */
private String getHTMLCode_Objects(ListExpr Value){
  ListExpr tmp = Value.rest(); // ignore  "SYMBOLS"
  if(tmp.isEmpty())
     return "no existing objects";
  StringBuffer res = new StringBuffer();
  res.append("<h2> Objects - short list </h2>\n ");
  res.append("<ul>\n");
  while(!tmp.isEmpty()){
     res.append("  <li>"+tmp.first().second().symbolValue()+ " </li> \n");
     tmp = tmp.rest();
  }
  res.append("</ul><br><hr><br>");
  res.append("<h2> Objects - full list </h2>\n");
  res.append("<pre>\n"+Value.rest().writeListExprToString() +"</pre>");
  return res.toString();
}

/** returns the html code for types */
private String getHTMLCode_Types(ListExpr Value){
  ListExpr tmp = Value.rest(); // ignore  "TYPES"
  if(tmp.isEmpty())
     return "no existing type";
  StringBuffer res = new StringBuffer();
  res.append("<h2> Types - short list </h2>\n ");
  res.append("<ul>\n");
  while(!tmp.isEmpty()){
     res.append("  <li>"+tmp.first().second().symbolValue()+ " </li> \n");
     tmp = tmp.rest();
  }
  res.append("</ul><br><hr><br>");
  res.append("<h2> Types - full list </h2>\n");
  res.append("<pre>\n"+Value.rest().writeListExprToString() +"</pre>");
  return res.toString();
}



/** returns a html formatted list for algebras */
private String getHTMLCode_Algebras(ListExpr Value){
  // use the same format like databases
```

```
  if(Value.isEmpty())
     return "no algebra is included <br>" +
            "please check your Secondo installation <br>";
  return getHTMLCode_Databases(Value);
}

/** returns the formatted html code for a algebra inquiry */
private String getHTMLCode_Algebra(ListExpr Value){
   // the format is
   // (name ((constructors) (operators)))
   // where constructors and operators are formatted like in the
   // non algebra version
   StringBuffer res = new StringBuffer();
   res.append("<h1> Algebra "+Value.first().symbolValue()+" </h1>\n");
   res.append("<h2> type constructors of algebra: "+
   Value.first().symbolValue()+" </h2>\n");
   res.append( getHTMLCode_Constructors(Value.second().first()));
   res.append("<br>\n<h2> operators of algebra: "+
         Value.first().symbolValue()+"</h2>\n");
   res.append( getHTMLCode_Operators(Value.second().second()));
   return res.toString();
}

/** returns the html code for a given list */
private String getHTMLCode(ListExpr VL){
     StringBuffer Text = new StringBuffer();
     Text.append(getHTMLHead());
     Text.append("<body>\n");
     String inquiryType = VL.first().symbolValue();
     if (inquiryType.equals(DATABASES)){
        Text.append("<h1> Databases </h1>\n");
        Text.append(getHTMLCode_Algebras(VL.second()));
     }
     else if(inquiryType.equals(ALGEBRAS)){
         Text.append("<h1> Algebras </h1>\n");
         Text.append(getHTMLCode_Algebras(VL.second()));
     }
     else if(inquiryType.equals(CONSTRUCTORS)){
         Text.append("<h1> Type Constructors </h1>\n");
         Text.append(getHTMLCode_Constructors(VL.second()));
     }
     else if(inquiryType.equals(OPERATORS)){
         Text.append("<h1> Operators </h1>\n");
         Text.append(getHTMLCode_Operators(VL.second()));
     }
     else if(inquiryType.equals(ALGEBRA)){
         Text.append(getHTMLCode_Algebra(VL.second()));
     }
     else if(inquiryType.equals(OBJECTS)){
         Text.append("<h1> Objects </h1>\n");
         Text.append(getHTMLCode_Objects(VL.second()));
     }
     else if(inquiryType.equals(TYPES)){
         Text.append("<h1> Types </h1>\n");
         Text.append(getHTMLCode_Types(VL.second()));
     }
```

```
     Text.append("\n</body>\n</html>\n");
     return Text.toString();
}



/* adds a new Object to this Viewer and display it */
public boolean addObject(SecondoObject o){
  if(!canDisplay(o))
     return false;
  if (isDisplayed(o))
      selectObject(o);
  else{
     ListExpr VL = o.toListExpr().second();
     ObjectTexts.add(getHTMLCode(VL));
     ComboBox.addItem(o.getName());
     SecondoObjects.add(o);
     try{
        ComboBox.setSelectedIndex(ComboBox.getItemCount()-1);
        showObject();
     }
     catch(Exception e){
       if(DEBUG_MODE)
          e.printStackTrace();
     }
  }
  return true;
}

/** write all htmls texts with a new format */
private void reformat(){
  int index = ComboBox.getSelectedIndex();
  ObjectTexts.removeAllElements();
  for(int i=0;i<SecondoObjects.size();i++){
     SecondoObject o = (SecondoObject) SecondoObjects.get(i);
     ListExpr VL = o.toListExpr().second();
     String inquiryType = VL.first().symbolValue();
     ObjectTexts.add(getHTMLCode(VL));
  }
  if(index>=0)
    ComboBox.setSelectedIndex(index);
}

/* returns true if o a SecondoObject in this viewer */
public boolean isDisplayed(SecondoObject o){
  return SecondoObjects.indexOf(o)>=0;
}

/** remove o from this Viewer */
public void removeObject(SecondoObject o){
   int index = SecondoObjects.indexOf(o);
   if(index>=0){
       ComboBox.removeItem(o.getName());
       SecondoObjects.remove(index);
       ObjectTexts.remove(index);
   }
```

```
 }


 /** remove all containing objects */
 public void removeAll(){
     ObjectTexts.removeAllElements();
     ComboBox.removeAllItems();
     SecondoObjects.removeAllElements();
     CurrentObject= null;
     if(VC!=null)
        VC.removeObject(null);
     showObject();
 }

/** check if this viewer can display the given object */
public boolean canDisplay(SecondoObject o){
    ListExpr LE = o.toListExpr();
    if(LE.listLength()!=2)
       return false;
    if(LE.first().atomType()!=ListExpr.SYMBOL_ATOM ||
       !LE.first().symbolValue().equals("inquiry"))
       return false;
    ListExpr VL = LE.second();
    if(VL.listLength()!=2)
       return false;
    ListExpr SubTypeList = VL.first();
    if(SubTypeList.atomType()!=ListExpr.SYMBOL_ATOM)
      return false;
    String SubType = SubTypeList.symbolValue();
    if(SubType.equals(DATABASES) || SubType.equals(CONSTRUCTORS) ||
       SubType.equals(OPERATORS) || SubType.equals(ALGEBRA) ||
       SubType.equals(ALGEBRAS)  || SubType.equals(OBJECTS) ||
       SubType.equals(TYPES))
       return true;
    return false;
  }

 /* returns the Menuextension of this viewer */
 public MenuVector getMenuVector(){
    return MV;
 }

 /* returns InquiryViewer */
 public String getName(){
    return "InquiryViewer";
 }

 public double getDisplayQuality(SecondoObject SO){
    if(canDisplay(SO))
       return 0.9;
    else
       return 0;
 }

 /* select O */
 public boolean selectObject(SecondoObject O){
```

```
    int i=SecondoObjects.indexOf(O);
    if (i>=0) {
       ComboBox.setSelectedIndex(i);
       showObject();
       return true;
    }else //object not found
      return false;
 }

 private void showObject(){
    String Text="";
    int index = ComboBox.getSelectedIndex();
    if (index>=0){
       HTMLArea.setText((String)ObjectTexts.get(index));
    }   else {
       // set an empty text
       HTMLArea.setText(" <html><head></head><body></body></html>");
    }
    LastSearchPos = 0;
  }
}
```

B The Source for Dsplmovingpoint

```
package  viewer.hoese.algebras;

import  java.awt.geom.*;
import  java.awt.*;
import  viewer.*;
import  viewer.hoese.*;
import  sj.lang.ListExpr;
import  java.util.*;
import  gui.Environment;


/**
  * A displayclass for the movingpoint-type (spatiotemp algebra),
  * 2D with TimePanel
  */
public class Dsplmovingpoint extends DisplayTimeGraph {
  //AffineTransform internAT;
  Point2D.Double point;
  Vector PointMaps;
  Rectangle2D.Double bounds;

  /**
    * Gets the shape of this instance at the ActualTime
    * @param at The actual transformation, used to calculate the correct
    * size.
    * @return Rectangle or Circle Shape if ActualTime is defined
    * otherwise null.
    * @see <a href="Dsplmovingpointsrc.html#getRenderObject">Source</a>
   */
  public Shape getRenderObject (AffineTransform at) {
    double t = RefLayer.getActualTime();
    int index = getTimeIndex(t,Intervals);
    if(index<0){
      RenderObject = null;
      return  RenderObject;
    }
    PointMap pm = (PointMap) PointMaps.get(index);
    Interval in = (Interval)Intervals.get(index);
    double t1 = in.getStart();
    double t2 = in.getEnd();
    double Delta = (t-t1)/(t2-t1);
    double x = pm.x1+Delta*(pm.x2-pm.x1);
    double y = pm.y1+Delta*(pm.y2-pm.y1);
    point = new Point2D.Double(x, y);
    double pixy = Math.abs(Cat.getPointSize()/at.getScaleY());
    double pix = Math.abs(Cat.getPointSize()/at.getScaleX());
    if (Cat.getPointasRect())
      RenderObject = new Rectangle2D.Double(point.getX()- pix/2,
                            point.getY() - pixy/2, pix, pixy);
    else {
      RenderObject = new Ellipse2D.Double(point.getX()- pix/2,
                            point.getY() - pixy/2, pix, pixy);
    }
```

```
  return  RenderObject;
}

/**
 * Reads the coefficients out of ListExpr for a map
 * @param le ListExpr of four reals.
 * @return The PointMap that was read.
 * @see <a href="Dsplmovingpointsrc.html#readPointMap">Source</a>
 */
private PointMap readPointMap (ListExpr le) {
  Double value[] =  {
    null, null, null, null
  };
  if (le.listLength() != 4)
    return  null;
  for (int i = 0; i < 4; i++) {
    value[i] = LEUtils.readNumeric(le.first());
    if (value[i] == null)
      return  null;
    le = le.rest();
  }
  return  new PointMap(value[0].doubleValue(), value[1].doubleValue(),
                       value[2].doubleValue(), value[3].doubleValue());
}

/**
 * Scans the representation of a movingpoint datatype
 * @param v A list of start and end intervals with ax,bx,ay,by values
 * @see sj.lang.ListExpr
 * @see <a href="Dsplmovingpointsrc.html#ScanValue">Source</a>
 */
private void ScanValue (ListExpr v) {
  err = true;
  if (v.isEmpty())
    return;
  while (!v.isEmpty()) {      // unit While maybe empty
    ListExpr aunit = v.first();
    ListExpr tmp = aunit;
    int L = aunit.listLength();
    if(L!=2 && L!=8){
       if(Environment.DEBUG_MODE)
          System.err.println("wrong ListLength in reading moving"+
                             " point unit");
       return;
    }
    // deprecated version of external representation
    Interval in=null;
    PointMap pm=null;
    if (L == 8){
       System.out.println("Warning: using deprecated external"+
                          "representation of a moving point !");
       in = LEUtils.readInterval(ListExpr.fourElemList(aunit.first(),
                aunit.second(), aunit.third(), aunit.fourth()));
       aunit = aunit.rest().rest().rest().rest();
       pm = readPointMap(ListExpr.fourElemList(aunit.first(),
                  aunit.second(), aunit.third(), aunit.fourth()));
```

```
      }
      // the corrected version of external representation
      if(L==2){
         in = LEUtils.readInterval(aunit.first());
         pm = readPointMap(aunit.second());
      }

      if ((in == null) || (pm == null)){
        if(Environment.DEBUG_MODE){
           System.err.println("Error in reading Unit");
           System.err.println(tmp.writeListExprToString());
           if(in==null){
              System.err.println("Error in reading interval");
           }
           if(pm==null){
              System.err.println("Error in reading Start and EndPoint");
           }
        }
        return;
      }
      Intervals.add(in);
      PointMaps.add(pm);
      v = v.rest();
    }
    err = false;
  }

  /**
    * Init. the Dsplmovingpoint instance and calculate the overall bounds
    *and Timebounds
    * @param type The symbol movingpoint
    * @param value A list of start and end intervals with ax,bx,ay,by values
    * @param qr queryresult to display output.
    * @see generic.QueryResult
    * @see sj.lang.ListExpr
    */
  public void init (ListExpr type, ListExpr value, QueryResult qr) {
    AttrName = type.symbolValue();
    ispointType = true;         //to create the desired form
    Intervals = new Vector(value.listLength()+2);
    PointMaps = new Vector(value.listLength()+2);
    ScanValue(value);
    if (err) {
      System.out.println("Dsplmovingpoint Error in ListExpr"+
                         ":parsing aborted");
      qr.addEntry(new String("(" + AttrName + ": GTA(mpoint))"));
      return;
    }
    else
      qr.addEntry(this);
    //ListIterator li=iv.listIterator();
    bounds = null;
    TimeBounds = null;
    for (int j = 0; j < Intervals.size(); j++) {
      Interval in = (Interval)Intervals.elementAt(j);
      PointMap pm = (PointMap)PointMaps.elementAt(j);
```

```
      Rectangle2D.Double r = new Rectangle2D.Double(pm.x1,pm.y1,0,0);
      r = (Rectangle2D.Double)r.createUnion(new
                              Rectangle2D.Double(pm.x2,pm.y2,0,0));
      if (bounds == null) {
        bounds = r;
        TimeBounds = in;
      }
      else {
        bounds = (Rectangle2D.Double)bounds.createUnion(r);
        TimeBounds = TimeBounds.union(in);
      }
    }
  }

  /**
   * @return The overall boundingbox of the movingpoint
   * @see <a href="Dsplmovingpointsrc.html#getBounds">Source</a>
   */
  public Rectangle2D.Double getBounds () {
    return  bounds;
  }

  class PointMap {
    double x1,x2,y1,y2;

    public PointMap (double x1, double y1, double x2, double y2) {
       this.x1 = x1;
       this.y1 = y1;
       this.x2 = x2;
       this.y2 = y2;

    }

    public String toString(){
      return ("[x1,y1 | x2,y2] = ["+x1+","+y1+" <> "+x2+","+y2+"]");
    }
  }
}
```