Tutorial: Secondo Algebra Implementation

Stefan Dieker, 12/03/1997

Contents

1 Introduction

1.1 Secondo in very brief

The aim of the Secondo project is the development of a generic database system frame, providing all typical services of a full-fledged DBMS like

- transaction management
- logging and recovery

- multi-user-access control via fine-grained locking
- query processing and optimization

while imposing the least possible restrictions with respect to the used data model and query language. Thus Secondo will be extensible to a greater extent than systems like Genesis [BBG+88], Starburst [HS90], Volcano [Gra94], or the commercially available Illustra Universal Server. On the other hand, Secondo will provide more functionality than toolboxes like the storage manager components of Exodus [CDRS86] or Shore [CDF+94], thereby making the implementation of a new DBMS easier. The system most similar to Secondo is Predator [SLR97], though it is still tightly bound to the relational model.

Obviously, data model and query language *must* be defined somewhere in order to obtain a working system. While the formal basis for this task is provided by the concept of *Second-Order-Signature* (SOS) [Güt93], the actual implementation of the data model (*type constructors* in SOS) and the executable query language (*operators* in SOS) is done in C++.

Another important feature of Secondo is its support of structuring the potentially large amount of types and operators by partitioning them into algebra modules. For instance, a typical relational system might consist of at least two different algebra modules: one for simple mathematical and boolean operations on types like `integer`, `real`, and `boolean`, and another one for relational operations like selection and join on types like `tuple` and `relation`. Of course the relational algebra module may use definitions of the other one in order to employ, e.g., `integer`, `real`, and `boolean` as attribute domains.

1.2 Algebra modules

A Secondo algebra module consists of a set of types (type constructors, to be precise) and a set of operators and is defined by statements in SOS-Format as described in [Güt93].

Type constructors in SOS are meta-types. Via type constructors, the back-end user of a complete Secondo system is not forced to work with a set of predefined types, but may define new types — as long as they are in the scope of an algebra's type constructors.

> **Example:** Let's consider a working Secondo system with two integrated algebra modules:
>
> 1. a standard algebra module, providing types like `int`, `real`, and `string`.
> 2. a relational algebra module, providing types like `tuple` and `rel`.
>
> Secondo allows type constructors to be specified in a way that enables the back-end user to define new types like the following ones:
>
> - `citytuple = tuple((`*`cityname`* `string)(`*`population`* `int)(`*`size`* `real))`
> - `cityrel = rel(citytuple)`.

An *implemented* type constructor of a Secondo algebra module essentially consists of

- a data structure capable to represent a value of a type generated by the actual constructor. Every value of every type has a representation as a single pointer to some structure. Such a structure may contain unique identifiers which can be used to retrieve persistent data.

- an optional *data model*. A data model is a miniature version of the data structure itself used in place of this for cost estimations (e.g. for a relation the number of tuples, the number of used segments, statistics for the distribution of chosen attributes could be taken into account within the model).

Operators consist of

- *value mapping functions*, computing a result value by means of an arbitrary number of input parameters. Each of the input values as well as the return value are of a type constructable by an existing type constructor.
- Some additional functions, supporting query processing and model handling.

1.3 About this document

This is a fully functional C++ source file demonstrating the implementation of a simple Secondo algebra module, formatted by the PD-System [Güt95]. The algebra consists of two type constructors (`ccint` and `ccreal`), three binary mathematical operators (`cc+`, `cc-`, and `cc*`), and four stream operators (`intfeed`, `intcount`, `intfold`, and `intshow`).

Please notice that we don't claim the *semantics* of these type constructors and operators to be sensible; their only use is to show how to *structure* an algebra module in order to make it known to the Secondo frame.

This file does not yet demonstrate

- *Kind checking*, necessary for creating persistent objects of the kinds constructed by `ccint` and `ccreal`
- Implementing complex types by using *SecondoSMI*, Secondo's interface to persistent data structures
- Supplying cost models and their operations.

Demonstration of these Secondo features will be added soon.

The rest of this document is structured as follows: Section 2 contains the `#include`-statements necessary for making the compiler know the Secondo class interfaces. In section 3 we explain in some detail the meaning of type constructors and their implementation. Section 4 elaborates on meaning and implementation of operators. Type constructors and operators are joined to one algebra in section 5. We conclude this document by giving some query examples in section 6.

2 Includes

The header file `AlgebraManagerCC.h` is included in order to have access to the class definitions of classes `TypeConstructor`, `Operator`, and `Algebra`. This classes are the main elements of the interface which enables us to connect an algebra to the Secondo frame.

```
#include "Algebra.h"
```

Inclusion of the header file `QueryProcessorCC.h` introduces the query processor primitives used for implementing stream operators, e.g. operator `intfold` defined in section 4.9.

```
#include "../QueryProcessor/QueryProcessorCC.h"
```

Eventually, `iostream.h` and `stdlib.h` are included in order to have access to standard C++ classes and functions.

```
#include <iostream.h>
#include <stdlib.h>

#include "Tuple.h"
```

3 Type constructors

3.1 Overview

A snapshot of a working Secondo system will show a collection of algebras, each of them "populated" by two different kinds of entities: objects and operators. Operators are fix in terms of number and functionality which are defined by the algebra implementor, as is done in section 4 of this document.

Objects, however, cannot be predefined, since they are constructed and may be deleted at runtime. Even the possible types of objects cannot be predeclared, because they can be declared at runtime, too. Only an algebras's *type constructors* are well known and fix. Consequently, an implementation of an algebra with m type constructors contains m definitions of instances of the predefined class `TypeConstructor`.

An algebra module offers four generic operations for each type constructor which describe all properties of a type constructor that are already known at compile-time:

Create. Allocate internal memory.

Destroy. Deallocate internal memory.

In. Map a nested list representation of a value of the actual type (constructor) to a main memory representation of the same value.

Out. Map a main memory representation of a value of the actual type (constructor) to a nested list representation of the same value.

A type constructor is created by defining an instance of class `TypeConstructor`. Before this instance definition can take place we have to define four functions which implement the generic operations *In*, *Out*, *Create*, and *Destroy* for the actual type constructor. These functions are passed as constructor arguments during `TypeConstructor` instantiation. Within the following sections 3.3 and 3.4, this is done for type constructors `ccint` and `ccreal`, respectively.

3.2 Type constructor implementation auxiliaries

```
bool IsEqual(ListExpr atom, char *s)
{
  STRING sym;
  if (IsAtom(atom) && AtomType(atom) == SymbolType)
  {
    SymbolValue(atom, sym);
    return (!strcmp(sym, s));
  }
  else
    return 0;
}
```

3.3 Type constructor `ccint`

3.3.1 Main memory representation

Each instance of below defined class `CCINT` will be the main memory representation of a value of type `ccint`. It consists of a boolean flag, `defined`, and an integer value, `value`. `defined` may be used to indicate whether an instance of class `CCINT` represents a valid value or not. For instance, a division by zero might result in a `CCINT` object with `defined` set to `false`.

Throughout this tutorial, we don't use the flag but just maintain it in order to demonstrate how to handle complex objects.[1]

```
class CCINT : public Attribute
{
  BOOLEAN defined;
  int value;
public:
    CCINT(){};
  CCINT(int v, BOOLEAN d = 1){value = v; defined = d;};
  ~CCINT(){};
  BOOLEAN GetDefined(){return defined;};
  int GetValue(){return value;};
  void Set(int v, BOOLEAN d = 1){value = v, defined = d;};
  CompResult Compare(const Attribute *arg);
  int Compare(Attribute *){return 0;}; //dummy
  int Sizeof(){ return sizeof(CCINT); };
  ostream &Print(ostream &os){return os << value;}
};

Attribute::CompResult CCINT::Compare(const Attribute *arg)
{
    cout << "Comparing two Integers\n";
    if (((CCINT*)arg)->value < value) return less;
    else if (((CCINT*)arg)->value > value) return greater;
    else return equal;
};
```

[1] Remember that we don't claim the sensibility of the semantics of type constructors and operators of this demo algebra implementation, but just do something in order to show how an algebra can be connected to the Secondo frame.

3.3.2 *Out*-function of type constructor `ccint`

The *out*-function of each type constructors takes as inputs a type description and a pointer to a value of respective type and returns the same value, now represented in nested list format. The function `OutInt` defined in the following C++ code segment performs exactly this task for the type constructor `ccint`.

```
ListExpr OutInt(ListExpr typeinfo, ADDRESS value)
{
  return IntAtom(((CCINT*)value)->GetValue());
}
```

In this very simple example we did not use the type description at all, because

`ccint` is a *constant* type constructor, i.e. type constructor `ccint` can only construct one type: `ccint`. We will need the type description, however, in the case of more powerful type constructors, e.g. to be able to compute the nested list representation of a tuple value we must know the types of the respective attribute values.

Note that we don't represent the `defined`-part of the `ccint`-value passed as parameter due to the fact that we maintain it just for demonstration reasons.

3.3.3 *In*-function of type constructor `ccint`

Complementary to the *out*-function, an *in*-function has to be defined for each type constructor, returning a value's main memory representation. It is calculated by taking the value's nested list representation and its type description as input parameters.

Again, the actual implementation of the *In*-function for type constructor `ccint` does not use the type information in this example due to simplicity of `ccint`.

The last three input parameters will be used to return error information.

```
ADDRESS InInt(ListExpr typeInfo, ListExpr value,
          int errorPos, ListExpr *errorInfo, BOOLEAN *correct)
{
  if (IsAtom(value) && AtomType(value) == IntType)
  {
    *correct = 1;
    return new CCINT(IntValue(value));
  }
  else
  {
    *correct = 0;
    return 0;
  }
}
```

3.3.4 *Create*-function of type constructor `ccint`

We also have to provide a *create*-function for each type constructor. This function may be used to allocate memory sufficient for keeping one instance of the respective type. Such a

function accepts an integer value as argument which is reserved for future use and should be ignored in the moment.

For type constructor `ccint`, we call this function `CreateInt`. It is used to allocate memory sufficient for keeping one instance of CCINT. The `Size`-parameter is not evaluated.

```
ADDRESS CreateInt(int Size)
{
  return new CCINT(0, 0);
}
```

3.3.5 *Destroy*-function of type constructor `ccint`

A type constructor's *destroy*-function is used by the query processor in order to deallocate memory occupied by instances of Secondo objects. They may have been created in two ways:

- as return values of operator calls like those defined in section 4
- by calling a type constructor's `create`-function.

The corresponding function of type constructor `ccint` is called `DeleteInt`.

```
void DeleteInt(ADDRESS *w)
{
  delete *w;
  *w = 0;
}
```

3.3.6 *Cast*-function of type constructor `ccint`

```
Attribute *CastInt(void *addr)
{
  return new (addr) CCINT;
}
```

3.3.7 *Type check* function of type constructor `ccint`

```
int CheckInt(ListExpr type, ListExpr *errorInfo)
{
  return (IsEqual(type, "ccint"));
}
```

3.3.8 Definition of type constructor `ccint`

Eventually a type constructor is created by defining an instance of class `TypeConstructor`. Constructor arguments are the type constructor's name and the four functions previously defined.

```
TypeConstructor ccInt("ccint", OutInt, InInt, CreateInt, DeleteInt,
                      CastInt, CheckInt);
```

3.4 Type constructor `ccreal`

The following type constructor, `ccreal`, is defined in the same way as `ccint` in section 3.3.

3.4.1 Main memory representation

Each instance of below defined class `CCREAL` will be the main memory representation of a value of type `ccreal`. It consists of a boolean flag, `defined`, and a real value, `value`. `defined` may be used to indicate whether an instance of class `CCREAL` represents a valid value or not. For instance, a division by zero might result in a `CCINT` object with `defined` set to `false`.

Throughout this tutorial, we don't use the flag but just maintain it in order to demonstrate how to handle complex objects.

```
class CCREAL : public Attribute
{
  BOOLEAN defined;
  float value;
public:
  CCREAL(){};
  CCREAL(float v, BOOLEAN d = 1){value = v; defined = d;};
  ~CCREAL(){};
  BOOLEAN GetDefined(){return defined;};
  float GetValue(){return value;};
  void Set(float v, BOOLEAN d = 1){value = v; defined = d;};
  CompResult Compare(const Attribute *arg);
  int Compare(Attribute *){return 0;}; //dummy
  int Sizeof(){ return sizeof(CCREAL); };
  ostream &Print(ostream &os){return os << value; };
};

Attribute::CompResult CCREAL::Compare(const Attribute *arg)
{
    cout << "Comparing two Reals\n";
    if (((CCREAL*)arg)->value < value) return less;
    else if (((CCREAL*)arg)->value > value) return greater;
    else return equal;
};
```

3.4.2 *Out*-function of type constructor `ccreal`

The *out*-function of each type constructors takes as inputs a type description and a pointer to a value of respective type and returns the same value, now represented in nested list format. The function `OutReal` defined in the following C++ code segment performs exactly this task for the type constructor `ccreal`.

```
ListExpr OutReal(ListExpr typeinfo, ADDRESS value)
{
  return RealAtom(((CCREAL*)value)->GetValue());
}
```

In this very simple example we did not use the type description at all, because `ccreal` is a *constant* type constructor, i.e. type constructor `ccreal` can only construct one type: `ccreal`. We will need the type description, however, in the case of more powerful type constructors, e.g. to be able to compute the nested list representation of a tuple value we must know the types of the respective attribute values.

Note that we don't represent the `defined`-part of the `ccreal`-value passed as parameter due to the fact that we maintain it just for demonstration reasons.

3.4.3 *In*-function of type constructor `ccreal`

Complementary to the *out*-function, an *in*-function has to be defined for each type constructor, returning a value's main memory representation. It is calculated by taking the value's nested list representation and its type description as input parameters.

Again, the actual implementation of the *In*-function for type constructor `ccreal` does not use the type information in this example due to simplicity of `ccreal`.

The last three input parameters will be used to return error information.

```
ADDRESS InReal(ListExpr typeInfo, ListExpr value,
          int errorPos, ListExpr *errorInfo, BOOLEAN *correct)
{
  if (IsAtom(value) && AtomType(value) == RealType)
  {
    *correct = 1;
    return new CCREAL(RealValue(value));
  }
  else
  {
    *correct = 0;
    return 0;
  }
}
```

3.4.4 *Create*-function of type constructor `ccreal`

We also have to provide a *create*-function for each type constructor. This function may be used to allocate memory sufficient for keeping one instance of the respective type. Such a function accepts an integer value as argument which may be used to indicate the number of elements of the actual instance if it is a *collection* type, for instance, a list, a set, or a relation.

For type constructor `ccreal`, we call this function `CreateReal`. It is used to allocate memory sufficient for keeping one instance of CCREAL. The `Size`-parameter is not evaluated.

```
ADDRESS CreateReal(int Size)
{
  return new CCREAL(0,0);
}
```

3.4.5 *Destroy*-function of type constructor `ccreal`

A type constructor's *destroy*-function is used by the query processor in order to deallocate memory occupied by instances of Secondo objects. They may have been created in two ways:

- as return values of operator calls like those defined in section 4
- by calling a type constructor's `create`-function

The corresponding function of type constructor `ccreal` is called `DeleteReal`.

```
void DeleteReal(ADDRESS *w)
{
  delete *w;
  *w = 0;
}
```

3.4.6 *Cast*-function of type constructor `ccreal`

```
Attribute *CastReal(void *addr)
{
  return new (addr) CCREAL;
}
```

3.4.7 *Type check* function of type constructor `ccreal`

```
int CheckReal(ListExpr type, ListExpr *errorInfo)
{
  return (IsEqual(type, "ccreal"));
}
```

3.4.8 Definition of type constructor `ccreal`

Eventually a type constructor is created by defining an instance of class `TypeConstructor`. Constructor arguments are the type constructor's name and the four functions previously defined.

```
TypeConstructor ccReal("ccreal", OutReal, InReal, CreateReal, DeleteReal,
CastReal, CheckReal);
```

4 Operators

4.1 Overview

In contrast to *objects*, all properties of every *operator* are well-known at compile-time. This gives rise to representing operators as instances of a predefined class `Operator`. Thus an implementation of an algebra with n operators contains n definitions of instances of class `Operator`.

From a technical point of view, definition of operators is similar to definition of type constructors: An operator is defined by creating an instance of class `Operator`. Before this instance definition can take place we have to define the following functions for each operator:

- One or more *value mapping* functions. These functions take values of predefined types as input parameters and return the result value. Hence these functions do the "real work" we expect an operator to do. An algebra implementor has to define more than one value mapping function for a single operator if the operator is overloaded.

  The interface of such a function is independent of the type of operation. Stream operators and parameter function calls are handled by calling the functions `request`, `open`, `close`, `cancel`, and `received` of the module `QueryProcessor` [GFB+97].

- Four additional functions are needed for processing and optimizing queries in which the actual operator is involved:

  Selection function. This function is given a list of argument types. Based upon these it returns an index of the operator's array of value mapping functions. This is used to determine the appropriate value mapping function for an overloaded operator.

  Type Mapping function. This function maps a list of input types to an output type. Used for type checking and mapping.

  Model mapping function. (optional; only existing if the type has a model). Computes a result model from the argument models.

  Cost mapping function. Estimates the costs for evaluating the operation (number of CPUs, number of swapped pages) by means of the argument models. If there are no models, it returns some constant cost.

Definition of operators is done in a way similar to definition of type constructors: an instance of predefined class `Operator` is defined. But compared to type constructors, operator definition is a bit trickier due to the variable number of value mapping functions which an operator might consist of. We distinguish two cases:

- The operator is *non-overloaded*. In this simple case we only have to deal with one value mapping function and four additional functions. We simply pass all these functions as constructor arguments in the definition of an operator instance, as is demonstrated for operators `intfeed`, `intcount`, `intfold`, and `intshow` in sections 4.7 to 4.10, respectively.

- The operator is *overloaded*. In this case we first define an array containing all value mapping functions. After that we pass the number of value mapping functions, the array of value mapping functions, and the four additional functions as constructor arguments in the definition of an operator instance, as is demonstrated for operators `cc+`, `cc-`, `cc*`, and `cc:` in sections 4.3 to 4.6, respectively.

4.2 Operator implementation auxiliaries

4.2.1 Type investigation

Within this algebra module, we have to deal with values of different types:

- Types of this algebra: `ccint` and `ccreal`
- Nested list equivalents to `ccint` and `ccreal`, in the following called `nlint` and `nlreal`, respectively
- Predefined Secondo types: `stream` for streams and `map` for function abstractions.

Later on we will examine nested list type descriptions. In particular, we are going to check whether they describe one of the four types just introduced. In order to simplify dealing with list expressions describing these types, we declare an enumeration, `CcType`, containing all used types, and a function, `TypeOfSymbol`, taking a nested list as argument and returning the corresponding `CcType` type name, or the special type identifier `ccerror` in case of error.

```
enum CcType {ccint, ccreal, nlint, nlreal, stream, map, ccerror};

CcType TypeOfSymbol(ListExpr symbol)
{
  STRING s;
  if (AtomType(symbol) == SymbolType)
  {
    SymbolValue(symbol, s);
    if (!strcmp(s, "ccint")) return ccint;
    if (!strcmp(s, "ccreal")) return ccreal;
    if (!strcmp(s, "int")) return nlint;
    if (!strcmp(s, "real")) return nlreal;
    if (!strcmp(s, "stream")) return stream;
    if (!strcmp(s, "map")) return map;
  }
  return ccerror;
}
```

4.2.2 Selection function for non-overloaded operators

For non-overloaded operators, the set of value mapping functions consists of exactly one element. Consequently, the selection function of such an operator always returns 0 as value mapping function index.[2]

```
int SelectOnly(ListExpr args)
{
  return 0;
}
```

4.2.3 Common functions of mathematical operators

4.2.3.1 Type mapping function A type mapping function takes a nested list as argument. Its contents are type descriptions of an operator's input parameters. A nested list describing the output type of the operator is returned.

[2]Note that the query processor doesn't require a selection function to check the correctness of input parameter types. This task is done by an operator's type mapping function, which is guaranteed to be called *before* the corresponding selection function.

In our case, we define one type mapping function, `CcMathTypeMap`, actually sufficient for the three operators `cc+`, `cc-`, and `cc*`, since the same rules for input and output types apply to all of them: The number of input parameters must be exactly 2. If both types are `ccint`, the return value is also of type `ccint`. If one input parameter type is `ccreal` and the other one is `ccint`, `ccreal` is returned as well as if both input types are `ccreal`. Otherwise a type error occured.

```
ListExpr CcMathTypeMap(ListExpr args)
{
  ListExpr arg1, arg2;
  if (ListLength(args) == 2)
  {
    arg1 = First(args);
    arg2 = Second(args);
    if (TypeOfSymbol(arg1) == ccint && TypeOfSymbol(arg2) == ccint)
      return SymbolAtom("ccint");
    if (TypeOfSymbol(arg1) == ccint && TypeOfSymbol(arg2) == ccreal)
      return SymbolAtom("ccreal");
    if (TypeOfSymbol(arg1) == ccreal && TypeOfSymbol(arg2) == ccint)
      return SymbolAtom("ccreal");
    if (TypeOfSymbol(arg1) == ccreal && TypeOfSymbol(arg2) == ccreal)
      return SymbolAtom("ccreal");
  }
  return SymbolAtom("typeerror");
}
```

Notice that the symbol `typeerror` is not chosen just by chance, but is recognized by the query processor.

4.2.3.2 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.

Again we use a single function, `CCmathSelect`, for operators `cc+`, `cc-`, and `cc*`, since they accept the same combinations of input parameters.

```
int CcMathSelect(ListExpr args)
{
  ListExpr arg1 = First(args);
  ListExpr arg2 = 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;
}
```

4.3 Operator cc+

4.3.1 Value mapping functions of operator cc+

A value mapping function implements an operator's main functionality: it takes input arguments and computes the result. Each operator consists of at least one value mapping function. In the case of overloaded operators — like in this example — there are several value mapping functions, one for each possible combination of input parameter types. We have to provide four functions for operator cc+, since each of it accepts four input parameter type combinations: ccint ×ccint, ccint × ccreal, ccreal × ccint, and ccreal × ccreal.

The last two input parameters and the return value are only used by stream operators.

```
int CcPlus_ii(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCINT(((CCINT*)args[0])->GetValue() +
                      ((CCINT*)args[1])->GetValue());
  return 0;
}

int CcPlus_ir(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCINT*)args[0])->GetValue() +
                       ((CCREAL*)args[1])->GetValue());
  return 0;
}

int CcPlus_ri(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCREAL*)args[0])->GetValue() +
                       ((CCINT*)args[1])->GetValue());
  return 0;
}

int CcPlus_rr(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCREAL*)args[0])->GetValue() +
                       ((CCREAL*)args[1])->GetValue());
  return 0;
}
```

4.3.2 Definition of operator cc+

Operator cc+ is overloaded. Hence we first have to define an array of value mapping functions, ccplusmap, as explained in section 4.1.

```
ValueMapping ccplusmap[] = {CcPlus_ii, CcPlus_ir, CcPlus_ri, CcPlus_rr};
```

Now we can define operator `ccplus` by passing the appropriate parameters as constructor arguments.

```
Operator ccplus("cc+", 4, ccplusmap, CcMathSelect, CcMathTypeMap);
```

4.4 Operator `cc-`

4.4.1 Value mapping functions of operator `cc-`

Value mapping functions of operator `cc-` are defined in a way similar to value mapping functions of operator `cc+` in section 4.3.1.

Again, we have to define four different functions for all input parameter type combinations: `ccint` ×`ccint`, `ccint` × `ccreal`, `ccreal` × `ccint`, and `ccreal` × `ccreal`.

The last two input parameters and the return value are only used by stream operators.

```
int CcMinus_ii(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCINT(((CCINT*)args[0])->GetValue() -
                      ((CCINT*)args[1])->GetValue());
  return 0;
}

int CcMinus_ir(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCINT*)args[0])->GetValue() -
                       ((CCREAL*)args[1])->GetValue());
  return 0;
}

int CcMinus_ri(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCREAL*)args[0])->GetValue() -
                       ((CCINT*)args[1])->GetValue());
  return 0;
}

int CcMinus_rr(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCREAL*)args[0])->GetValue() -
                       ((CCREAL*)args[1])->GetValue());
  return 0;
}
```

4.4.2 Definition of operator `cc-`

Operator `cc-` is overloaded. Hence we first have to define an array of value mapping functions, `ccminusmap`, as explained in section 4.1.

```
ValueMapping ccminusmap[] = {CcMinus_ii, CcMinus_ir, CcMinus_ri, CcMinus_rr};
```

Now we can define operator `ccminus` by passing the appropriate parameters as constructor arguments.

```
Operator ccminus("cc-", 4, ccminusmap, CcMathSelect, CcMathTypeMap);
```

4.5 Operator `cc*`

4.5.1 Value mapping functions of operator `cc*`

Value mapping functions of operator `cc*` are defined in a way similar to value mapping functions of operator `cc+` in section 4.3.1.

Again, we have to define four different functions for all input parameter type combinations: `ccint` ×`ccint`, `ccint` × `ccreal`, `ccreal` × `ccint`, and `ccreal` × `ccreal`.

The last three input parameters and the return value are only used by stream operators.

```
int CcProduct_ii(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCINT(((CCINT*)args[0])->GetValue() *
                      ((CCINT*)args[1])->GetValue());
  return 0;
}

int CcProduct_ir(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCINT*)args[0])->GetValue() *
                       ((CCREAL*)args[1])->GetValue());
  return 0;
}

int CcProduct_ri(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCREAL*)args[0])->GetValue() *
                       ((CCINT*)args[1])->GetValue());
  return 0;
}

int CcProduct_rr(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(((CCREAL*)args[0])->GetValue() *
                       ((CCREAL*)args[1])->GetValue());
  return 0;
}
```

4.5.2 Definition of operator `cc*`

Operator `cc*` is overloaded. Hence we first have to define an array of value mapping functions, `ccproductmap`, as explained in section 4.1.

```
ValueMapping ccproductmap[] = {CcProduct_ii, CcProduct_ir, CcProduct_ri,
                               CcProduct_rr};
```

Now we can define operator `ccproduct` by passing the appropriate parameters as constructor arguments.

```
Operator ccproduct("cc*", 4, ccproductmap, CcMathSelect, CcMathTypeMap);
```

4.6 Preliminary: cast operator `cc:`

4.6.1 Motivation

In order to be able to test our algebra by hacking in some command line input using *SecondoTTY* we must be able to produce values of type `ccint` and `ccreal`. Unfortunately, the actual release of Secondo only recognizes the nested list built-in types `int`, `real`, `boolean`, `symbol`, and `string` from command-line input. As an interim solution, we provide another operator, `cc:`, which takes a single `int` or `real` parameter as input and returns a `ccint` or `ccreal` value as output, respectively.

Using the actual release, a command line input for adding 5 and 7.6 thus will look as follows:

```
(query (cc+ (cc: 5) (cc: 7.6))
```

This will be replaced soon by the general possibility of giving a tuple (typename value) as part of a query It will be the task of the query processor to call the respective type constructor's In-function with value as input parameter. value will be allowed to be any appropriate nested list expression. Hence adding 5 and 7.6 could then — without having defined any cast operator — be expressed as

---

```
(query (cc+ (ccint 5) (ccreal 7.6))).
```

Unfortunately, the query processor is not yet able to handle such queries, thus we have to define type mapping, selection, and value mapping functions of operator `cc:`.

4.6.2 Type mapping function of operator `cc:`

Operator `cc:` maps a value of type `nlint` or `nlreal` to `ccint` or `ccreal`, respectively.

```
ListExpr CcCreateTypeMap(ListExpr args)
{
  ListExpr arg;
  if (ListLength(args) == 1)
  {
    arg = First(args);
    if (TypeOfSymbol(arg) == nlreal)
      return SymbolAtom("ccreal");
    if (TypeOfSymbol(arg) == nlint)
      return SymbolAtom("ccint");
  }
  return SymbolAtom("typeerror");
}
```

4.6.3 Selection function of operator `cc:`

There are two value mapping functions for operator `cc:`, processing `nlint`- or `nlreal` argument types.

```
int CcCreateSelect(ListExpr args)
{
  ListExpr arg = First(args);

  if (TypeOfSymbol(arg) == nlint) return 0; else return 1;
}
```

4.6.4 Value mapping functions of operator `cc:`

`CcCreate_i` takes as argument an `nlint` value and returns an instance of class `CCINT` representing the same integer value.

```
int CcCreate_i(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCINT(*(int *)&args[0]);
  return 0;
}
```

`CcCreate_r` takes as argument an `nlreal` value and returns an instance of class `CCREAL` representing the same real value.

```
int CcCreate_r(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  *result = new CCREAL(*(float *)&args[0]);
  return 0;
}
```

4.6.5 Definition of operator `cc:`

Operator `cc:` is overloaded. Hence we first have to define an array of value mapping functions, `cccreatemap`, as explained in section 4.1.

```
ValueMapping cccreatemap[] = {CcCreate_i, CcCreate_r};
```

Now we can define operator `cccreate` by passing the appropriate parameters as constructor arguments.

```
Operator cccreate("cc:", 2, cccreatemap, CcCreateSelect, CcCreateTypeMap);
```

4.7 Operator `intfeed`

4.7.1 Type mapping function of operator `intfeed`

Operator `intfeed` accepts three parameters of type `ccint` and returns a stream of `ccint` values.

```
ListExpr IntFeedTypeMap(ListExpr args)
{
  ListExpr arg1, arg2, arg3;
  if(ListLength(args) == 3)
  {
    arg1 = First(args);
    arg2 = Second(args);
    arg3 = Third(args);
    if (TypeOfSymbol(arg1) == ccint &&
        TypeOfSymbol(arg2) == ccint &&
        TypeOfSymbol(arg3) == ccint)
      return TwoElemList(SymbolAtom("stream"), SymbolAtom("ccint"));
  }
  return SymbolAtom("typeerror");
}
```

4.7.2 Value mapping function of operator `intfeed`

Operator `intfeed` generates a stream of increasingly ordered `ccint`-values. The contents of the result stream are determined by three arguments passed in the `OPEN`-call of operator `intfeed`: lower bound, upper bound, and incrementation amount. For instance, passing arguments (3, 10, 2) produces the result stream 3, 5, 7, 9.

```
int IntFeed(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  CCINT *limits;
  switch (message)
  {
```

Within the OPEN-part of this function, we simply save the three input arguments by using the parameter local, which is guaranteed to remain the same for all calls of Intfeed until a CLOSE-message is passed.

```
        case OPEN:
          limits = new CCINT[3];
          *local = limits;
          limits[0] = *(((CCINT**)args)[0]);
          limits[1] = *(((CCINT**)args)[1]);
          limits[2] = *(((CCINT**)args)[2]);
          return 0;
```

In case of a REQUEST, we restore the previously lower and upper bounds and incrementation amount. If the stream is empty (lower bound > upper bound), CANCEL is returned, otherwise the new stream element is computed, saved as new lower bound, and assigned to the output parameter result. After that YIELD can be returned.

```
        case REQUEST:
          limits = *(CCINT**)local;
          if (limits[0].GetValue() > limits[1].GetValue())
            return CANCEL;
          else
          {
            *result = new CCINT(limits[0]);
            limits[0].Set(limits[0].GetValue() + limits[2].GetValue());
            return YIELD;
          }
```

Calling IntFeed with the CLOSE-message causes deallocation of previously allocated local memory. After that, the stream doesn't exist any longer.

```
        case CLOSE:
          delete[] *local;
          return 0;
      }
    }
```

4.7.3 Definition of operator intfeed

Non-overloaded operators are defined by constructing a new instance of class Operator, passing all operator functions as constructor arguments.

```
Operator intfeed("intfeed", IntFeed, SelectOnly, IntFeedTypeMap);
```

4.8 Operator intcount

4.8.1 Type mapping function of operator intcount

Operator intcount accepts a stream of ccint values and returns a ccint value.

```
ListExpr IntCountTypeMap(ListExpr args)
{
  ListExpr arg;
  if (ListLength(args) == 1)
  {
    arg = First(args);
    if (ListLength(arg) == 2)
    {
      if (TypeOfSymbol(First(arg)) == stream &&
          TypeOfSymbol(Second(arg)) == ccint)
        return SymbolAtom("ccint");
    }
  }
  return SymbolAtom("typeerror");
}
```

4.8.2 Value mapping function of operator `intcount`

Operator `intcount` returns the number of elements of the `ccint` stream passed as input parameter.

```
int IntCount(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  int count = 0;
  void *actual;
  open(args[0]);
  request(args[0], &actual);
  while (received(args[0]))
  {
    request(args[0], &actual);
    count++;
  }
  *result = new CCINT (count);
  return 0;
}
```

4.8.3 Definition of operator `intcount`

Non-overloaded operators are defined by constructing a new instance of class `Operator`, passing all operator functions as constructor arguments.

```
Operator intcount("intcount", IntCount, SelectOnly, IntCountTypeMap);
```

4.9 Operator `intfold`

4.9.1 Type mapping function of operator `intfold`

Operator `intfold` accepts three arguments:

- a `ccint` stream.

- a function, taking two `ccint` parameters and returning a `ccint` result value
- a single `ccint` value.

A `ccint` value is returned.

```
ListExpr IntFoldTypeMap(ListExpr args)
{
  ListExpr arg1, arg2, arg3;
  if (ListLength(args) == 3)
  {
    arg1 = First(args);
    arg2 = Second(args);
    arg3 = Third(args);
    if (ListLength(arg1) == 2 &&
        TypeOfSymbol(First(arg1)) == stream &&
        TypeOfSymbol(Second(arg1)) == ccint &&
        ListLength(arg2) == 4 &&
        TypeOfSymbol(First(arg2)) == map &&
        TypeOfSymbol(Second(arg2)) == ccint &&
        TypeOfSymbol(Third(arg2)) == ccint &&
        TypeOfSymbol(Fourth(arg2)) == ccint &&
        TypeOfSymbol(arg3) == ccint)
      return SymbolAtom("ccint");
  }
  return SymbolAtom("typeerror");
}
```

4.9.2 Value mapping function of operator `intfold`

Function `IntFold` takes as arguments

- a `ccint stream` $i_1, i_2, \cdots, i_n$
- a function f : `ccint` $\times$ `ccint` $\rightarrow$ `ccint`
- a single `ccint` value i_0.

$$\text{The return value is} \begin{cases} i_0 & , \text{ if input stream is empty} \\ f(\cdots f(f(i_0, i_1), i_2), \cdots, i_n) & , \text{ otherwise} \end{cases}$$

```
int IntFold(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  ArgVectorPointer vector;
  void *actual;
  *result = new CCINT(*(((CCINT**)args)[2]));
  open(args[0]);
  request(args[0], &actual);
  while (received(args[0]))
```

```
  {
    vector = argument(args[1]);
    (*vector)[0] = *result;
    (*vector)[1] = actual;
    request(args[1], result);
    request(args[0], &actual);
  }
  return 0;
}
```

Notice the twofold meaning of `request` within `IntFold`:

- The statement `request(args[0], &actual)` requests the next element of the *stream* given in `args[0]`.
- The statement `request(args[1], result)` requests a call of the *function* in `args[1]` with the arguments in `(*vector)[0]` and `(*vector)[1]`.

4.9.3 Definition of operator `intfold`

Non-overloaded operators are defined by constructing a new instance of class `Operator`, passing all operator functions as constructor arguments.

```
Operator intfold("intfold", IntFold, SelectOnly, IntFoldTypeMap);
```

4.10 Operator `intshow`

4.10.1 Type mapping function of operator `intshow`

Operator `intshow` takes a `ccint` stream as argument and returns a stream of same type (actually, the same stream).

```
ListExpr IntShowTypeMap(ListExpr args)
{
  ListExpr arg;
  if (ListLength(args) == 1)
  {
    arg = First(args);
    if (ListLength(arg) == 2)
    {
      if (TypeOfSymbol(First(arg)) == stream &&
          TypeOfSymbol(Second(arg)) == ccint)
        return TwoElemList(SymbolAtom("stream"), SymbolAtom("ccint"));
    }
  }
  return SymbolAtom("typeerror");
}
```

4.10.2 Value mapping function of operator `intshow`

`IntShow` takes a `ccint` stream as argument, prints each of its elements to standard output, and returns the same stream by elementwise assigning an input stream element to the `result` output parameter within the `REQUEST`-part in the following implementation of `IntShow`.

```
int IntShow(ADDRESS *args, ADDRESS *result, int message, ADDRESS *local,
Supplier)
{
  void *actual;
  switch (message)
  {
    case OPEN:
      open(args[0]);
      return 0;
    case REQUEST:
      request(args[0], &actual);
      if (received(args[0]))
      {
        cout << ((CCINT*)actual)->GetValue() << "\n";
        *result = actual;
        return YIELD;
      }
      else
        return CANCEL;
    case CANCEL:
      close(args[0]);
      return 0;
  }
}
```

4.10.3 Definition of operator `intshow`

Non-overloaded operators are defined by constructing a new instance of class `Operator`, passing all operator functions as constructor arguments.

```
Operator intshow("intshow", IntShow, SelectOnly, IntShowTypeMap);
```

5 Joining all components: class `CcAlgebra`

The last step of implementing a new algebra module consists of integrating all type constructors (section 3) and operators (section 4) defined so far into one instance of class `Algebra`. Therefore, a new subclass `CcAlgebra` has to be derived from class `Algebra`. The only specialization with respect to class `Algebra` takes place within the constructor: all type constructors and operators are added to the actual algebra.

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

Adding all type constructors and operators to the new algebra:

```
    AddTypeConstructor(&ccInt);
    AddTypeConstructor(&ccReal);

    AddOperator(&ccplus);
    AddOperator(&ccminus);
    AddOperator(&ccproduct);
    AddOperator(&cccreate);
    AddOperator(&intfeed);
    AddOperator(&intfold);
    AddOperator(&intcount);
    AddOperator(&intshow);
  }
  ~CcAlgebra(){};
};
```

Now we are ready to complete this implementation of an Secondo algebra module by defining an instance of the new algebra class:

```
CcAlgebra ccalgebra;
```

As soon as this definition has been executed, all type constructors and operators implemented within this algebra module are registered in the Secondo catalog. The Secondo interface will process queries involving these type constructors and operators by calling the corresponding functions defined in this algebra module.

6 Query examples

Now we are ready to compile this algebra implementation and link the resulting object file with the Secondo library to a Secondo executable containing our example algebra.

6.1 Catalog investigation

- `(list operators)`

  Among others, the operators `cc+`, `cc-`, `cc*`, `cc:`, `intfeed`, `intshow`, `intcount`, and `intconsume` are listed.

- `(list type constructors)`

  `ccint` and `ccreal` are part of the output.

6.2 Simple mathematical queries

- `(query (cc+ (cc:4) (cc:5)))`

  Result value 9 is returned.

- `(query (cc* (cc+ (cc:4) (cc:5.7))(cc- (cc:2.5)(cc:4))))`

  Result value -14.55 is returned.

6.3 Stream queries

- `(query (intcount (intfeed (cc: 8)(cc: 24)(cc: 3))))`

  Result value 6 is returned, which is the number of elements of the stream 8, 11, 14, 17, 20, 23.

- `(query (intcount (intshow (intfeed (cc: 8) (cc: 24) (cc: 3)))))`

  Result value 6 is returned again. In additiaon to that, the stream elements 8, 11, 14, 17, 20, 23 are printed to standard output during processing the query.

- (query (intfold (intfeed (cc: 1)(cc: 100)(cc: 1)) (fun (x ccint)(y ccint) (cc+ x y)) (cc: 0)))

  $\sum_{1}^{100} = 5050$ is returned.

References

[BBG+88] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An extensible database management system. *Transactions on Software Engeneering*, 14(11):1711–1730, November 1988.

[CDF+94] M. J. Carey, D. J. Dewitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):383–394, June 1994.

[CDRS86] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelth International Conference on Very Large Data Bases*, pages 91–100, Kyoto, Japan, August 1986.

[GFB+97] Ralf Hartmut Güting, Claudia Freundorfer, Ludger Becker, Stefan Dieker, and Holger Schenk. Secondo/QP: Implementation of a generic query processor. Informatik Bericht 215, FernUniversität - Gesamthochschule in Hagen, February 1997.

[Gra94] Goetz Graefe. Volcano—an extensible and parallel query evaluation system. *Transactions on Knowledge and Data Engineering*, 6(1):120–135, February 1994.

[Güt93] Ralf Hartmut Güting. Second-order signature: A tool for specifying data models, query processing, and optimization. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 277–286, Washington, D.C., 26–28 May 1993.

[Güt95] Ralf Hartmut Güting. Integrating programs and documentation. Informatik Bericht 182, FernUniversität - Gesamthochschule in Hagen, May 1995.

[HS90] Haas, L., Chang, W., Lohman, G., McPherson, M., Wilms, P., Lapis, G., Lindsay, B., Pirahesh, H., Carey, M. and E. Shekita. Starburst mid-flight: As the dust clears. *Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[SLR97] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The case for enhanced abstract data types. In *Proceedings of the 23rd VLDB Conference*, pages 66–5, Athens, Greece, 1997.