An Object Model Correspondence for *Aldor* and C++

Marc Gaëtano & Stephen M. Watt
INRIA – Projet Safir
2004, route des Lucioles, BP 93
06902 Sophia Antipolis Cedex - France
{marc.gaetano,stephen.watt}@sophia.inria.fr

Deliverable 2.2.1

1 Introduction

One of the major emphasis of the FRISCO project is to provide software tools for the linkage of FRISCO components with existing, well-understood technologies actually used in industrial settings. While much of the new FRISCO library development will occur in *Aldor*, the natural interoperation of *Aldor* and C++ objects becomes a central goal the project wants to achieve. The language C++ is widely used for applications programs in the industrial world and most of the background material of the project (the PoSSo library) is written in C++. Therefore, a well-defined semantic correspondence between the *Aldor* and the C++ computational models is needed in order to provide an interface suitable for direct end use. The first benefit of this work will be to provide natural interoperation of the C++ PoSSo Library and *Aldor* FRISCO Library elements.

This report provides a semantic correspondence between the object models of C++ and *Aldor*. This correspondence will be used for two purposes:

- as the basis for manual interface development between specific C++ and *Aldor* programs,
- as the starting point for the implementation of internal *Aldor* compiler support for a C++ interface (Frisco deliverable D2.2.2).

There are two levels at which the object model correspondence must be discussed:

- low level: object data layout and compatibility
- high level: object hierarchy and inheritance

This report addresses both these questions.

The C++ model of objects has a single-level class-based inheritance hierarchy, and single object dynamic dispatch. This model has proven not sufficiently rich to write statically type-safe algebraic software. The *Aldor* model of objects has a two-level, category-based, inheritance hierarchy, and dynamic dispatch based on dependent types. This two-level model is particularly suitable to the implementation of algebraic software. Domains and categories look in some way similar to classes and abstract classes. However, there are major differences between the two concepts. For example, *Aldor* functions match both on the types of their arguments and on the types of the results while C++ functions ignore result type when trying to match function signature. *Aldor* reaches an higher abstraction level than C++. For instance, every *Aldor* domain can provide a generator abstracting the traversal of its element. These iterators can iterate over structures in a more elegant and abstract fashion than in C++ where the programmer needs to define a friend iterator class.

The object model of C++ is based on the class concept. A class is a user-defined data type. A class specifies the type of class members. This type defines the representation of a variable (an object of the class). The class specifies also the set of operations (the functions) that manipulates these objects and the access the user has on these members. The object model for *Aldor* is based on the category/domain concept. A category is an abstract data type and describes a collection of exports, namely a (family of) type and the functions acting on this (family of) type. A domain belongs to a category and gives an implementation of the corresponding abstract type.

A simple mapping rule between C++ class and *Aldor* category and domain is given by the following example:

```
// A C++ class                    --- The corresponding Aldor Category
Class A : B,C {                   define A__Cat : Category ==
public:                              Join (B__Cat, C__Cat) with {
    f: (A, A) -> A                     f: % -> (%,%) -> %
}                                    }
```

The correspondence between a class and a category follows three simple rules :

- a C++ class (`A`) corresponds to an *Aldor* category (`A_Cat`)
- class multiple inheritance in C++ is handled by `Joining` the corresponding *Aldor* categories
- a public C++ function corresponds to an exported *Aldor* function with as an extra argument the actual domain (the type) the function is exported from.

In the example above, the C++ class `A` maps to the *Aldor* category `A_Cat`. The class `A` inherits from class `B` and `C`. Therefore, the corresponding *Aldor* category `A_Cat` has the union of all the exports of the `B_Cat` and `C_Cat` categories. The public function `f` with two arguments exported by the class `A` corresponds to the *Aldor* function `f` with three arguments exported by (any domain of) category `A_Cat`. The extra argument is the domain the function is exported from.

Functions and data structures may be shared between programs written in *Aldor* and other languages. An actual example of such interoperability is given with the C programming language. A C function can be called from an *Aldor* program and vice-versa. Moreover, there is a correspondence between the way data values are represented in *Aldor* and the way they are in C. It is foreseen that this capability will be extended to C++. This requires modifications to the *Aldor* compiler and will be investigated at a later stage.

2 Summary of the *Aldor* Object Model

The *Aldor* object model is a two-level model based on category and domain. A category is used to specify information about domains. A domain is an environment providing a collection of exported constants like types, functions and variables. A domain can be viewed as an *abstract data type* which defines a distinguished type and a collection of related exports. A category is used to place restrictions on the sort of domains it is prepared to handle and to make promises about the domains which it may return. It can be viewed as a *type* for these domains. In a category object, the restrictions and promises are expressed in terms of collections of exports which the domains typed by this category will be required to provide. Conversely, a domain may belong to any number of categories so long as it has the necessary exports.

The primitive for forming categories in *Aldor* is the "`with`" expression:

```
define StackCategory(S: Type): Category == with {

        new:    () -> %;       ++ creates an empty stack
        push!:  (S, %) -> %;   ++ destructively push an element into the stack
        pop!:   % -> S;        ++ destructively pop the top of the stack
        top:    % -> S;        ++ returns the top of the stack
        empty?: % -> Boolean;  ++ returns true if the stack is empty
}
```

The category `Category` is an *Aldor* built-in type which serves as the root type for all categories. The `StackCategory` is an abstract data type for stack objects. It has one parameter of type `Type` (any type), the type of the objects stored in a stack. A domain of type `StackCategory`, (i.e. an actual stack) is denoted by the identifier `%` and must export at least the functions specified in the category. A category can extend another category in a simple inheritance scheme:

```
define AbelianGroup: Category == AbelianMonoid with {

        -: % -> %;              ++ Negation.
        -: (%, %) -> %;         ++ Subtraction.

        default (a: %) - (b: %): % == a + (-b);
}
```

In this example, `AbelianGroup` inherits from `AbelianMonoid` and exports two additional functions named "`-`". Thus, function names can be overloaded in *Aldor*. The `default` construct gives an implementation for the subtraction function using only the information supplied in the category. A domain of type `AbelianGroup` is free to over-ride the implementation of "`-`" or to use the default. Multiple inheritance is achieved by the `Join` function on categories:

```
define OrderedFinite: Category == Join(Ordered, Finite) with {
        min: %;                 ++ Minimum value belonging to the type.
        max: %;                 ++ Maximum value belonging to the type.
}
```

The category `OrderedFinite` includes all the exports from both `Ordered` and `Finite` plus the two constants `min` and `max`.

A domain belongs to a category and defines a distinguished type of that category together with a collection of related exports. Unlike categories which only specify information, a domain describes how data values belonging to the distinguished type are encoded and exports callable functions acting on those values. The

representation type of the elements of a domain remains hidden from outside the domain and any domain can serve as representation type. The representation of a domain is typically a more primitive domain which is chosen to achieve a certain level of efficiency in the operations provided by the domain. These operations can be arranged so that a client programs which uses a domain need not depend on the representation of the domain. Thus, the representation of the domain provides the private view of its elements. On the public part, values from the domain are viewed through the exported functions which are acting on them. The domain `Stack` belongs to the `StackCategory`:

```
Stack(S: Type): StackCategory(S) == add {

        Rep ==> Record(l: List(S));
        import from List(S), Rep;

        new(): % == per [nil];
        push!(x: S, s: %): % == {
           rep(s).l := cons(x,rep(s).l);
           s};
        pop!(s: %): S == {
           x := first(rep(s).l);
           rep(s).l := rest(rep(s).l);
           x};
        top(s: %): S == first(rep(s).l);
        empty?(s: %): Boolean == empty?(rep(s).l);
}
```

The `Stack` domain is parameterized by another domain (the formal parameter `S`) of type `Type` which indicates that the actual parameter could be of any type. The `Stack` domain itself could serve as actual parameter and be used everywhere a type of category `StackCategory` is required. A domain may inherit the implementation of many of its operations from another domain:

```
IntegerMod(n: Integer): ModularIntegerNumberSystem(Integer)
== ModularIntegerNumberRep(Integer)(n) add {

      Rep == Integer;

      coerce(i: SI): %        == per(i::Integer mod n);
      coerce(i: Integer): %   == per(i mod n);
      (x: %) * (y: %): %      == per((rep x * rep y) mod n);
      (x: %) ^ (k: Integer): %== power(1, x, k)$BinaryPowering(%,*,Integer);
      (x: %) ^ (k: SI): %     == power(1, x, k)$BinaryPowering(%,*,SI);
      (x: %) / (y: %): %      == x * inv y;
}
```

The domain `IntegerMod` inherits from `ModularIntegerNumberRep`. Consequently it inherits the operations exported by this domain (called its parent) and so need not implement them explicitly in the `add` expression.

3 Summary of the C++ Object Model

The object model of C++ is based on the class concept. A class is a user-defined data type. Classes in C++ are similar to structures in C, except that the visibility of the class members can be specified explicitly. Class members can be variables and functions. The inheritance model of C++ supports dynamic binding through virtual functions and multiple inheritance. C++ has complicated mechanisms for controlling the visibility of class members. They can be declared as either `public`, `protected` or `private`. Only `public` members are available outside the class. However, if a class `B` inherits items from a class `A`, the protected items of class `A` can be used by the derived class `B`.

A C++ class is a description of a kind of object and it contains a description of the internal variables of the class, and the operations that can act on the objects of the class. For example, the class `Stack` can be defined as:

```
class Stack {
  protected:
    int buffer[50]; // the stack
    int index;      // the top of the stack
  public:
    void push(int i); // push an integer into the stack
    int  top(void);   // returns the top of the stack
    int  pop(void);   // pop the stack and returns the top
};
```

Explicit reference to the `index` member such as `index++` is not allowed. The actual code implementing the operations specified in the class declaration is usually separated from the class definition itself. Using a scope resolution operator, it is possible to give the code implementing the operations of the class `Stack`:

```
int stack::pop(void)
{
    return buffer[--index];
}

void stack::push(int i)
{
    buffer[index++] = i;
}
```

Inheritance allows to define new classes by extending existing classes, using the following notation:

```
class NewClass: ExistingClass {
    ...
};
```

The class `NewClass` is called the derived class and the `ExistingClass` is called the base class. If the base class is derived from yet another class, the members of this class are also inherited. It is often the case that classes in a program are organized in a hierarchy and a class inherits from all its ancestors. If a class is viewed as a type, a derived class can be consider as a subtype of the base type (class). Therefore, it is possible to assign variables of a derived class to variables of its base class. A derived class may have more than one base class and this situation is called multiple inheritance. The syntax for expressing multiple inheritance is simply an extension of that for single inheritance:

```
class C: public A, public B {
    ...
};
```

The derived class `C` inherits all of the member variables and functions of both classes `A` and `B` and is a subtype of both classes.

C++ allows both static and dynamic bindings. Dynamic binding of operations looks at the class of the object to which the operation is applied, and possibly at ancestors of this class. In this case, there is some execution time overhead since the run-time system must decide dynamically which operation to invoke. Dynamic binding is achieved in C++ by the use of virtual functions. A virtual function is declared in a base class or derived class and redefined in descendant classes. The set of classes and subclasses that define and redefine virtual function is viewed as a polymorphic cluster associated with the particular virtual function. Within the polymorphic cluster, a message is bound to a particular virtual member function at run-time. For example in:

```
class employee {
    char*  name;
    short  department;
    employee* next;
    static employee* list;
public:
    employee(char* n, int d);
    static void print_list()
    virtual void print() const;
};
```

the keyword `virtual` indicates that the function `print` is a virtual function that will be redefined in descendant classes of `employee`. Virtual functions work with simple and multiple inheritance in a way similar to other functions. That is, a call to a virtual function invokes the implementation provided by the whatever class is closest to the bottom of the object's inheritance DAG.

Object in C++ can be created through a declaration or with the `new` construct. For example, the following code:

```
SomeClass v;        // static allocation
SomeClass *v1, *v2; // pointer variables
v1 = &v;            // v1 points to object v
v2 = new SomeClass; // dynamic allocation
```

declares a new variable `v` containing an object of class `SomeClass` and two variables, `v1` and `v2`, that are pointers to `SomeClass` objects. The variable `v1` is set to point to object `v`, while `v2` will contain a reference to a newly created `SomeClass` object. Dynamic binding and polymorphism only work for pointers to objects.

4 Data Correspondence

We develop the C++-*Aldor* object correspondence by beginning at the lowest level, showing how individual data objects in C++ and *Aldor* are related.

Both C++ and *Aldor* have well-defined mechanisms for calling programs written in the C language. The simplest way to describe the data correspondence between C++ and *Aldor* is therefore in terms of C data types.

4.1 C++ and C

In theory, there need not be a strict correspondence even between two C compilers by different manufacturers for the same hardware platform. For example, it can be the case that two C compilers for a PC will have compatible meanings for `short` and `long`, but will use different sizes for `int`.

In practice, C compilers by different manufacturers can be made to behave compatibly through the use of command line flags to control the size, alignment, (and sometimes the representation) of the various primitive types.

The C++ programming language is essentially an upward extension of C, so all the types of C have their well defined meaning in C++, and in practice are represented compatibly. This *de facto* compatibility between C compilers' data layout is exploited by C++ compilers for a straightforward linkage to C programs using `extern "C"`, as in the declaration

```
extern "C" {
        extern char *strcpy(char *);
        extern int   fprintf(FILE *, char *, ...);
}
```

The treatment of types specific to C++ (i.e. not in C), however, is quite implementation dependent. This becomes relevant when dealing with the representation of objects of complex classes. Here, the issues are:

- the order and relation between the various collections of fields under multiple inheritance, and
- the location of virtual function tables.

4.2 *Aldor* and C

Note that the NAG *Aldor* compiler can generate code for two quite different environments: for embedding in closed Lisp systems, and for linking with C, Fortran and other open programming languages. The correspondence we describe here is applicable in the second case.

In *Aldor*, while there is multiple inheritance of interface and behavior, the inheritance of representation is extremely limited: Each domain in *Aldor* defines its own representation *ab initio*. The constructed domain may elect to use some of the exported operations of the representing type

```
NewDomain: InterfaceSpecification == RepresentingDomain add {
        ...
}
```

or it may elect that none of the behavior of the representing domain be inherited automatically:

```
NewDomain: InterfaceSpecification == add {
        Rep == RepresentingDomain;
        ...
}
```

Note that in either case, the representation is given by a single type and not a combination of ancestors. Therefore inheritance of representation is an either/or decision — there is no extension of representation and certainly no multiple inheritance of representation.

Secondly, in *Aldor* the polymorphism and dynamic dispatch associated with C++ virtual functions is provided through the more transparent mechanism of dependent types. This means that individual *Aldor* objects to not need to carry with them any sort of virtual function table. As a consequence *Aldor* data values can be mapped directly to C.

The functionality associated with the virtual function tables in C++ is provided by explicit type values in *Aldor*. Function closures and type values are represented as pointers to (undocumented) structures, so they may be passed as opaque structures to programs in other programming languages.

The precise rules of the correspondence are detailed in the *Aldor* users guide [?]. Here we give a brief summary:

Aldor's first order abstract machine (FOAM) defines a number of types which correspond to types on the target machine (in this case C on top of some operating system). The "`Machine`" package, described in [?], exports the types provided by the abstract machine. All *Aldor* values are represented internally as elements of one of these types. The complete listing and definition of the types is given in the FOAM reference guide.

Because many *Aldor* domains can be parameterized over different types, *Aldor* uses a pointer-sized object when passing domains. Thus, double precision floating point numbers (which are typically bigger than pointers) are "boxed", and a pointer to the box is passed, rather that the number itself. Types which are the same size or smaller than pointer-size are cast to the pointer type when used in a generic context and cast back as appropriate.

In order to make the underlying types available, *Aldor* provides the `Machine` package, which exports these types and operations on them. For example, the underlying representation type of `DoubleFloat` is `DFlo$Machine`. This type should be used when calling foreign functions, and the result coerced back to appropriate generic type at the *Aldor* level.

Records in *Aldor* are compatible with C `structs`. When calling C-defined functions that use records it is important to ensure that the elements of the *Aldor* record correspond to elements in the C structure. This implies that records intended for use in Foreign functions should use the underlying types, rather than the user-level types. The file "`$AXIOMXLROOT/samples/lib/libaxlX11`" provides an example.

The *Aldor* types correspond to C types given as `typedefs` in the file `$AXIOMXLROOT/include/foam_c.h`. The following table shows the correspondance between the types exported from the *Aldor* package `Machine` and C. It should be possible from this to understand which *Aldor* declaration will correspond to a declaration in C.

Aldor type	C typedef	Usual C type
`Nil$Machine`	`FiNil`	`Ptr`
`Word$Machine`	`FiWord`	`int`
`Arb$Machine`	`FiArb`	`long int`
`Ptr$Machine`	`FiPtr`	`Ptr`
`Bool$Machine`	`FiBool`	`char`
`Byte$Machine`	`FiByte`	`char`
`HInt$Machine`	`FiHInt`	`short`
`SInt$Machine`	`FiSInt`	`long`
`Char$Machine`	`FiChar`	`char`
`Arr$Machine`	`FiArr`	`Ptr`
`Rec$Machine`	`FiRec`	`Ptr`
`BInt$Machine`	`FiBInt`	`Ptr`
`SFlo$Machine`	`FiSFlo`	`float`
`DFlo$Machine`	`FiDFlo`	`double`
A `->` *B*	`FiClos`	`struct _FiClos *`

Here `Ptr` is defined as the type `char *` for compatibility with old C dialects, but could equally well be

```
    }
    fun(sin);
    fun(cos);
    fun(tan);
}
Test();
```

Aldor functions can be exported to the C foreign language in a similar way, using the `export to` construct:

```
Int  == SingleInteger;
export {
        fact:           Int -> Int;
        print:          Int -> ()   } to Foreign C;

fact (x: Int) : Int == if x = 0 then 1 else x * fact(x-1);
print (x: Int) : () == print<<x<<newline;
```

5 Class/Category Correspondence

In C++ classes specify both the sharing of interfaces and the sharing of representation. These aspects are treated separately in *Aldor*, with the specification of type interfaces given by Categories and the specification of type representation given by individual Domains.

For a C++ class library to be used from *Aldor*, a set of *Aldor* declarations have to be provided to make available information about

- the data layout for members of the individual classes
- the relationship and inheritance among classes in the library
- the global objects defined by the library.

We have already seen in Section ?? how to handle the data layout information. This section details how to capture the relationships and interitance properties of a class library. Once this is done, the declaration of global library objects is straightforward.

5.1 Base Classes

Suppose we have a C++ base class A, specifying member functions `fa1` and `fa2` as follows:

```
class A {
public:
    A fa1(int, int);
    int fa2(int);
};
```

Then, if a is a member of `class A` then it may be used as follows:

```
A aa = a.fa1(2,3);
```

That is, given a value a, one can apply its member function `fa1` to two integer parameters to produce a new value of type A.

The corresponding *Aldor* category definition is

```
define A__Cat: Category == with {
        fa1: % -> (SingleInteger, SingleInteger) -> %;
        fa2: % -> (SingleInteger) -> SingleInteger;
};
```

Note that all of the exports are derived from the public member functions of the C++ class, but are curried to take an additional first parameter, corresponding to the object from which the C++ method is selected. There is no need to provide exports corresponding to the private member functions.

Then, if `A__Dom` is a particular domain satisfying this category interface, and a is an element of `A__Dom`, we may use it as follows:

```
aa := fa1(a)(2,3);
```

That is, given a value a, one can apply the (curried) function `fa1` to two integer parameters to produce a new value of type `A__Dom`.

We note that it would be possible to develop a correspondence where the extra first parameter were just added to the argument list of each function. Then all of the exported C++ unary functions would become binary *Aldor* functions, etc. However, by using curried functions in the way we propose here, we have a strong visual parallel to the C++ code, and the late binding achieved through dependent functions (to model virtual functions) is more evident.

5.2 Derived Classes

In C++ new classes may be derived from existing classes. For example

```
class X: A, B {
public:
    // additional members, if desired.
}
```

In Aldor, inheritance is specified by including a Join of one or more categories in the new definition. For example:

```
define X__Cat: Category == Join(A__Cat, B__Cat) with {
    -- additional members, if desired.
}
```

5.3 Public and Private Derivation

There are two independent aspects in the relation between the derived (`X`) and the base classes (`A`, `B`) in the example above:

- Whether the inheritance is visible to the outside world, i.e. whether an instance of the derived class is known to be an instance of the base class.
- Under multiple inheritance, whether repeated inheritance from the same base class counts as distinct or shared behaviour.

The first issue is controlled in C++ by a declaration of the base class as either `private`, `protected` or `public`. E.g.

```
class X: private A, public B { ...}
```

In *Aldor* the inheritance relationships among categories are normally publicly visible. E.g. above, the inheritance relationship between `X__Cat` and `A__Cat`, and between `X__Cat` and `B__Cat`, is public.

Private inheritance can be achieved through an additional level of declaration. To reproduce the example above, with a private inheritance from `A` and public from `B`, the following would be used:

```
define A__Private: Category == with { ..... };

define A__Cat: Category == A__Private;

define X__Cat: Category == Join(A__Private, B__Cat) with { .... };
```

5.4 Shared and Repeated Inheritance

In C++ it is possible to specify whether the instances of the same base class appearing in an ancestry are to be kept distinct or to be shared. The normal behaviour is to keep the repeated base classes distinct. This is overridden to provide shared behaviour by the use of the "`virtual`" keyword.

In *Aldor*, the normal behaviour is for repeated inherited categories to be deemed the same. This is natural, since categorical inheritance specifies what operations are available on the type, and do not imply anything about the data representation. So, for example, in the following code the category `Y__Cat` inherits from `A__Cat` through two distinct paths, but the semantics of the base category `A__Cat` is the same.

```
define A__Cat: Category == { a: () -> SingleInteger };

define B1__Cat: Category == A with { b1: () -> %; };
define B2__Cat: Category == A with { b2: % -> (); };

define Y__Cat: Category == Join(B1, B2);
```

In the rare case where one really does wish to imply two distinct sets of behaviours corresponding to (logical) subsets of the data value, the way to do this is to provide separate "member" functions providing the necessary views:

```
define Y__Cat: Category == with {
     view1: % -> B1__Cat;
     view2: % -> B2__Cat;
}
```

Then these may be accessed as follows:

```
myfun(Y: Y__Cat, y: Y): () == {
    n1 := a()$view1(y);
    n2 := a()$view2(y);
    .....
}
```

6 Class/Domain Correspondence

So far, we have seen in Section ?? how to build a set of parallel delcarations to yield an *Aldor* category heirarchy mirroring a C++ class hierarchy.

To do any computation in *Aldor*, however, it is also necessary to specify some concrete *domains*. To finish the model of a class hierarchy, it is necessary to select those classes from which objects will be declared and used. For each of those C++ classes, it will be necessary to build a corresponding *Aldor* domain. This *Aldor* domain will serve as a wrapper, representing its values as pointers to the corresponding C++ objects (if they are extended objects), or as the values themselves (if they are pointers/handles).

6.1 The Exact Case

The simple case of C++ object use is when each object is known to belong to an exact class. That is, when the programs do not pass values belonging to derived classes to functions expecting objects of a base class.

```
class A { ... };
class B : A { ... };

A f(A a) { ... }

A a1;

B b1;

f(a1);  -- Exact.
f(b1);  -- Not exact.
```

In this case, all that is necessary is to declare *Aldor* values belonging to the corresponding domains, and to use them as normal *Aldor* objects. The domains will stand in the place of the C++ classes for object declarations in programs. The implementation of each domain will be very regular and could even be automatically generated.

6.2 The Derived Case

When the C++ library being modelled is designed to make use of late-binding via subclassing, then its use from *Aldor* is more complicated.

Subclassing and virtual functions combine in C++ to give a very powerful form of object oriented programming, where individual values can be viewed as carrying around all their own methods.

There are two mechanisms for providing similar function in *Aldor*. The first is explicit parametric polymorphism. An example would be

```
double(R: Ring, r: R): R == r + r;
```

Here the value `r` and its type `R` are passed as individual dependent parameters to the function `double`, and the addition applied to `r` is taken from `R`.

This might seem more complicated than the subclassed object polymorphism of C++, because of the requirement that `R` be passed explicitly. An important factor, however, is that in mathematical applications, it is quite common to have to combine different values in various steps of a computation. For example:

```
excess(R: Ring, a: R, b: R): R == a*b - b*a;
```

In C++, without specifying `R` explicitly, there is no guarantee that the two elements would be coming from the same algebraic structure and so run-time tests would be required to guarantee coherence between the operands of the multiplications and the subtraction.

Even though it is much less common than in C++, there are cases where one does want to explicitly use subclassed object polymorphism in *Aldor*. For this, the type constructor `Object` is provided,

```
Object: (C: Category)->  with {
        object:          (T: C, T) -> %;
        avail:           % -> (T: C, T);
}
```

This constructor builds objects in exactly the C++ sense, each carrying around a virtual function table (in this case modelled as the type argument to the constructor "`object`").

An automatic interface generator could, in principle, build interfaces to all of all of the classes of a C++ hierarchy using an `Object` representation. This would have a significant overhead, however, in the case when it was not needed, so a more judicious approach, using a direct pointer representation for classes used exactly, would seem to hold the most promise.

7 Examples from the PoSSo Library

Some examples of mapping using the principles detailed in the previous sections have been written for a subset of the PoSSo library. A few typical PoSSo types have been translated to the corresponding *Aldor* categories using the simple mapping rules described in section ??. For instance, the class PL_PP of the PoSSo library defines the power product type and its operations:

```
class PL_PP {

friend class PL_PPMonoid;

 ..... // more friend class

public:

  PL_Bool operator >  ( const PL_PP& pp );
  PL_Bool operator >= ( const PL_PP& pp );
  PL_Bool operator <  ( const PL_PP& pp );
  PL_Bool operator <= ( const PL_PP& pp );
  PL_Bool operator == ( const PL_PP& pp );
  PL_Bool operator != ( const PL_PP& pp );

  PL_PP& operator = ( const PL_PP& pp );
  PL_PP& operator * ( const PL_PP& pp );
  PL_PP& operator / ( const PL_PP& pp );
  PL_PP& operator *= ( const PL_PP& pp );
  PL_PP& operator /= ( const PL_PP& pp );

  PL_PP& mulvar( const int var, const int exp = 1 );
  PL_PP& dmulvar( const int var, const int exp = 1 );

  PL_PP& homogenize( const int var,
                       const Directives dir, const int target );
  PL_PP& dishomogenize( const int var );

  int operator [] ( const int i );

  int expsum();
  int weight();
  int sugar();
  int ecart();
  int userwgt();

  PL_Bool iszero();

private:

  static PL_PPMonoid* CurrentPPMonoid;
};
```

The corresponding category in *Aldor* exports the public methods of PL_PP as functions with an extra argument:

```
Directives ==> SI;

define PL_PP_Cat : Category == with {

  -- Member functions
  >: % -> % -> Bool;
  >=: % -> % -> Bool;
  <: % -> % -> Bool;
  <=: % -> % -> Bool;
  _=_=: % -> % -> Bool;
  _!_=: % -> % -> Bool;
  =: % -> % -> %;
  *: % -> % -> %;
  /: % -> % -> %;
  _*_=: % -> % -> %;
  _/_=: % -> % -> %;
  mulvar: % -> (SI, SI) -> %;
  dmulvar: % -> (SI, SI) -> %;
  homogenize: % -> (SI, Directives, SI) -> %;
  dishomogenize: % -> SI -> %;
  _[_]: % -> SI -> SI;
  expsum: % -> () -> SI;
  weight: % -> () -> SI;
  sugar: % -> () -> SI;
  ecart: % -> () -> SI;
  userwgt: % -> () -> SI;
  iszero: % -> () -> SI;
};
```

Besides this manual interface development between the C++ PoSSo library and the *Aldor* language, some work is in progress toward the implementation of internal *Aldor* compiler support for a C++ interface.

8 Conclusion

This report presents a preliminary work on *Aldor* and C++ interoperability. The mapping rule mechanism described shows how to define a simple correspondence between C++ classes and *Aldor* categories and domains. Work in progress towards modifying the *Aldor* compiler so that it is possible to call C++ functions from *Aldor* programs will provide further experience in programming language interoperability which can be usefully applied to related tasks (like 2.3 on Numerical Interoperability).

References

[W+94] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, Robert S. Sutor, *"AXIOM Library Compiler user Guide,"* The Numerical Algorithms Group Limited, 1994.