PointRectangle Algebra

July 2002, R. H. Gueting

2003 - 2006, V. Almeida. Code changes due to interface changes.

Oct. 2006, M. Spiekermann. Introduction of a namespace, string constants and usage of the `static_cast<>` templates. Additionally, more comments and hints were documented.

Sept. 2007, M. Spiekermann. Many code changes to demonstrate new programming interfaces.

1 Overview

This little example algebra provides two type constructors *xpoint* and *xrectangle* and two operators:

1. *inside*, which checks whether a point is within a rectangle, and

2. *intersects* which checks two rectangles for intersection.

2 Preliminaries

2.1 Includes

```
#include "Algebra.h"
#include "NestedList.h"
#include "NList.h"
#include "LogMsg.h"
```

```
#include "QueryProcessor.h"
#include "ConstructorTemplates.h"
#include "StandardTypes.h"
```

The file `Algebra.h` is included, since the new algebra must be a subclass of class Algebra. All of the data available in Secondo has a nested list representation. 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, e.g. `CcInt`, `CcReal`, `CcString`, `CcBool`, which is needed as the result type of the implemented operations. To use them `StandardTypes.h` needs to be included.

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

The variables above define some global references to unique system-wide instances of the query processor and the nested list storage.

2.2 Auxiliaries

Within this algebra module implementation, we have to handle values of four different types defined in namespace symbols: *INT* and *REAL*, *BOOL* and *STRING*. They are constant values of the C++-string class.

Moreover, for type mappings some auxiliary helper functions are defined in the file `TypeMapUtils.h` which defines a namespace *mappings*.

```
#include "TypeMapUtils.h"
#include "Symbols.h"

using namespace symbols;
using namespace mappings;

#include <string>
using namespace std;
```

The implementation of the algebra will be embedded into a namespace in order to avoid name conflicts with other modules.

```
namespace prt {
```

3 Type Constructor *xpoint*

3.1 Data Structure - Class *XPoint*

```
class XPoint
{
 public:
```

Constructors and destructor

```
  XPoint( int x, int y );
  XPoint(const XPoint& rhs);
  ~XPoint();
  int   GetX() const;
  int   GetY() const;
  void SetX( int x );
  void SetY( int y );

  XPoint* Clone();
```

Below the mandatory set of algebra support functions is declared. Note that these functions need to be static member functions of the class. Their implementations do nothing which depends on the state of an instance.

```
  static Word      In( const ListExpr typeInfo, const ListExpr instance,
                       const int errorPos, ListExpr& errorInfo, bool& correct );

  static ListExpr Out( ListExpr typeInfo, Word value );

  static Word      Create( const ListExpr typeInfo );

  static void      Delete( const ListExpr typeInfo, Word& w );

  static void      Close( const ListExpr typeInfo, Word& w );

  static Word      Clone( const ListExpr typeInfo, const Word& w );

  static bool      KindCheck( ListExpr type, ListExpr& errorInfo );

  static int       SizeOfObj();

  static ListExpr Property();

 private:
  inline XPoint() {}
```

Warning: Do never initializations in the default constructor! It will be used in a special way in the cast function which is needed for making a class persistent when acting as an attribute in a tuple. In order to guarantee this we will make this constructor private.

One needs always provide at least a second constructor, here `XPoint( int x, int y )` in order to construct an instance. Moreover, avoid declarations like `XPoint p1;` since these will create an uninitialized class instance. Instead you should use only proper initialized variables like `XPoint(0,0) p1;`

```
  int x;
  int y;

};
```

We recommend to separate strictly class declarations from their implementations. This makes life easier if you want to use the type provided in one algebra in another algebra. Only for the sake of a compact presentation, we did not out source the declarations in special header files in this example algebra.

```
XPoint::XPoint(int X, int Y) : x(X), y(Y) {}

XPoint::XPoint(const XPoint& rhs) : x(rhs.x), y(rhs.y) {}
```

```
XPoint::~XPoint() {}


int XPoint::GetX() const { return x; }
int XPoint::GetY() const { return y; }

void XPoint::SetX(int X) { x = X; }
void XPoint::SetY(int Y) { y = Y; }
```

3.2 List Representation

The list representation of an xpoint is

```
(x y)
```

3.3 *In* and *Out* Functions

The *In*-function gets a nested list representation of an *xpoint* value passed in the variable `instance`. It is represented by the C++ type `ListExpr`. Moreover there is a global pointer variable `nl` which points to the (single) instance of class *NestedList*. This class provides a set of functions which can investigate and manipulate nested lists. For details refer to the file `NestedList.h`.

The parameter `errorInfo` can be used to return specific error information if the retrieved list is not correct. In the latter case the boolean parameter `correct` needs to be set to false.

The return value of the function is of type *Word* which can simply be regarded as a pointer. The query processor operates with this type-less abstraction for objects. If all integrity checks are correct we will return a pointer to a new instance of class *XPoint*.

```
Word
XPoint::In( const ListExpr typeInfo, const ListExpr instance,
            const int errorPos, ListExpr& errorInfo, bool& correct )
{
  Word w = SetWord(Address(0));
  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;
      w.addr = new XPoint(nl->IntValue(First), nl->IntValue(Second));
      return w;
    }
  }
  correct = false;
  cmsg.inFunError("Expecting a list of two integer atoms!");
  return w;
}
```

The *Out*-function will get a pointer to an *XPoint* representation. Before we can use member function of class *XPoint* we need to do a type cast in order to tell the compiler about the object's type.

Note: At this point we can be sure that it is a pointer to type *XPoint*, hence it is safe to do it. But in general, type casts can be a source for *strange* errors, e.g. segmentation faults, if you cast to a type which is not compatible to the object where the pointer belongs to.

```
ListExpr
XPoint::Out( ListExpr typeInfo, Word value )
{
  XPoint* point = static_cast<XPoint*>( value.addr );

  return nl->TwoElemList(nl->IntAtom(point->GetX()),
                         nl->IntAtom(point->GetY()));
}


Word
XPoint::Create( const ListExpr typeInfo )
{
  return (SetWord( new XPoint( 0, 0 ) ));
}

void
XPoint::Delete( const ListExpr typeInfo, Word& w )
{
  delete static_cast<XPoint*>( w.addr );
  w.addr = 0;
}

void
XPoint::Close( const ListExpr typeInfo, Word& w )
{
  delete static_cast<XPoint*>( w.addr );
  w.addr = 0;
}

Word
XPoint::Clone( const ListExpr typeInfo, const Word& w )
{
  XPoint* p = static_cast<XPoint*>( w.addr );
  return SetWord( new XPoint(*p) );
}
```

Here, a clone simply calls the copy constructor but for other types, which may have also a disk part some code for copying the disk parts would be needed also. Often this is implemented in a special member function `Clone()`.

```
int
XPoint::SizeOfObj()
{
  return sizeof(XPoint);
}
```

3.4 Type Describtion

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.

Generally, a property can be a list of any structure which describes the data type. However, currently a structure like the one below has been established to be the standard.

```
ListExpr
XPoint::Property()
{

  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."))));
}
```

3.5 Kind Checking Function

This function checks whether the type constructor is applied correctly. Since type constructor *xpoint* does not have arguments, this is trivial.

```
bool
XPoint::KindCheck( ListExpr type, ListExpr& errorInfo )
{
  //cerr << "KindCheck XPOINT" << endl;
  return (nl->IsEqual( type, XPOINT ));
}
```

3.6 Creation of the Type Constructor Instance

```
TypeConstructor xpointTC(
  XPOINT,                              // name of the type in SECONDO
  XPoint::Property,                    // property function describing signature
  XPoint::Out, XPoint::In,             // Out and In functions
  0, 0,                                // SaveToList, RestoreFromList functions
  XPoint::Create, XPoint::Delete,      // object creation and deletion
  0, 0,                                // object open, save
  XPoint::Close, XPoint::Clone,        // close, and clone
  0,                                   // cast function
  XPoint::SizeOfObj,                   // sizeof function
  XPoint::KindCheck );                 // kind checking function
```

4 Class *XRectangle*

After we have studied the old-style programming interface for a type we will show some more recent alternative programming interfaces for implementing a type.

To define the Secondo type *xrectangle*, we need to (i) define a data structure, that is a class, to (ii) decide about a nested list representation, and (iii) write conversion functions from and to nested list representation.

The function for converting from the list representation is the most involved one, since it has to check that the given list structure is entirely correct.

```
class XRectangle
{
 public:
  XRectangle( int XLeft, int XRight, int YBottom, int YTop );
  XRectangle( const XRectangle& rhs );
  ~XRectangle() {}

  int GetXLeft()   const;
  int GetXRight()  const;
  int GetYBottom() const;
  int GetYTop()    const;

  bool intersects( const XRectangle& r) const;
```

Here we will only implement the three support functions above, since the others have default implementations which can be generated at compile time using C++ template functionality.

```
  static Word     In( const ListExpr typeInfo, const ListExpr instance,
                      const int errorPos, ListExpr& errorInfo, bool& correct );

  static ListExpr Out( ListExpr typeInfo, Word value );

  static Word     Create( const ListExpr typeInfo );
```

In contrast to the example above, we will implement specific *open* and *save* function instead of using the generic persistent mechanism.

```
  static bool     Open( SmiRecord& valueRecord,
                        size_t& offset, const ListExpr typeInfo, Word& value );

  static bool     Save( SmiRecord& valueRecord, size_t& offset,
                        const ListExpr typeInfo, Word& w );

 private:
  XRectangle() {}
  // Since we want to use some default implementations we need
  // to allow access to private members for the class below.
  friend class ConstructorFunctions<XRectangle>;

  int xl;
  int xr;
  int yb;
  int yt;

};
```

```
XRectangle::XRectangle( int XLeft, int XRight, int YBottom, int YTop )
{
  xl = XLeft; xr = XRight; yb = YBottom; yt = YTop;
}

XRectangle::XRectangle( const XRectangle& rhs )
{
  xl = rhs.xl; xr = rhs.xr; yb = rhs.yb; yt = rhs.yt;
}

int XRectangle::GetXLeft()   const { return xl; }
int XRectangle::GetXRight()  const { return xr; }
int XRectangle::GetYBottom() const { return yb; }
int XRectangle::GetYTop()    const { return yt; }
```

4.1 Implementation of Operations

To implement rectangle intersection, we first introduce an auxiliary function which tests if two intervals overlap.

```
bool overlap ( int low1, int high1, int low2, int high2 )
{
  if ( high1 < low2 || high2 < low1 )
    return false;
  else
    return true;
}


bool
XRectangle::intersects( const XRectangle& r ) const
{
  return ( overlap(xl, xr, r.GetXLeft(), r.GetXRight())
           && overlap(yb, yt, r.GetYBottom(), r.GetYTop()) );
}
```

Similar to the *property* function an operator needs to be described. This will now be done in a more structured way by creating a subclass of class *OperatorInfo*.

```
struct intersectsInfo : OperatorInfo {

  intersectsInfo() : OperatorInfo()
  {
    name      = INTERSECTS;
    signature = XRECTANGLE + " x " + XRECTANGLE + " -> " + BOOL;
    syntax    = "_" + INTERSECTS + "_";
    meaning   = "Intersection predicate for two xrectangles.";
  }

}; // don't forget the semicolon here otherwise the compiler returns strange
   // error messages
```

4.2 List Representation and *In*/*Out* Functions

The list representation of an xrectangle is

```
(XLeft XRight YBottom YTop)
```

In contrast to the code examples above we will use here the class *NList* instead of the static functions `nl->f(...)`. Its interface is described in file `NList.h`. It is a simple wrapper for calls like `nl->f(...)` and allows a more object-oriented access to a nested list.

These class was implemented more recently hence you will find much code which uses the older interface. But as you can observe, the code based on *NList* is much more compact, easier to read, understand, and maintain. Thus we recommend to use this interface.

```
Word
XRectangle::In( const ListExpr typeInfo, const ListExpr instance,
                const int errorPos, ListExpr& errorInfo, bool& correct )
{
  correct = false;
  Word result = SetWord(Address(0));
  const string errMsg = "Expecting a list of four integer atoms!";

  NList list(instance);
  // When you check list structures it will be a good advice to detect
  // errors as early as possible to avoid deep nestings of if statements.
  if ( list.length() != 4 ) {
    cmsg.inFunError(errMsg);
    return result;
  }

  NList First = list.first();
  NList Second = list.second();
  NList Third = list.third();
  NList Fourth = list.fourth();

  if ( First.isInt() && Second.isInt()
           && Third.isInt() && Fourth.isInt() )
  {
    int xl = First.intval();
    int xr = Second.intval();
    int yb = Third.intval();
    int yt = Fourth.intval();

    if ( xl < xr && yb < yt )
    {
      correct = true;
      XRectangle* r = new XRectangle(xl, xr, yb, yt);
      result.addr = r;
    }
  }
  else
  {
    cmsg.inFunError(errMsg);
  }
  return result;
}

ListExpr
```

```
XRectangle::Out( ListExpr typeInfo, Word value )
{
  XRectangle* rectangle = static_cast<XRectangle*>( value.addr );
  NList fourElems(
           NList( rectangle->GetXLeft()   ),
           NList( rectangle->GetXRight()  ),
           NList( rectangle->GetYBottom() ),
           NList( rectangle->GetYTop()    )  );

  return fourElems.listExpr();
}
```

The *open* and *save* functions need a *SmiRecord* as argument which contains the binary representation of the type, starting at the position indicated by *offset*. The implementor has to read out or write in data there and adjust the offset. The argument *typeinfo* will be needed only for complex types whose constructors can be parameterized, e.g. rel(tuple(...)).

```
bool
XRectangle::Open( SmiRecord& valueRecord,
                  size_t& offset, const ListExpr typeInfo, Word& value )
{
  //cerr << "OPEN XRectangle" << endl;
  size_t size = sizeof(int);
  int xl = 0, xr = 0, yb = 0, yt = 0;

  bool ok = true;
  ok = ok && valueRecord.Read( &xl, size, offset );
  offset += size;
  ok = ok && valueRecord.Read( &xr, size, offset );
  offset += size;
  ok = ok && valueRecord.Read( &yb, size, offset );
  offset += size;
  ok = ok && valueRecord.Read( &yt, size, offset );
  offset += size;

  value.addr = new XRectangle(xl, xr, yb, yt);

  return ok;
}


bool
XRectangle::Save( SmiRecord& valueRecord, size_t& offset,
                  const ListExpr typeInfo, Word& value )
{
  //cerr << "SAVE XRectangle" << endl;
  XRectangle* r = static_cast<XRectangle*>( value.addr );
  size_t size = sizeof(int);

  bool ok = true;
  ok = ok && valueRecord.Write( &r->xl, size, offset );
  offset += size;
  ok = ok && valueRecord.Write( &r->xr, size, offset );
  offset += size;
  ok = ok && valueRecord.Write( &r->yb, size, offset );
  offset += size;
  ok = ok && valueRecord.Write( &r->yt, size, offset );
  offset += size;

  return ok;
}


Word
XRectangle::Create( const ListExpr typeInfo )
{
  return (SetWord( new XRectangle( 0, 0, 0, 0 ) ));
}
```

The property function is deprecated. Similar as the operator descriptions this will be done by implementing a subclass of *ConstructorInfo*.

```
struct xrectangleInfo : ConstructorInfo {

  xrectangleInfo() : ConstructorInfo() {

    name         = XRECTANGLE;
    signature    = "-> " + SIMPLE;
    typeExample  = XRECTANGLE;
    listRep      =  "(<xleft> <xright> <ybottom> <ytop>)";
    valueExample = "(4 12 8 2)";
    remarks      = "all coordinates must be of type int.";
  }
};
```

As you may have observed, most implementations of the support functions needed for registering a secondo type are trivial to implement. Hence, we offer a template class which provides default implementations (see below). Thus only functions which need to do special things need to be implemented.

5 Creating Operators

5.1 Type Mapping Function

Checks whether the correct argument types are supplied for an operator; if so, it returns a list expression for the result type, otherwise the symbol *typeerror*.

```
ListExpr
RectRectBool( ListExpr args )
{
  NList list(args);
  const string errMsg = "Expecting (xrectangle xrectangle)";

  if ( list.length() != 2 )
    return list.typeError(errMsg);

  if ( list.first().isSymbol(XRECTANGLE) && list.second().isSymbol(XRECTANGLE) )
    return NList(BOOL).listExpr();

  return list.typeError(errMsg);
}
```

```
ListExpr
insideTypeMap( ListExpr args )
{
  NList list(args);
  const string errMsg = "Expecting (xrectangle xrectangle) "
                        "or (xpoint xrectangle)";

  if ( list.length() != 2 )
    return list.typeError(errMsg);

  NList arg1 = list.first();
  NList arg2 = list.second();

  // first alternative: xpoint x xrecangle -> bool
  if ( arg1.isSymbol(XPOINT) && arg2.isSymbol(XRECTANGLE) )
    return NList(BOOL).listExpr();

  // second alternative: xrectangle x xrectangle -> bool
  if ( arg1.isSymbol(XRECTANGLE) && arg2.isSymbol(XRECTANGLE) )
    return NList(BOOL).listExpr();

  return list.typeError(errMsg);
}
```

5.2 Selection Function

A selection function is quite similar to a type mapping function. The only difference is that it doesn't return a type but the index of a value mapping function being able to deal with the respective combination of input parameter types.

Note that a selection function does not need to check the correctness of argument types; it has already been checked by the type mapping function that it is applied to correct arguments.

```
int
insideSelect( ListExpr args )
{
  NList list(args);
  if ( list.first().isSymbol( XRECTANGLE ) )
    return 1;
  else
    return 0;
}


struct insideInfo : OperatorInfo {

  insideInfo() : OperatorInfo()
  {
    name      = INSIDE;

    signature = XPOINT + " x " + XRECTANGLE + " -> " + BOOL;
    // since this is an overloaded operator we append
    // an alternative signature here
    appendSignature( XRECTANGLE + " x " + XRECTANGLE
                                             + " -> " + BOOL );
    syntax    = "_" + INSIDE + "_";
    meaning    = "Inside predicate.";
  }
};
```

5.3 Value Mapping Functions

5.3.1 The *intersects* predicate for two rectangles

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

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

  CcBool* b = static_cast<CcBool*>( result.addr );
  b->Set(true, r1->intersects(*r2));
                                     //the first argument says the boolean
                                     //value is defined, the second is the
                                     //real boolean value)
  return 0;
}
```

5.3.2 The *inside* predicate for a point and a rectangle

```
int
insideFun_PR (Word* args, Word& result, int message, Word& local, Supplier s)
{
  //cout << "insideFun_PR" << endl;
  XPoint* p = static_cast<XPoint*>( args[0].addr );
  XRectangle* r = static_cast<XRectangle*>( args[1].addr );

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

  CcBool* b = static_cast<CcBool*>( result.addr );

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

  b->Set(true, res); //the first argument says the boolean
                     //value is defined, the second is the
                     //real boolean value)
  return 0;
}
```

4.3.3. The *inside* predicate for two rectangles

```
int
insideFun_RR (Word* args, Word& result, int message, Word& local, Supplier s)
{
  //cout << "insideFun_RR" << endl;
  XRectangle* r1 = static_cast<XRectangle*>( args[0].addr );
  XRectangle* r2 = static_cast<XRectangle*>( args[1].addr );
  result = qp->ResultStorage(s);   //query processor has provided
                                   //a CcBool instance to take the result

  CcBool* b = static_cast<CcBool*>( result.addr );

  bool res = true;
  res = res && r1->GetXLeft() >= r2->GetXLeft();
  res = res && r1->GetXLeft() <= r2->GetXRight();

  res = res && r1->GetXRight() >= r2->GetXLeft();
  res = res && r1->GetXRight() <= r2->GetXRight();

  res = res && r1->GetYBottom() >= r2->GetYBottom();
  res = res && r1->GetYBottom() <= r2->GetYTop();

  res = res && r1->GetYTop() >= r2->GetYBottom();
  res = res && r1->GetYTop() <= r2->GetYTop();

  b->Set(true, res); //the first argument says the boolean
                     //value is defined, the second is the
                     //real boolean value)
  return 0;
}
```

6 Implementation of the Algebra Class

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

6.1 Registration of Types

The class *ConstructorFunctions* is a template class and will create many default implementations of functions used by a secondo type for details refer to `ConstructorFunctions.h`. However, some functions need to be implemented since the default may not be sufficient. The default kind check function assumes, that the type constructor does not have any arguments.

```
    ConstructorFunctions<XRectangle> cf;

    // re-assign some function pointers
    cf.create = XRectangle::Create;
    cf.in = XRectangle::In;
    cf.out = XRectangle::Out;

    // the default implementations for open and save are only
    // suitable for a class which is derived from class ~Attribute~, hence
    // open and save functions must be overwritten here.

    cf.open = XRectangle::Open;
    cf.save = XRectangle::Save;

    static TypeConstructor xrectangleTC( xrectangleInfo(), cf );
    AddTypeConstructor( &xpointTC );
    AddTypeConstructor( &xrectangleTC );

    //the lines below define that xpoint and xrectangle
    //can be used in places where types of kind SIMPLE are expected
    xpointTC.AssociateKind( SIMPLE );
    xrectangleTC.AssociateKind( SIMPLE );
```

6.2 Registration of Operators

```
    AddOperator( intersectsInfo(), intersectsFun, RectRectBool );

    // the overloaded inside operator needs an array of function pointers
    // which must be null terminated!
    ValueMapping insideFuns[] = { insideFun_PR, insideFun_RR, 0 };

    AddOperator( insideInfo(), insideFuns, insideSelect, insideTypeMap );
  }
  ~PointRectangleAlgebra() {};
};
```

7 Initialization

Each algebra module needs 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 (if it is built as dynamic link library). The name of the initialization function defines the name of the algebra module by convention it must start with `Initialize<AlgebraName>`.

In order to link the algebra together with the system you must create an entry in the file `makefile.algebra` and to define an algebra ID in the file `Algebras/Management/AlgebraList.i.cfg`.

```
} // end of namespace ~PointRectangle~

extern "C"
Algebra*
InitializePointRectangleAlgebra( NestedList* nlRef, QueryProcessor* qpRef )
{
  // The C++ scope-operator :: must be used to qualify the full name
  return new prt::PointRectangleAlgebra;
}
```

8 Examples and Tests

The file `PointRectangle.examples` contains for every operator one example. This allows to verify that the examples are running and allow to provide a coarse regression for all algebra modules. The command `Selftest <file>` will execute the examples. Without any arguments the examples for all active algebras are executed. This helps to detect side effects, if you have touched central parts of `Secondo` or existing types and operators.

In order to setup more comprehensive automated test procedures one can write a test specification for the *TestRunner* application. You will find the file `example.test` in directory bin and others in the directory `Tests/Testspecs`. There is also one for this algebra.

Accurate testing is often treated as an unpopular daunting task. But it is absolutely inevitable if you want to provide a reliable algebra module.

Try to write tests covering every signature of your operators and consider special cases, as undefined arguments, illegal argument values and critical argument value combinations, etc.