

Object-oriented Implementation of Intensional Languages

Weichang Du

*Department of Mathematics and Computer Science
University of New Brunswick
Saint John, N.B. Canada E2L 4L5
wdu@unb.ca*

Abstract

This paper presents an object-oriented implementation of Lucid. We implement each variable in a Lucid program as a class. An object of the class is the variable in a context. An object evaluates itself when it receives a demand for its value and stores the value. Evaluation of a program involves object self-evaluations and message passing of demands among objects. Some optimization issues are also discussed in the paper.

1 Introduction

Object-oriented programming has become one of the main stream paradigms for software construction. We may consider to incorporate object-oriented technology in development of intensional languages at two levels. At the language extension level, we may add the concept of objects into intensional languages. [FB91] and [DF94] have addressed this issue. At the implementation level, we may use object-oriented techniques to compile intensional programs to object-oriented programs. This is addressed in this paper.

Intensional languages are implemented based on the eduction model. The operational semantics of the languages is defined on an abstract machine [AFJ91] [RW93]. The machine consists of a global stack machine, a global value warehouse, and a set of global context registers. The demand driven computation is centralized around those global structures: all computations are done by the stack machine, accesses to variable values are through the value warehouse, and context switching is manipulated in the context registers. When an intensional program is compiled to a conventional program, such as C, the global data structures corresponding to the abstract machine are created, and the compiled code basically manipulates those global structures using eduction [RW94].

In the traditional implementation of intensional languages, we can think of that there is an implicit control process as the main thread of eduction. It receives demands for variable values, checks the value warehouse, fetches and interprets variable definitions, creates more demands, evaluates expressions on the stack machine, stores values in the value warehouse, and switches contexts in the contexts registers. This centralized control process, in a sense, obscures the distributed nature of the eduction model.

In this paper, we present an object-oriented implementation of the eduction model. In the implementation, we represent a variable in an intensional program in a context as an object which is identified by the variable and the context. All objects of the variable constitute a class named by the variable. The state of an object consists of a value and an evaluation status. The behavior of an object is the evaluation of the variable in the context, according to the variable's definition in the program. In this implementation, there is neither any global structure nor any centralized control process. Computation in this implementation consists of self-evaluations of objects and message passing of demands among objects. The computation is fully distributed eduction. An object starts to evaluate itself when there is a message sent to it from another object that demands its value.

During evaluation of an object, when a value of another object is needed, a demand message is sent to that object. In the object-oriented implementation, we can think of that the evaluation engine, value warehouse and context registers are distributed to individual objects, and context switching means sending messages to other objects with switched contexts.

In Section 2, we describe compiling Lucid programs to C++ programs in details. In section 3, we discuss some optimization issues. Section 4 is some concluding remarks.

2 Compiling Lucid to C++

Given a Lucid program, we can first transform it to 0-order using Yaghi's method [Yag84], and without nested *where* clauses using λ -lifting [Jon85]. In the following discussion, by a Lucid program we mean its flatten 0-order form.

We compile each variable definition in a Lucid program into a C++ class definition, which we call a **variable class**, which has the same name as the variable. The whole compiled C++ program consists of these variable classes plus a main function to start computation.

Depending on the (static) dimensionality of the variable, an instance of a variable class (i.e an object) has a set of private *context* members. The value of a context member is the coordinate of a dimension in which the variable varies. The values of all context members of the object determine the identity of the object in its class, i.e. its corresponding context in the context space. The value of any context member is not mutable after the object is created.

An instance of a variable class has a private *value* member. The *value* will hold the object's value after it is evaluated by its behavior.

There is also a private boolean member *iseval* that tells if the object's value has been evaluated and the value has been stored in the *value* member.

An object has two member functions in a variable class. The public *getvalue()* sends the value of this object to the object that demanded it. If the value has not been evaluated, it calls the *eval()* member function to evaluate the object first.

The private *eval()* function is compiled from the defining expression of the variable whose class the object belongs to. When it is called, *eval()* evaluates the object and then stored the value to the member *value*. The implicit context in the evaluation refers to the context of this object. When value of another object is needed during the evaluation, a *getvalue* message is sent to that object to get the needed value from that object.

The following is an example. The given Lucid program computes the sequence of powers $\langle 1^1, 2^2, 3^3, 4^4, 5^5, \dots \rangle$. The context space of the program is two-dimensional. The variables *i2i* and *n* vary in dimension 0 only, and *power* varies in the both dimensions. *n*'s value is the sequence of natural numbers in dimension 0. *power* is a matrix where at each (i,j) , its value is i^j . *i2i* as a vector in dimension 0 extracts *power*'s value in diagonal.

```
i2i
where
  i2i = power at(1) n-1;
  power = n fby(1) power * n;
  n = 1 fby(0) n + 1;
end
```

Program A in Appendix is the compiled C++ program from the above Lucid program. In the program, we define a superclass *variable* of all variable classes, which captures the common members and behaviors of all subclasses.

We define intensional operator *prev* as a macro in the program and treat *fby* as a conditional *if current_context eq 0 then x else prev y*.

When an object is constructed from a variable class, it is initialized by its corresponding context which is the object's identity.

Obviously, this compiled C++ program would not be efficient. It even does not completely fulfill the education model. For there are repeated evaluations to the same values. Every time an object's

value is demanded, the object is first created as a new object then be evaluated. Although the value of an evaluated object does be stored in the object, the object can never be referred to again by other objects. The problem is that objects are dynamically created during program evaluation. In order to refer to a dynamically created object, we have to find a way to let other objects know where the object is. We give a solution to this problem in the next section.

3 Some Optimization Issues

In the traditional implementation of education, we use a global value warehouse to store values of variables in various contexts. The value warehouse can be accessed to in evaluation of any variable value. In the object-oriented implementation, there is no global data structure including the value warehouse. When an object's evaluation demands another object's value, it first has to find where the demanded object is if it has been created. An object is dynamically created from the variable class which the object will belong to. Therefore, logically, to find a demanded object a , the demanding object b needs to ask a 's class A . This can be done by (i) memorizing all the objects created in A and (ii) sending A a message from b to ask the service. Thus a variable class itself is also an object, whose state at any time during the computation is a table of all its objects created so far, and whose behavior is the service to return a particular object. From another point of view, in this way, we distribute the traditional global hashing table of the value warehouse to individual classes.

Program B in Appendix is a C++ implementation of the power example using the above scheme. In the compiled C++ program, the static members of a variable class are the class members and methods. In the program, for each variable class we dynamically allocate an array of pointers to objects of the class. The dimensionality of the array depends on the dimensionality of the variable. Each array element corresponds to a context, i.e. the hashing function here is the identity function. The pointers to dynamically created objects are stored in the array at corresponding positions. When an object is demanded, if it has been created, the class will create the object, otherwise the pointer to the object will be sent back to the demanding object.

When implementing a recursive function (in the form of $Call_i f$) on stock hardware, we can use a stack to store object references in the function's class, instead of a hash table. Consider the *call* dimension of a function is an n -ary infinite tree, where n is the number of calls in the static program and each branch of the tree corresponds to a call. A context in the *call* dimension is a node of the tree, and the function in that context is represented by an object of the function's class with the context. The object will be created and evaluated when the function's value in that context is demanded. However, an evaluated object can only be demanded by the objects at the child nodes of the tree. Thus when all the child objects have been evaluated, the object can be deleted from the stack because there will be no further reference to it.

Dynamic creation of objects during program execution costs run-time efficiency. If the range of contexts in which values will be evaluated is known at compile time, we may statically create those objects that may be needed in the execution at compile-time, in the cost of storage space. This does not violate the rule of lazy evaluation, because no created object will be evaluated until it is demanded, though it does exist from the beginning of the computation.

Program C in Appendix shows the static-allocation version of the power example. In the program, all objects are created at compile time in the form of global object arrays each of which is created from a variable class.

The object reference tables contained in variable classes can actually hold values of the objects, instead of pointers to the objects, when the program is implemented on stock hardware. For when an object is created in a variable class, it will be evaluated immediately, and the evaluation will be completed before any new demand comes for its value. In other words, in this case we can think of that they are value warehouses distributed to individual classes.

However, when we consider to implement the program on a parallel environment, there is an advantage to store object references, instead of values. The idea is to allow more than one object to be evaluated actively (not suspended) in parallel. When performing a strict operation on values of two or more objects, we can demand the objects simultaneously. Base on strictness analysis, we can

build a partial order for all other objects which the object's evaluation depends on. Thus an object may start to evaluate itself as soon as its predecessors in the partial order have been evaluated. During evaluation, the class tables may contain references to those objects that are being evaluated. For an object that is being evaluated, we can associate it with a waiting queue of references to the objects that are demanding its value. When the value is available, it will be returned to those waiting objects.

4 Conclusions

There are several advantages of implementing intensional programs to object-oriented programs.

In theoretical study, this semantics-preserving compilation from intensional programs to object-oriented programs shows a close relationship between operational semantics of intensional languages and object-oriented languages.

In software development, one-to-one compilation of variable or function definitions to class definitions individually supports program modularity and software reuse. For compiled classes can be imported to and be reused in many programs.

In language implementation, we can take advantage of the existing advanced compilation techniques for object-oriented languages to improve implementation of intensional languages on conventional systems.

In parallel computing, object-oriented education provides a natural way to evaluate intensional programs in a distributed and parallel manner. We can make use of and extend research results on concurrent object-oriented languages for parallel implementation of intensional languages.

References

- [AFJ91] E. A. Ashcroft, A. A. Faustini, and R. Jagannathan. An intensional language for parallel applications programming. In *Parallel Functional Languages and Compilers*, pages 11–50. ACM Press, 1991.
- [DF94] W. Du and A.A. Faustini. Objectflow – adding object to glu. In *Submit to The 1994 International Symposium on Lucid and Intensional Programming*, September 1994.
- [FB91] B. Freeman-Benson. Lobjcid: Objects in lucid. In *The 1991 International Symposium on Lucid and Intensional Programming*, pages 80–87, April 1991.
- [Jon85] T. Jonsson. Lambda lifting: Transforming programs to recursive equations. In *IFIP Conference on Functional Programming Languages and Computer Architectures*, pages 190–203, 1985.
- [RW93] P. Rondogiannis and W.W. Wadge. A dataflow implementation technique for lazy typed functional languages. In *The Sixth International Symposium on Lucid and Intensional Programming*, pages 23–42, April 1993.
- [RW94] P. Rondogiannis and W.W. Wadge. High-order dataflow and its implementation on stock hardware. In *the ACM Symposium on Applied Computing*, pages 431–435. ACM Press, 1994.
- [Yag84] A.A. Yaghi. *The Intensional Implementation Technique for Functional Languages*. PhD thesis, Department of Computer Science, University of Warwick, Coventry, UK, 1984.

APPENDIX

Program A:

```
#include<iostream.h>
#define context_range 5
#define prev(x) x-1

class variable {
    int iseval;
    virtual void eval() = 0;
protected:
    int value;
public:
    variable() { iseval = 0;}
    int getvalue()
    {if(!iseval) {eval(); iseval=1;}; return value;}
};

class n: public variable {
    int d0;
    void eval() {
        if (d0 == 0)
            value = 1;
        else { n *pn = new n(prev(d0));
              value = pn->getvalue() + 1;};
    }
public:
    n(int c) { d0 = c;}
};

class power: public variable {
    int d0, d1;
    void eval() {
        if (d1 == 0)
        { n* pn = new n(d0);
          value = pn->getvalue();}
        else
        {power *pp = new power(d0, prev(d1));
          n* pn = new n(d0);
          int temp1 = pp->getvalue();
          int temp2 = pn->getvalue();
          value = temp1 * temp2;};
    }
public:
    power(int c0, int c1) {d0 = c0; d1 = c1;}
};

class i2i: public variable {
    int d0;
    void eval() {
        n* pn = new n(d0);
        int temp1 = pn->getvalue();
        power* pp = new power(d0, temp1-1);
    }
};
```

```

        value = pp->getvalue();
    }
public:
    i2i(int c) {d0 = c;}
};

main() {
    for (int d0 = 0; d0 < context_range; d0++)
    { cout << "d0 = "; cin<< d0;
      i2i *pi = new i2i(d0);
      cout << pi->getvalue() << endl;}
}

```

Program B:

```

// headings as same as Program A

class n: public variable {
    static n **objtable;
    int d0;
    void eval() {
        if (d0 == 0) value = 1;
        else { n *pn = n::getobj(prev(d0));
              value = pn->getvalue() + 1;};
    }
public:
    n(int c) { d0 = c;}
    static n* getobj(int c0)
        {if (!objtable[c0]) objtable[c0]=new n(c0);
         return objtable[c0];}
    static init_objtable(int size) {objtable = new n*[size];}
};
n** n::objtable = 0;

class power: public variable {
    static power ***objtable;
    int d0, d1;
    void eval() {
        if (d1 == 0){ n* pn = n::getobj(d0);
                     value = pn->getvalue();}
        else { power *pp = power::getobj(d0, prev(d1));
              n* pn = n::getobj(d0);
              int temp1 = pp->getvalue();
              int temp2 = pn->getvalue();
              value = temp1 * temp2;};
    }
public:
    power(int c0, int c1) {d0 = c0; d1 = c1;}
    static power* getobj(int c0, int c1)
        {if (!objtable[c0][c1]) objtable[c0][c1]=new power(c0,c1);
         return objtable[c0][c1];}
};

```

```

        static init_objtable(int size1, int size2)
        {objtable = new power**[size1];
         for (int i=0;i<size1;i++) objtable[i] = new power*[size2];}
};
power ***power::objtable = 0;

class i2i: public variable {
    static i2i **objtable;
    int d0;
    void eval() {
        n* pn = n::getobj(d0);
        int temp1 = pn->getvalue();
        power* pp = power::getobj(d0, temp1-1);
        value = pp->getvalue();
    }
public:
    i2i(int c) {d0 = c;}
    static i2i* getobj(int c0)
        {if (!objtable[c0]) objtable[c0]=new i2i(c0);
         return objtable[c0];}
    static init_objtable(int size) {objtable = new i2i*[size];}
};
i2i **i2i::objtable = 0;

main() {
    n::init_objtable(context_range);
    power::init_objtable(context_range,context_range);
    i2i::init_objtable(context_range);
    for (int d0 = 0; d0 < context_range; d0++)
    { i2i *pi = i2i::getobj(d0);
      cout << pi->getvalue() << endl; }
}

```

Program C:

```

// headings as same as Program A

class n: public variable {
    int d0;
    void eval();
public:
    n(int c=0) { d0 = c;}
};

class power: public variable {
    int d0, d1;
    void eval();
public:
    power(int c0=0, int c1=0) {d0 = c0; d1 = c1;}
};

```



```

class i2i: public variable {
    int d0;
    void eval();
public:
    i2i(int c=0) {d0 = c;}
};

n n_objtable[context_range] = {n(0),n(1),n(2),n(3),n(4)};
power power_objtable[context_range][context_range] = {
    {power(0,0),power(0,1),power(0,2),power(0,3),power(0,4)},
    {power(1,0),power(1,1),power(1,2),power(1,3),power(1,4)},
    {power(2,0),power(2,1),power(2,2),power(2,3),power(2,4)},
    {power(3,0),power(3,1),power(3,2),power(3,3),power(3,4)},
    {power(4,0),power(4,1),power(4,2),power(4,3),power(4,4)}};
i2i i2i_objtable[context_range] =
    {i2i(0),i2i(1),i2i(2),i2i(3),i2i(4)};

void n::eval() {
    if (d0 == 0) value = 1;
    else value = n_objtable[prev(d0)].getvalue() + 1;
}

void power::eval() {
    if (d1 == 0) value = n_objtable[d0].getvalue();
    else
        {int temp1 = power_objtable[d0][prev(d1)].getvalue();
         int temp2 = n_objtable[d0].getvalue();
         value = temp1 * temp2;};
}

void i2i::eval() {
    int temp1 = n_objtable[d0].getvalue();
    value = power_objtable[d0][temp1-1].getvalue();
}

main() {
    for (int d0 = 1; d0 < context_range; d0++)
        cout << i2i_objtable[d0].getvalue() << endl;
}

```