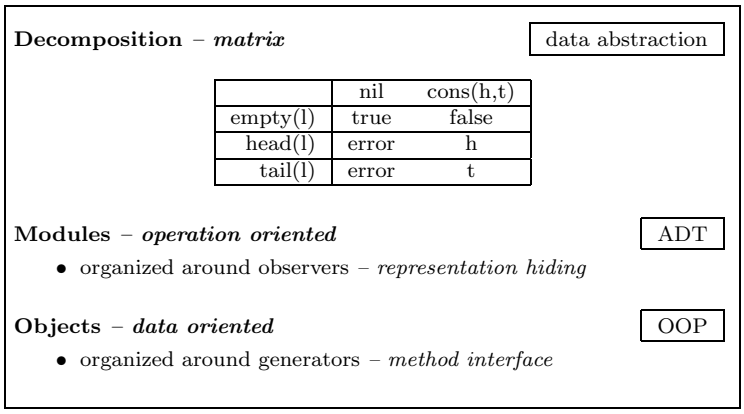
0.1 Decomposition – modules versus objects

Abstract data types allow the programmer to define a complex data structure and an associated collection of functions, operating on that structure, in a consistent way. Historically, the idea of data abstraction was originally not type-oriented but arose from a more pragmatic concern with information hiding and representation abstraction, see [Parnas72b]. The first realization of the idea of data abstraction was in the form of modules grouping a collection of functions and allowing the actual representation of the data structures underlying the values of the (abstract) type domain to be hidden, see also [Parnas72a].

In [Cook90], a comparison is made between the way in which abstract data types are realized traditionally (as modules) and the way abstract data types may be realized using object-oriented programming techniques. According to [Cook90], these approaches must be regarded as being orthogonal to one another and, being to some extent complementary, deserve to be integrated in a common framework.

After presenting an example highlighting the differences between the two approaches, we will further explore these differences and study the trade-offs with respect to possible extensions and reuse of code.



Slide 0-1: Decomposition and data abstraction

Recall that abstract data types may be completely characterized by a finite collection of generators and a number of observer functions that are defined with respect to each possible generator. Following this idea, we may approach the specification of a data abstraction by constructing a *matrix* listing the generators column-wise and the observers row-wise, which for each *observer/generator* pair specifies the value of the observer for that particular generator. Incidentally, the definition of such a matrix allows us to check in an easy way whether we have given a complete characterization of the data type. Above, an example is given of the specification of a *list*, with generators *nil* and *cons*, and observers *empty*, *head* and *tail*. (Note that we group the secondary producer *tail* with the observers.)

Now, the traditional way of realizing abstract data types as modules may be

characterized as *operation oriented*, in the sense that the module realization of the type is organized around the observers, resulting in a horizontal decomposition of the matrix.

On the other hand, an object-oriented approach may be characterized as *data oriented*, since the object realization of a type is based on specifying a method interface for each possible generator (sub)type, resulting in a vertical decomposition of the matrix. See slide **??**.

Note, however, that in practice, different generators need not necessarily correspond to different (sub)classes. Behavior may be subsumed in variables, as an object cannot change its class/type.

0.1.1 Abstract interfaces

When choosing for the module realization of the data abstraction *list* in C style, we are likely to have an abstract functional interface as specified in slide **??**.

Modules – ***a functional interface*** ADT *0-2*

```
typedef int element;
struct list;

extern list* nil();
extern list* cons(element e, list* l);
extern element head(list* l);
extern list* tail(list* l);
extern bool equal(list* l, list* m);
```

Slide 0-2: Modules – a functional interface

For convenience, the *list* has been restricted to contain integer elements only. However, at the expense of additional notation, we could also easily define a generic list by employing template functions as provided by C++. This is left as an exercise for the reader.

The interface of the abstract class *list* given in slide **??** has been defined generically by employing templates.

Note that the *equal* function in the ADT interface takes two arguments, whereas the *operator*`==` function in the OOP interface takes only one, since the other is implicitly provided by the object itself.

0.1.2 Representation and implementation

The realization of abstract data types as modules with functions requires additional means to hide the representation of the *list* type. In contrast, with an object-oriented approach, data hiding is effected by employing the encapsulation facilities of classes.

Objects – ***a method interface*** OOP *0-3*

```
template¡ class E ¿
class list {
public:
list() { }
virtual ∼list() { }
virtual bool empty() = 0;
virtual E head()  = 0;
virtual list¡E¿* tail() = 0;
virtual bool operator==(list¡E¿* m) = 0;
};
```

Slide 0-3: Objects – a method interface

Modules – representation hiding Modules provide a syntactic means to group related pieces of code and to hide particular aspects of that code. In slide **??** an example is given of the representation and the generator functions for a list of integers.

For implementing the *list* as a collection of functions (ADT style), we employ a *struct* with an explicit tag field, indicating whether the list corresponds to *nil* or a *cons*.

The functions corresponding with the generators create a new structure and initialize the tag field. In addition, the *cons* operator sets the *element* and *next* field of the structure to the arguments of *cons*.

The implementation of the *observers* is given in slide **??**.

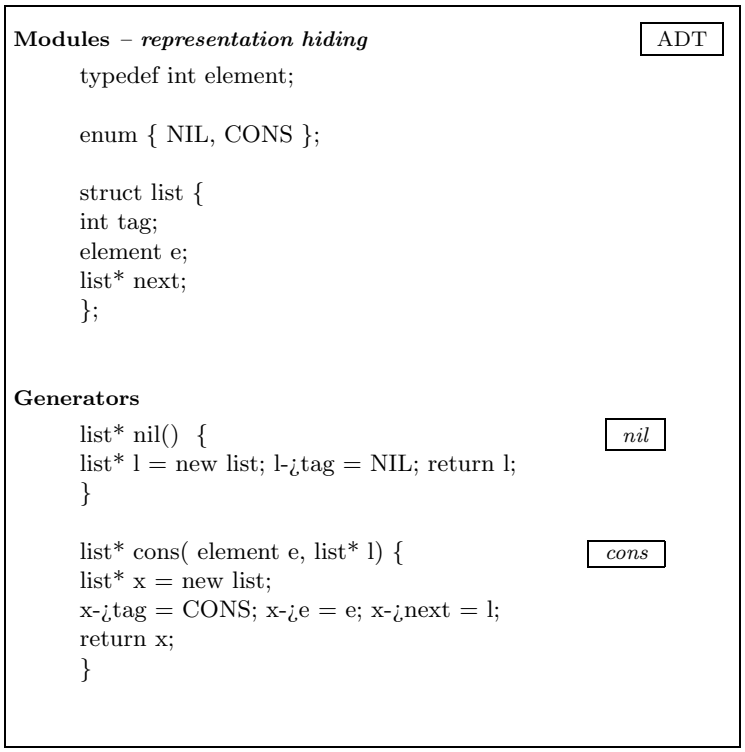
To determine whether the list is *empty* it suffices to check whether the tag of the list is equal to *NIL*. For both *head* and *tail* the pre-condition is that the list given as an argument is not empty. If the pre-condition holds, the appropriate field of the *list* structure is returned.

The equality operator, finally, performs an explicit switch on the tag field, stating for each case under what conditions the lists are equal.

Below, a program fragment is given that illustrates the use of the list.

```
list* r = cons(1,cons(2,nil()));
while (!empty(r)) {
        cout << head(r) << endl;
        r = tail(r);
        }
```

Note that both the generator functions *nil* and *cons* take care of creating a new *list* structure. Writing a function to destroy a list is left as an exercise for the reader.



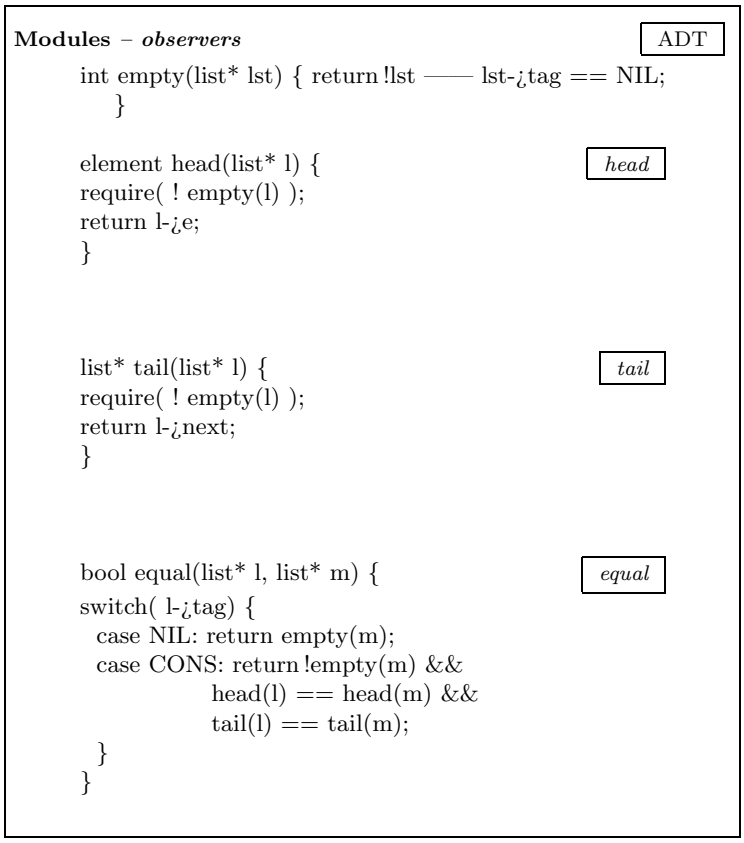
Slide 0-4: Data abstraction and modules

Objects – method interface The idea underlying an object-oriented decomposition of the specification matrix of an abstract type is to make a distinction between the (syntactic) subtypes of the data type (corresponding with its generators) and to specify for each subtype the value of all possible observer functions. (We speak of *syntactic* subtypes, following [Dahl92], since these subtypes correspond to the generators defining the value domain of the data type. See [Dahl92] for a more extensive treatment.)

In the object realization in slide **??**, each subtype element is defined as a class inheriting from the *list* class. For both generator types *nil* and *cons* the observer functions are defined in a straightforward way. Note that, in contrast to the ADT realization, the distinction between the various cases is implicit in the member function definitions of the generator classes.

As an example of using the *list* classes consider the program fragment below.

```
list<int>* r = new cons<int>(1, new cons<int>(2, new nil<int>));
while (! r->empty()) {
        cout << r->head() << endl;
        r = r->tail();
        }
delete r;
```



Slide 0-5: Modules – observers

For deleting a list we may employ the (virtual) destructor of *list*, which recursively destroys the tail of a list.

0.1.3 Adding new generators

Abstract data types were developed with correctness and security in mind, and not so much from a concern with extensibility and reuse. Nevertheless, it is interesting to compare the traditional approach of realizing abstract data types (employing modules) and the object-oriented approach (employing objects as generator subtypes) with regard to the ease with which a specification may be extended, either by adding new generators or by adding new observers.

Let us first look at what happens when we add a new generator to a data type, such as an interval list subtype, containing the integers in the interval between two given integers.

For the module realization of the list, adding an $interval(x, y)$ generator will

Method interface – *list* OOP *0-6*

```
template¡ class E ¿
class nil : public list¡ E ¿ {                    nil
public:
nil() {}
bool empty() { return 1; }
E head() { require( false ); return E(); }
list¡ E ¿* tail() { require( 0 ); return 0; }
bool operator==(list¡E¿* m) { return m-¿empty(); }
};


template¡ class E ¿
class cons : public list¡ E ¿ {                   cons
public:
cons(E e, list¡E¿* l) : _e(e), next(l) {}
~cons() { delete next; }
bool empty() { return 0; }
E head() { return _e; }
list¡E¿* tail() { return next; }
bool operator==(list¡E¿* m);
protected:
E _e;
list¡E¿* next;
};
```

Slide 0-6: Data abstraction and objects

result in an extension of the (hidden) representation types with an additional representation tag type *INTERVAL* and the definition of a suitable generator function.

To represent the *interval* list type, we employ a union to select between the *next* field, which is used by the *cons* generator, and the *z* field, which indicates the end of the interval.

Also, we need to modify the observer functions by adding an appropriate case for the new interval representation type, as pictured in slide **??**.

Clearly, unless special constructs are provided, the addition of a new generator case requires disrupting the code implementing the given data type manually, to extend the definition of the observers with the new case.

In contrast, not surprisingly, when we wish to add a new generator case to the object realization of the list, we do not need to disrupt the given code, but we may simply add the definition of the generator subtype as given in slide **??**.

Adding a new generator subtype corresponds to defining the realization for an

Adding new generators – ***representation*** ADT *0-7*

```
typedef int element;

enum { NIL, CONS, INTERVAL };

struct list {
int tag;
element e;
union { element z; list* next; };
};
```

Generator

```
list* interval( element x, element y ) {
        list* l = new list;
        if ( x ¡= y ) {
                l-¿tag = INTERVAL;
                l-¿e = x; l-¿z = y;
                }
        else l-¿tag = NIL;
        return l;
        }
```

Slide 0-7: Modules and generators

abstract interface class, which gives a method interface that its subclasses must respect.

Observe, however, that we cannot exploit the fact that a list is defined by an interval when testing equality, since we cannot inspect the type of the list as for the ADT implementation.

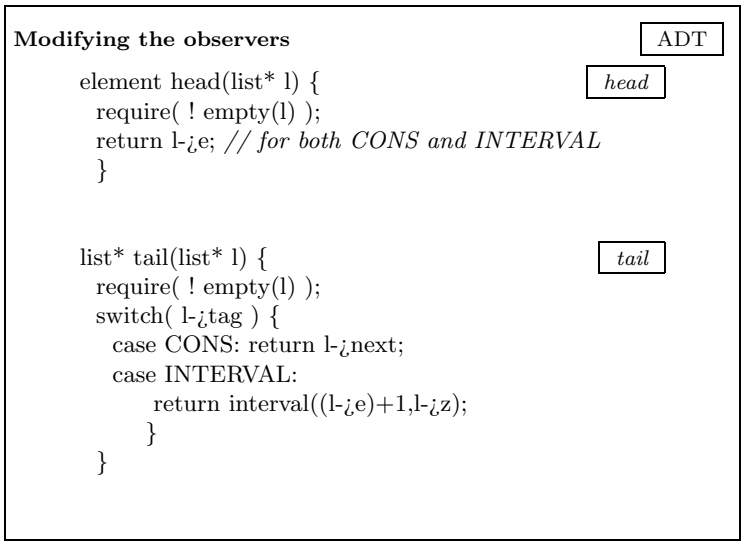
0.1.4 Adding new observers

Now, for the complementary case, what happens when we add new observers to the specification of a data type? Somewhat surprisingly, the object-oriented approach now seems to be at a disadvantage.

Since in a module realization of an abstract data type the code is organized around observers, adding a new observer function amounts simply to adding a new operation with a case for each of the possible generator types, as shown in slide **??**.

When we look at how we may extend a given object realization of an abstract data type with a new observer we are facing a problem.

The obvious solution is to modify the source code and add the *length* function



Slide 0-8: Modifying the observers

to the *list* interface class and each of the generator classes. This is, however, against the spirit of object orientation and may not always be feasible.

Another, rather awkward solution, is to extend the collection of possible generator subtypes with a number of new generator subtypes that explicitly incorporate the new observer function. However, this also means redefining the *tail* function since it must deliver an instance of a *list with length* class.

As a workaround, one may define a function *length* and an extended version of the *list* template class supporting only the *length* (observer) member function as depicted in slide **??**.

A program fragment illustrating the use of the *listWL* class is given below.

```
list<int¿* r = new cons<int¿(1,new cons<int¿(2,new interval(3,7)));
while (! r-¿empty()) {
        cout ¡¡ ((listWL¡ int ¿*)r)-¿length() ¡¡ endl;
        r = r-¿tail();
        }
delete r;
```

Evidently, we need to employ a cast whenever we wish to apply the *length* observer function. Hence, this seems not to be the right solution.

Alternatively, we may use the function *length* directly. However, we are then forced to mix method syntax of the form $ref \rightarrow op(args)$ with function syntax of the form $fun(ref, args)$, which may easily lead to confusion.

Discussion

Adding new generators OOP *0-9*

interval

```
class interval : public list<int¿ {
public:
interval(int x, int y) : _x(x), _y(y) { require( x ¡= y );
    }
bool empty() { return 0; }
int head() { return _x; }
list¡ int ¿* tail() {
  return (_x+1 ¡= _y)?
        new interval(_x+1,_y):
        new nil<int¿;
  }


bool operator==(list@lt;int¿* m) {
  return !m-¿empty() &&
  _x == m-¿head() && tail() == m-¿tail();
  }
protected:
int _x; int _y;
};
```

Slide 0-9: Objects and generators

We may wonder why an object-oriented approach, that is supposed to support extensibility, is at a disadvantage here when compared to a more traditional module-based approach.

As observed in [Cook90], the problem lies in the fact that neither of the two approaches reflect the full potential and flexibility of the matrix specification of an abstract data type. Each of the approaches represents a particular choice with respect to the decomposition of the matrix, into either an *operations-oriented* (horizontal) decomposition or a *data-oriented* (vertical) decomposition.

The apparent misbehavior of an object realization with respect to extending the specification with observer functions explains why in some cases we prefer the use of overloaded functions rather than methods, since overloaded functions allow for implicit dispatching to take place on multiple arguments, whereas method dispatching behavior is determined only by the type of the object.

However, it must be noted that the dispatching behavior of overloaded functions in C++ is of a purely syntactic nature. This means that we cannot exploit the information specific for a class type as we can when using virtual functions. Hence, to employ this information we would be required to write as many variants of overloaded functions as there are combinations of argument types.

Dynamic dispatching on multiple arguments is supported by *multi-methods*

Adding new observers ADT *0-10*

length

```
int length( list* l ) {
      switch( l-¿tag ) {
      case NIL: return 0;
      case CONS: return 1 + length(l-¿next);
      case INTERVAL: return l-¿z - l-¿e + 1;
      };
}
```

Slide 0-10: Modules and observers

Adding new observers OOP *0-11*

length

```
template¡ class E ¿
int length(list¡ E ¿* l) {
   return l-¿empty() ? 0 : 1 + length( l-¿tail() );
 }
```

listWL

```
template¡ class E ¿
class listWL : public list<E> {
public:
int length() { return ::length( this ); }
};
```

Slide 0-11: Objects and observers

in CLOS, see [Paepcke93]. According to [Cook90], the need for such methods might be taken as a hint that objects only partially realize the true potential of *data abstraction.*