Software Reuse in Intensional Programming

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 a technique to compile intensional functions to class libraries for reuse. We compile an intensional function to a class module in an object-oriented language such as C++. The key issue is to avoid global analysis of the intensional program for information of function applications. Instead, we let a function application register itself to the function dynamically at runtime. At the registration time, a registration number is given for future references to the application. This technique is implemented by message passing between two classes, one is compiled from the user function which contains the application, and the other is compiled from the function which the user applies for.

1 Introduction

In the past years, many Lucid-based intensional programs have been developed. Most of these programs were written from scratch or built up from reused source code. As intensional programming is becoming more mature, more and more intensional programs for real-world applications will be developed. These programs will be very complex, compared to "toy" programs we wrote in the past. To reduce development complexity, we need to develop software libraries of common basic components which can be reused by application programs. Each library component can be well tested for its functionality and pre-compiled. Thus a complex application can be written as a composition or linkage of those building blocks at high level. we do not need to rewrite and/or we compile these components again and again.

GLU is the first attempt to introduce software reuse in intensional programming, besides its other features [JF90]. In GLU programming, we use conventional C functions as basic building blocks. Those C functions can be reused from existing C libraries without rewriting and even recompile them. Since these C functions were well tested when the libraries were built and proved working, we only use their functionalities even without knowing how they are implemented in C. The composition of
the C functions in GLU is specified by Lucid program. Thus the program hierarchy of
GLU has two levels. The lower level is C functions which are mainly reused software.
The higher level is the Lucid program which consists of definitions of intensional
variables and functions.

In pure intensional programming (or at the higher level of GLU programming),
we may consider intensional functions as basic building blocks, and an intensional
program as a (complex) composition of intensional functions. In this paper, we discuss
software reuse of intensional functions, in particular Lucid functions, at compiled
binary code level.

In order to build a library of intensional functions, we need to implement, in a
conventional language, each intensional function independent of their program con-
texts where they will be used. However, this is difficult based on the standard Yaghi’s
algorithm for intensional function implementation [Yag84]. In the algorithm, we re-
duce a non-zero order intensional program to a zero order one by transforming each
function and each of its formal parameters to variables. A global static analysis of the
program is required in the transformation to determine the correspondence between a
function application (call\(i\),\(f\)) and each of its arguments in the definitions of function’s
parameter (\(\text{actuals}(\ldots, e_i, \ldots)\)). The following example shows the transformation.

\[
f(3) + g(4)
\]
where\[
  \begin{align*}
  f(x) & = x \times g(x) \\
  g(y) & = y + 1 \\
  \end{align*}
\]
end\[
call(0,f) + call(0,g)
\]
where\[
  \begin{align*}
  f & = x \times call(1,g) \\
  x & = \text{actuals}(3) \\
  g & = y + 1 \\
  y & = \text{actuals}(4, x) \\
  \end{align*}
\]
end

In the above example, to implement function \(f\) and \(g\), we need to know the
numbers of static applications of \(f\) and \(g\) in the entire program, as well as one-to-one
correspondence of an argument in \(x\) and \(y\)’s \textit{actuals} operators and their \textit{calls}. If we
implement the functions out of the program context, we wouldn’t know what is the
call number for \(g\) in \(f\)’s code and how this call number corresponds to the exact
argument in \(y\)’s \textit{actuals}.

In this paper, we describe a technique that dynamically establishes this corre-
spondence at program execution time. It allows us to implement functions independent
of program contexts. Hence the implemented functions can be reused to build many
programs.

The idea is simple. We let each function register, at runtime, each function appli-
cation in its code to the applied function. The applied function will return a unique
registration number. The number will be used as the call number in the application side and used as the index of the argument of the application in the parameter’s actuals list. This dynamic registration activity can be done through message passing between objects in an object-oriented implementation of intensional programs.

In the next section, we describe a general strategy to implement intensional programs with classes and objects. In Section 3, we give details about compiling functions to class modules with emphasis on the dynamic registration of functions. Section 4 is some concluding remarks.

2 Object-oriented Implementation Strategy

In this strategy [Du94], the eductive computation model is implemented by cooperation of classes and objects through local evaluation and message passing for demands and values. Unlike the conventional, abstract machine-based implementation, the OO implementation is not based on any abstract architecture; it directly implements the eduction with the OOP semantics.

The OO implementation is based on the relationship between concepts of intensional programming and object-oriented programming. [Wad96] gives a theoretical foundation for the relationship. The idea is quite simple to compile or map intensional programs to OO programs.

Given a zero-order intensional program consisting of a set of variable definitions, we compile each variable to a class. An object of an variable class corresponds to the variable in a context. The object evaluates the value of the variable in that context when there is a demand for it. The object is dynamically created by the class when the value is being demanded first time during the computation.

In evaluation of an object \( a \), when a value of a variable \( B \) is needed, the object \( a \) sends a demand, by message passing, to the object \( b \) that evaluates the value through \( b \)'s class \( B \).

In the target object system, there are objects at two levels. At the higher level, there are classes as objects which manages its dynamic objects, and transfers demands to its objects. At the lower level, there are objects belonging to individual variable classes. Each such object is identified by a context. Its behavior is to evaluate the value of the variable in the context.

In detailed implementation, each class maintains a hash table of its objects. The hash function applies to given contexts. During computation, the evaluation of an object may generate a demand to another object in another variable class. The demanding object sends a demand message to the variable class. The class will check the demanded object in its table, if it already exists, a pointer to object is returned to the demanding object. Otherwise, the class will create a new object in the class, start object's evaluation, and then return the object pointer to the demanding object.

For a non-zero-order intensional program, we first transform each function definition into a set of variable definitions, using Yaghi's algorithm [Yag84], to make the program zero-order. An \( n \)-ary function is transformed to \( n + 1 \) variables: one corresponding to the function itself, each of the others corresponding to a formal parameter.
which is defined by the `actualls` operator. All function applications are transformed to
the `call` operators. A tree dimension is added to the program’s context space to han-
dle function calls using intensional semantics. We then implement the transformed
program in the same way as zero-order intensional programs but with special care to
the two operators `call` and `actualls`.

The following is an example. Figure 1a shows the original intensional program,
Figure 1b shows the transformed zero-order program, and Figure 1c shows a skeleton
of the compiler OO program in C++.

**Figure 1a:**

```
r
where
  r = f(n) @ 5
  f(x) = if odd(x) then x else next x fi;
  n = 1 fby n + 1;
end
```

**Figure 1b:**

```
r
where
  r = call(0,f) @ 5
  f = if odd(x) then x else next x fi;
  x = actualls(n);
  n = 1 fby n + 1;
end
```

**Figure 1c:**

```cpp
class r {
  // class members
  static Table obj_table;
  static r* get_obj(Context);
  // instance members
  Context context;
  Data value;
  void eval() {
    value = f::get_obj(Context(5, child(0,context(i))))->value;
  }
};

class f {
  ....
  void eval() {
    Data temp = (x::get_obj(context))->value;
  }
};
```
value = odd(temp)?temp:
   (x: get_obj(Context(context(0)+1, context(1))))->value;
}

class x {
   ....
   void eval()
   { switch child_no(context(1)) { case 0:
      value = n:get_obj(Context(context(0), parent(context(1))))->value;}}
};

class n {
   ....
   void eval()
   {value = context(0)===0?1:
      (n:get_obj(Context(context(0)-1, context(1))))->value + 1;}
};

3 Compiling Functions to Class Libraries

Compiling an intensional function to a runtime class library function is independent of the program context where it will be used. In this case, we need to let the compiled function class to build its program context – function call numbers and actuals lists, dynamically at the program execution time, instead of depending the static program analysis at compile time.

We use a class registration approach to accomplish it. We define a registration handler method for each function class. The method is used to handle registration requests from other function classes to this function class. We also define an initialization method for each function class. The method sends all the registration requests to the corresponding function classes. A registration request is generated from each function application in the definition of the function. During program execution, before eduction starts, we run the initialization method for each function class.

A function class keeps a registration count with initial value 0. When receiving a registration request, the registration handler increases the registration count and returns its value as the registration number of this request. Along with the registration request, the requesting function class also sends the function pointers to all the arguments of the registered function application. The registration handler of the function class then registers these arguments to the corresponding parameter classes with the registration number.

When receiving an argument pointer and the registration number, the registration handler of a parameter class adds the argument into its actuals list at the index given by the registration number.

After the initialization, each function call in the code is assigned a unique call number. Also, during eduction, this call number will be used as index to find the
corresponding argument in the parameter class’ actuals list.

Figure 2 shows the registration handler and initialization method for the function class \( f \) and the registration handler for the parameter class \( x \). Here we assume \( \text{odd}(x) \) is a user-defined function instead of a built-in operator.

Figure 2:

class f {

    ....
    int call_count;
    int register_handler(Data (*g)(Context))
    { x::register_handler(\+call_count, g); return call_count;}
    int odd_call[1];
    void init() {odd_call[0] = odd::register_handler(odd_arg_1);}
    void eval()
    {Data temp = (x::get_obj(context))->value;
      value =
      odd::get_obj(Context(context(0), child(odd_call[0],context(1))))?
      temp: (x::get_obj(Context(context(0)+1,context(1))))->value;}
    }
    Data odd_arg_1(Context context)
    {return x::get_obj(context)->value;}

class x {

    ....
    Data (* fp)(Context) actuals_list[];
    void register_handler(Data (* fp)(Context), int index)
    {actuals_list[index] = fp;}
    void eval()
    {value = actual_list[child_no(context(1))](context);}  
    
};

Thus we can first compile an intensional function to a class module which consists of the function class and parameter classes. We then use C++ compiler to compile the module into a binary class library.

We may consider an intensional program to be a composition of individually compiled library functions. To build a program, we only need a linker that links these functions together to make the program. The linker’s task is very simple. It creates a main executor function. The function imports all the library class modules needed by the program, and calls the initialization method for each function class, then starts eduction by sending demand to the object which will hold the result value of the program.
4 Concluding Remarks

To make intensional programming become a practical application programming system, we have to deal with the issue of software reuse. We need to build our software libraries and hence the API. In this paper we demonstrated that using the object-oriented approach it is feasible to accomplish this goal. In future work, we will optimize the sequential class libraries and develop parallel and distributed OO implementations.

References


