The Object-Oriented Architecture of High-Performance Compilers

Version 1.2

Lutz Hamel and Diane Meirowitz and Spiro Michaylov
Thinking Machines Corporation
14 Crosby Drive
Bedford, MA 01730
USA

October 23, 1996

1 Introduction

With the recent advances in object technology, specifically the Standard Template Library [5] and Design Patterns [3], we found it interesting and necessary to revisit the object-oriented design and construction of compilers. In our case, the interest is in the design and construction of high-performance compilers. Here we concentrate on the design of the Intermediate Language (IL) constructs within a compiler. We consider this the main problem facing object-oriented compiler construction today and assume that parsing, symbol tables, error reporting, *etc.* are readily represented within the object-oriented paradigm and we will therefore ignore it here for the sake of clarity and brevity.

The most obvious ("naive") approach to object-oriented compiler construction is, to attach all necessary compilation behavior to the IL [4]. Then, once the IL tree is built, the compiler would simply instruct the objects within the tree to perform transformations on themselves or to translate themselves into assembly language.

However, we feel that this approach does not adequately address the needs of today's state-of-the-art high-performance compiler. Consider some of the characteristics of high-performance compilers:

- not syntax directed
- multiphase
- different perspective of the IL from different phases

- elaborate attribute decoration during various optimization phases
- different IL traversal strategies during different phases
- IL term replacement mechanisms

Therefore, we propose here an approach more amenable to the requirements of high-performance compiler construction. In particular, we advocate what we call the *separation of concerns*: The IL should only be concerned with efficiently and effectively representing the abstract nature of the source program and provide basic functionality for the manipulation and maintenance of this representation. Any agent manipulating the IL should be kept physically separate from the IL and should only access the IL via its interface. We examine the viability of this approach by studying two key problems within compiler design: The interaction of compiler phases with the IL and tree decoration.

The paper proceeds by first examining the question of separation of concerns with regards to the IL and compiler phases in Section 2. We then continue with the more pragmatic aspects of our design in Section 3. Section 4 looks at the problem of tree decoration in the context of the compiler IL and applies our notion of separation of concerns to this design problem as well. Finally, Section 5 finishes off with some concluding remarks.

2 Separation of Concerns: Data vs. Algorithms

A simplified view of high-performance compilation is to view it as a succession of phases each of which represents a transformation algorithm on the IL. Given this view, it makes sense to separate the algorithms and IL physically, allowing the algorithms to easily replace each other during the course of the compilation. In the object-oriented approach to system design this means that the IL and the algorithms are to be encapsulated in separate objects. Here, the IL has an interface which only contains methods pertaining to the manipulation and maintenance of the IL data structures. The algorithms, on the other hand, are separate objects each with an appropriate interface for the algorithm at hand. The algorithmic objects may only manipulate the IL via the explicit IL interface. This is in stark contrast to the naive approach mentioned earlier where the IL data structures and the algorithms are lumped together into a single entity.

Consider Figure 1. Here, the "Compiler Substrate" represents the compiler infrastructure we chose to ignore for the sake of this discussion, i.e., it consists of IO, the symbol table, text tables, type system, *etc.* Figure a) represents the naive object-oriented approach to compiler construction. We assume that at some point during compilation the compiler substrate attempts to perform certain transformation algorithms on the IL. These algorithm invocations are denoted with arrows labeled 'a' and 'b'. Here, the compiler substrate issues the requests 'a' and 'b' to the algorithms 'A' and 'B' embedded in the IL structure

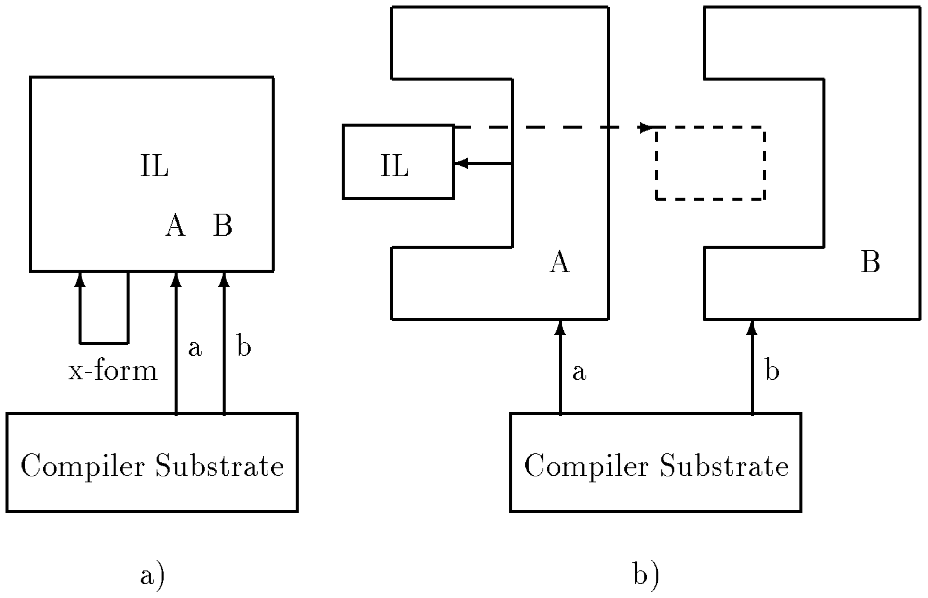

Figure 1: Interaction of compiler phases with the IL; a) the naive design, b) the design using the 'separation of concerns'.

which in turn call upon the IL with transformation requests. These transformation requests are denoted with the arrow labeled 'x-form'. Figure b), on the other hand, represents the more modular approach to objet-oriented compiler construction which we advocate here. Again, the compiler substrate will issue the requests 'a' and 'b'. However, the main distinction here is, that the algorithms are no longer part of the IL but are objects in their own right which manipulate the IL. The algorithmic object 'A' fulfills request 'a' by applying appropriate transformations to the IL indicated by the arrow from 'A' to the 'IL'. Once 'A' has fulfilled its request it relinquishes control of the IL so that algorithmic object 'B' may gain control of it (indicated by the bold dashed arrow) in order to fulfill request 'b'.

Let us examine some of the advantages of this architecture:

- The resulting compiler is much more modular and furthermore, the physical module structure directly reflects the logical structure of the compiler making it easier to understand and maintain.
- The separate encapsulation of the IL as well as the algorithms hides implementation concerns from each other. This makes the compiler more resilient to change over time, since changing implementation details tend not to affect the overall system assuming that the semantics of the interfaces are preserved.
- The interfaces of each of the objects are only as "wide" as they need to be to provide the functionality needed at a particular point in time. This conforms precisely to Booch's key points for the assessment of the quality of an abstraction: *primitiveness* and *completeness* [2]. Compare this to the naive approach where all methods of all phases are visible at all stages of compilation.

3 Achieving the Separation of Concerns

The idea of separating data structures from algorithms is not new. The most widespread use of this idea is in the Standard Template Library [5]. In STL jargon, computational structures break down into *containers* and *generic algorithms.* Containers are lists, queues, stacks and sets. The idea is that any generic algorithm can be hooked up with any container in order to provide an appropriate computational structure. The algorithms gain access to individual data items within a container via iterators.

Even though the STL approach seems attractive for compiler construction it falls short on two accounts:

- The IL structure we are interested in separating from its algorithms is much more complicated than any of the containers considered in the STL.

More specifically, the IL is an n-ary tree and the STL iterator model does not work well with this kind of structure.

- More importantly, iterators are not type sensitive to items over which they traverse. Given a polymorphic container, iterators only know how to retrieve objects by their common base class possibly necessitating a creative type casting scheme in order to make them work with these polymorphic containers [6].

The "Visitor Pattern" introduced in the Design Patterns book [3] offers a much more elegant solution. It does not suffer the short comings of the iterators due to the fact that it places iteration under complete user control and that it is sensitive to polymorphic type structures. Specifically, the visitor allows one to attach operations to an object structure in a non-intrusive way. Put in other words, the visitor allows one to define new operations for an IL structure without having to modify the classes of the IL structure. This allows us to define phase-specific IL operations without having them embedded in the IL class definitions. Thus, the visitor enables us to construct algorithms which make use of phase-specific IL operations without imposing on the IL structure itself.

The key idea here is that we incorporate the operations in question in visitor objects distinct from the IL objects. We then connect the visitor objects and the IL objects via a *double dispatch scheme*.

Let us examing this a little bit closer. Assume that we have a very simple IL structure such as

```
     Ast
     / \
    /   \
Plus  Const
```

More concretely, the `Ast` class looks like the following:

```
class Visitor;

class Ast
{
     vector<Ast *> _children;
protected:
     Ast ();
     Ast (Ast * child1);
     Ast (Ast * child1, Ast * child2);
public:
     Ast * getChild (int pos);
     int getArity ();
     virtual void accept (Visitor * v) = 0;
     virtual ~ Ast ();
};
```

This is a simple abstract base class specifying the basic shape of the IL. Here, each `Ast` node in the IL has zero or more `Ast` nodes as children which we can access through the object's public interface. The most prominent feature of this class is the `accept` function. This function allows the objects in the IL structure to accept a handle to a visitor object.

Let us turn to a more concrete node in the IL, namely the `Plus` node:

```
class Plus : public Ast
{
public:
     Plus (Ast * child1, Ast * child2);
     void accept (Visitor * v)  { v->visitPlus(this); }
     ~ Plus ();
};
```

And similarly the `Const` class:

```
class Const : public Ast
{
     int _val;
public:
     Const (int val);
     int getVal ();
     void accept (Visitor * v)  { v->visitConst(this); }
     ~ Const ();
};
```

The most striking aspect of these IL class definitions is, that they are completely void of any unrelevant methods providing a clean interface to the essentials of the IL. The next thing to be noticed is the `accept` function which overrides the corresponding pure virtual function in the `Ast` base class and does nothing more than to immediately turn around and invoke the appropriate `visit` method of the visitor with the instance pointer as an argument. Given this, the visitor now has a handle on the IL node of interest and may now execute its operations in the context of this IL node. Our double dispatch scheme is now in place: we invoke the method `accept` on our IL object with a visitor as argument, then we turn around and call the appropriate `visit` function on our visitor object with the IL node as argument.

Let us take a look at the visitor side of things. Assume we have a class called `Visitor` which is defined as follows:

```
class Visitor
{
```

```
protected:
     Visitor ();
public:
     virtual void visitAst (Ast *) = 0;
     virtual void visitPlus (Plus *) = 0;
     virtual void visitConst (Const *) = 0;
     virtual ~ Visitor ();
};
```

This abstract visitor has one **visit** method for each **Ast** node in the IL class hierarchy. Let us specialize this visitor to something which does something useful, for example, print the IL structure in some human readable form to the standard output. Consider the following class definition:

```
class PrintVisitor : public Visitor
{
public:
     PrintVisitor ();
     void visitAst (Ast *)   { abort(); }  // should never happen
     void visitPlus (Plus *);
     void visitConst (Const *);
     ~ PrintVisitor ();
};
```

Again, **PrintVisitor** has a function for each **Ast** node which overrides the pure virtual functions in the **Visitor** class from which it was derived. Since **Ast** is an abstract class, the function **visitAst** should never be invoked. We mark it as such by aborting if it ever should get invoked. The more interesting cases are the other two functions. Consider the implementation of **visitConst**:

```
void PrintVisitor::visitConst (Const * node)
{
     cout << node->getVal();
}
```

This function, given a pointer to a **Const** node, will obtain the value of the constant and put it onto the standard output stream. In essence, this **visitConst** function implements a print function for the IL node **Const**. Therefore, by attaching the visitor to the IL node we are able to implement a print function for the IL node without modifying the **Const** class.

The **visitPlus** function is defined as follows:

```
void PrintVisitor::visitPlus (Plus * node)
{
     PrintVisitor * visitor_handle;
```

```
          cout << "(";

          visitor_handle = new PrintVisitor;
          node->getChild(0)->accept(visitor_handle);
          delete visitor_handle;

          cout << "+";

          visitor_handle = new PrintVisitor;
          node->getChild(1)->accept(visitor_handle);
          delete visitor_handle;

          cout << ")";
     }
```

The function prints out a left parenthesis and then creates a new `PrintVisitor` for the left child node of the `Plus` node. Once the visitor node is created it is attached to the left child via the `accept` function. Attaching the new visitor to the child node forces the execution of the methods within the visitor in the context of this child node, i.e., the child node is printed to the standard output stream. This of course happens recursively on every level of the IL tree. On return of the `accept` function we delete this visitor object. We then print out a plus sign and continue to create a visitor for the right child of this `Plus` node. Again, once created we attach it to the right child and then delete it when it is no longer used. As the last thing we print out the closing parenthesis.

Now that we saw what the IL and visitor class structure looks like, what does the `main` program driving the whole thing look like? The answer is; it looks very similar to the `visitPlus` function:

```
main()
{
     // set up some IL tree: 2 + 1
     Ast * ast = new Plus(new Const(2), new Const(1));

     PrintVisitor * visitor_handle = new PrintVisitor;

     cout << "Printing IL: ";
     ast->accept(visitor_handle);
     cout << endl;
}
```

Here, we create some piece of IL[1]. Following that we instantiate a `PrintVisitor`. After printing out some prologue we hand the visitor to the IL which in turn will print the IL to the standard output. Finally, we print a new line.

[1]Presumably, in a real compiler this is done by the parser.

In summary, the "Visitor Pattern" is a powerful idiom for separating algorithms from data structures. It is preferable over iterators for data structures which have a non-linear internal structure. Furthermore, it addresses polymorphism in such situations much more adequately than iterators. As the example above demonstrates, it is particularly well suited for IL data structure and algorithms design.

4 Decoration on the Fly

One of the more vexing problems in high-performance compiler construction is the management of attributes on the IL structure. These attributes are typically things such as *du*-chains for dataflow and dependence analysis or other context sensitive information for IL nodes [1]. Traditionally, attributes are attached to the IL nodes with the consequence that phase-specific attributes become visible globally. Here we advocate that attributes should in fact be separated from the IL structure and be managed by separate objects with the cooperation of the IL. In some sense, we are arguing again for the separation of concerns: the IL should represent the abstract nature of the program in question, nothing more nothing less. Attributes should be concerned with the presentation and management of local context sensitive IL information.

As a solution to this problem we advocate the notion of an *attribute manager*. Our view of an attribute manager is that of a very thin veneer over the IL structure. Each attribute is associated with exactly one manager. Adding or deleting attributes to or from the IL becomes a matter of instantiating or deleting the appropriate managers, respectively. Attaching an attribute value to an IL node is then handled by the particular manager by providing an association between the node and the attribute value. The IL node itself is not modified.

Figure 2 attempts to illustrate this. Here we have the "Compiler Substrate", "Phase" and "IL" similar to the discussion in the previous section. We denote the attribute veneer with a dashed box around the "IL". The veneer is provided by the "Attribute Manager" indicated with the dashed line. Given an attribute manager, it allows one to associate specific IL nodes with particular attribute values. Compiler phases may then deposit and recall attribute values for IL nodes using the attribute manager. This is indicated with the bidirectional arrow between the "Phase" and the "Attribute Manager".

This is a powerful mechanism which allows one to attach values to IL nodes without having to modify the underlying IL classes.

The implementation of the attribute veneer through the associated managers is accomplished with C++ template classes allowing us to instantiate attribute manager classes which may represent any type of attribute. Ignoring some of the details such as the need to keep lists of attribute managers around for book keeping purposes, let us take a look at the basic `AttributeManager` class:

```
template <class NodeClass, class BaseClass, class ValueClass>
```

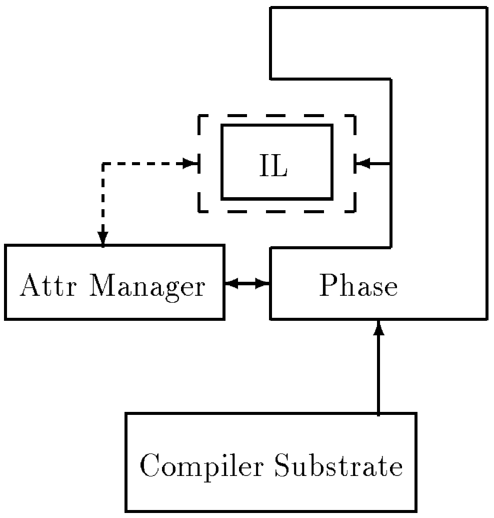

Figure 2: Decorating the IL using Attribute Managers.

```
class AttributeManager : public AttributeManagerBase<BaseClass>
{
public:
    AttributeManager(ValueClass null_value, char *attr_name);
    ~AttributeManager();

    void set_attr(NodeClass *node, ValueClass value);
    ValueClass attr(NodeClass *node) const;
    ValueClass is_null(NodeClass *node) const;
    void print(ostream &out, NodeClass *node) const;
private:
        // ...
};
```

Let us start with the template parameter classes. The type `NodeClass` denotes the possible types of IL nodes this manager can be instantiated with. The type `BaseClass` denotes the root class of the particular IL in question and is used in the `AttributeManagerBase` class for the polymorphic handling of lists of attribute managers. Finally, the `ValueClass` denotes possible attribute types for the attribute manager.

Let us look at the public interface. The constructor takes the *null value* for a particular attribute and a name for the attribute in question. The name string is there for debugging and printing purposes only. The next interesting function is `set_attr`. Given a pointer to an IL node and an appropriate attribute value, this function associates the IL node with the attribute value (presumably this is done via some sort of associative array or hash table). The function `attr` returns the attribute value associated with a particular node. The function `is_null` allows one to test the presence of an attribute value for a particular node. And finally, `print` allows one to print the attribute value for a particular node to an output stream.

Consider the following IL class hierarchy:

```
   Ast
   / \
  /   \
Plus  Const
```

Instantiating an attribute manager for `Plus` nodes which manages some sort of expression nesting count which is simply an integer looks like this:

```
AttributeManager<Plus,Ast,int> nesting_count;
```

Setting and inquiring attributes is achieved as follows:

```
// create our attribute manager
AttributeManager<Plus,Ast,int> nesting_count;

// create some IL structure
Plus * node = new Plus(...);

// attach an attribute value to the 'node'
nesting_count.set_attr(node, 1);

// retrieve an attribute value
int local_count = nesting_count.attr(node);
```

Some observations are in order. First, since we defined the attribute manager as a template, we may create any attribute type we like by appropriately instantiating the `AttributeManager` template class. Second, because the attributes are managed as a physically separate entity from the IL structure, creating and destroying attributes "on the fly" becomes a reality and desirable. Attribute managers only live as long as they are needed for a particular computation and are then removed without any impact on the IL. Furthermore, because the memory allocated to attribute values resides within the manager there is no need to walk the IL tree to clean up unused attribute values. Simply, removing the appropriate attribute manager achieves that goal in this scheme.

As we briefly mentioned above, we ignored some fundamental issues in this discussion. The most important one is the need of keeping lists of attribute managers around. The need arises due to the fact that the IL might change while attribute managers are instantiated. In particular, IL nodes which have entries in attribute managers might get deleted. It turns out that there needs to be a cooperation between the IL and the attribute managers: if an IL node is deleted the destructor for that IL node will notify the list of attribute manager of this event. The list will then continue to notify all its associated managers that this particular node no longer exists. The managers in turn will delete the entries for this node from their association tables.

5 Conclusions

With the recent advances in object-oriented technology we found it necessary to revisit the design and construction of high-performance compilers. We primarily focused on IL design, since we believe that other parts of compilers such as symbol tables and type systems are readily represented in the object-oriented paradigm.

Our compiler design philosophy is based on the *separation of concerns*: The IL should only be concerned with the abstract representation of the input program and provide an interface just "wide" enough for its effective manipulation and maintenance. Agents accessing and manipulating the IL should be kept separate from the IL and should only access it through its interface.

By successfully solving two key problems in object-oriented compiler design, the interaction between compiler phases and the IL as well as tree decoration, we have shown that this separation of concerns leads to clean physical compiler designs which directly reflect its logical structure and therefore are easily understood, modified, and maintained. The "Visitor Design Pattern" is a fundamental building block within these designs.

References

[1] Alfred Aho, Ravi Sethi, and Jeff Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[2] Grady Booch. *Object-Oriented Analysis and Design with Applications, Second Edition.* Benjamin/Cummings, 1993.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1995.

[4] Jim Holmes. *Object-Oriented Compiler Construction.* Prentice-Hall, 1995.

[5] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, Reading, Massachusetts, 1996.

[6] Matthew Nguyen. Extending the visitor design pattern. *Dr. Dobbs Journal*, October 1996.

[7] Bjarne Stroustrup. *The C++ Programming Language Second Edition.* Addison-Wesley, Reading, Massachusetts, USA, 1995.