Integrating Functional Programming Into C++: Implementation and Verification

Rose Hafsah Abdul Rauf *

Department of Computer Science, University of Wales Swansea

Abstract. We describe a parser-translator program that translates typed λ-terms to C++ classes so as to integrate functional programming. We prove the correctness of the translation with respect to a denotational semantics using Kripke-style logical relations.

1 Introduction

C++ is a general purpose language that supports object oriented programming as well as procedural and generic programming, but unfortunately not functional programming. We have developed a parser-translator program that translates a simply typed λ-term to C++ statements so as to integrate functional progamming. This translated code uses the C++ object oriented concept of classes and inheritance in the definition of the λ-term. We build a mathematical model from the formal semantics of the translated code to prove its correctness. First, we give the denotational semantics of the typed λ-calculus. Then the correctness of the implementation of the typed λ-calculus by C++ classes is proved with respect to the denotational semantics. The correctness proof of the translated code is based on a Kripke-style of logical relation between the C++ class and the denotational model.

The parser-translator program that has been developed will parse a string represention of simply typed λ-term and translate it to a sequence of C++ statements. The translation of this λ-term will be discussed in the next section. How the translated code is executed will also be discussed along with the representation of the memory allocation. The mathematical model was based on the execution of the translated code. In building up this mathematical model, we will first give the denotational semantics of the typed λ-calculus. Then we will implement the C++ classes with the denotational semantics. These will be discussed in section 3. Some related and future work on integrating functional programing into C++ will be discussed at the end of this paper.

The approach of using denotational semantics and logical relation in proving the correctness of programs has been used before by researchers such as Plotkin[7], and many others. The method of logical relation can be traced back

* This paper is part of my PhD project and I would like to thank my supervisors Dr. Ulrich Berger and Dr. Anton Setzer for their knowledge and guidance making it possible for me to complete it.

at least to Tait[14] and has been used for a large variety of purposes (eg. Jung and Tiuryn[2], Statman[11] and Plotkin[6]). To our knowledge the verification of the implementation of λ-calculus in C++ using logical relation is new.

2 Translation

For the purpose of explaining how the λ-term is translated to its equivalent C++ statements and execution of the translated code, we will give an example of a λ-term input to the parser-translator program where the term must follow a syntax that has been determined. A λ-term $\lambda\, x^{\text{int}} \,.\, t$ where t is of the type `int` is written in our syntax as `\int x.int t .`
The statement shown below is the string that was entered to the parser-translator program:

```
int k =(\(int->int) f.\int x.int f^^(f^^x))^^(\int x. int x+2)^^3;
```

and it is equivalent to $k = (\lambda f^{(\text{int}\rightarrow\text{int})} \cdot \lambda x^{\text{int}} \cdot f(fx))(\lambda x^{\text{int}} \cdot x + 2)3$

First, the function type is defined as an abstract class with a virtual operator() method that will be overloaded in the definition of the λ-term and the type itself is the type pointers to an object of this abstract class. For the type class names we make use of letters `C` and `D` to represent open and close brackets respectively and an underscore for an arrow. For example, `Cint_intD` means (int→int). The concept of inheritance are involved in the definition of a λ-term where the function type abstract class will be the base class for the λ-term class.

The subterm `\int x.int f^^(f^^x)` in the statement above is translated as an instance of the class `Cint_intD_aux` ;

```
class lambda1 : public Cint_intD_aux{
 public :Cint_intD f;
 lambda1( Cint_intD f)  {   this-> f = f;};
 virtual int operator () (int x)
 { return (*(f))((*(f))(x)); };
};
```

The subterm `\(int->int) f.int x.int f^^(f^^x)` is translated as an instance of `CCint_intD_Cint_intDD_aux`:

```
class lambda0 : public CCint_intD_Cint_intDD_aux{
 public :
 lambda0( ) { };
 virtual Cint_intD operator () (Cint_intD f)
 { return new  lambda1( f); }
};
```

and the λ-term `\int x.int x+2` is translated as follows :

```
 class lambda2 : public Cint_intD_aux{
 public :
 lambda2( ) { };
 virtual int operator () (int x)
 { return x + 2; };
};
```

The term `k` will be finally translated as the expression :

```
int k = (*((*( new lambda0( )))( new lambda2( ))))(3);
```

The classes for the λ-terms are instantiated by statements new lambda0() and new lambda2() where pointers will be created that point to the addresses of the classes on the heap. The heap, which is also known as free store, is a dynamic store in the memory. Classes are created for each λ-term objects and each class has a pointer to its address on the heap. The evaluation for the expression above follows the call-by-value evaluation strategy (which will result in the value of 7). Note that the storage allocated for the instances of the classes are not deleted afterwards. The deletion depends on the garbage collection version of C++. One can also use smart pointers to enforce deletion.

3 Proof of correctness

Before we start building a mathematical model of the translated code, we list some of the mathematical preliminaries that will be frequently used in this section. The presentation of the proof follows the style of Winskel[15].

3.1 Mathematical preliminaries

Mappings

1. If X, Y are sets, then a list $m = (x_1 : y_1), \ldots, (x_n : y_n) \in$ list(X $\times$ Y) is considered as a finite map from X to Y which is defined as follows : If $x \in$ X, $y \in$ Y, then $m(x) := y$ where $x = x_i, y = y_i$ and $x \neq x_j$ for $j > i$.
2. We usually define dom(m) = the domain of $m = x_1, \ldots, x_n$.
   If $x \in$ X ,$y \in$ Y, then $m[x \mapsto y]$:= m, (x,y), the extension of the list m by (x,y). Note that dom($m[x \mapsto y]$) = dom(m) $\cup\{x\}$ and

$$m[x \mapsto y](x') = \begin{cases} y & \text{if } x' = x \\ m(x') & \text{if } x' \in \text{dom}(m)\backslash\{x\} \quad (x' \in \text{X}) \end{cases}$$

3.2 Implementation of the typed λ-calculus

a) **Types**
The set **Typ** of types is inductively given by :

i) Int $\in$ Typ

ii) if $A, B \in \mathrm{Typ}$, then $A \rightarrow B \in \mathrm{Typ}$

b)**Terms**
The **Terms** for the λ-calculus can be any of the following shown below.

i) $n \in \mathrm{N}$ (any number)
ii) $x \in \mathrm{Var}$ (where Var = String)
iii) $r\,s$ (term r is applied to term s)
iv) $\lambda x : A.r$ (term is an abstraction)
v) $f[r_1 \ldots r_n] = f[\boldsymbol{r}]$ ($f \in \mathcal{F}$ is a set of names for computable functions on N).The function denoted by f is written as $[\![f]\!]$

c) **Typing**
A **Context** $\boldsymbol{\Gamma}$ is a map from variables to types i.e. a list of variables and their type : **Context=list(Var× Typ)**
Context will be denoted as $\Gamma = x_1 : A_1, \ldots, x_n : A_n$
The **Typing** rules of the simply typed λ-calculus are :
i)

$$\frac{}{\Gamma, x : A \vdash x : A}$$

ii)

$$\frac{}{\Gamma \vdash n : \mathrm{Int}}$$

iii)

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x r : A \rightarrow B}$$

iv)

$$\frac{\Gamma \vdash r : A \rightarrow B \qquad \Gamma \vdash s : A}{\Gamma \vdash rs : B}$$

v)

$$\frac{f : \mathrm{Int} \times \ldots \times \mathrm{Int} \rightarrow \mathrm{Int} \qquad \Gamma \vdash r_1 : \mathrm{Int} \ldots \Gamma \vdash r_n : \mathrm{Int}}{\Gamma \vdash f[r_1, \ldots, r_n] : \mathrm{Int}}$$

d) **Denotational semantics**
The sets of **functionals** of type A denoted as $\mathrm{D}(A)$ are defined as follows :

i) D(Int)= N
ii) $\mathrm{D}(A \rightarrow B) = \{f | f : \mathrm{D}(A) \rightarrow \mathrm{D}(B)\}$
iii) $\mathrm{D} := \biguplus_{A \in \mathrm{Typ}} \mathrm{D}(A)$ where $\biguplus$ denotes disjoint union

A **Functional Environment** is a mapping of $\xi : \mathrm{Var} \rightarrow \mathrm{D}$. We let $\mathrm{FEnv} := \mathrm{Var} \rightarrow \mathrm{D}$ be the set of all functional environments. If Γ is a context, then $\xi : \Gamma$ means $\forall x \in \mathrm{dom}(\Gamma).\xi(x) \in \mathrm{D}(\Gamma(x))$.

For every typed λ-term $\Gamma \vdash r : A$ and every functional environment $\xi : \Gamma$ the denotational value $[\![r]\!]\xi \in \mathrm{D}(A)$ is defined as follows :

i) $[\![n]\!]\xi = n$
ii) $[\![x]\!]\xi = \xi(x)$

iii) $[\![r\ \ s]\!]\xi = [\![r]\!]\xi([\![s]\!]\xi)$
iv) $[\![\lambda x : A.r]\!]\xi(a) = [\![r]\!]\xi[x \mapsto a]$
v) $[\![f[\boldsymbol{r}]]\!] = [\![f]\!]([\![\boldsymbol{r}]\!]\xi)$

By an **implementation** of the typed λ-calculus we mean an (implementation of an) algorithm computing for every closed term r : Int the value $[\![r]\!] \in \mathrm{N}$.

3.3 Implementation by C++ classes

The classes that will be created depend on the λ-term that is being parsed, the more complex the term is the more level of classes will be created. When the class is instantiated, an address of the class will be stored on the heap, and further instantiation of other classes will create a stack of addresses on the heap with addresses of any variables which is bound to the classes (or λ-term).

Every class is instantiated by calling the constructor of the object i.e. the name of the class with or without any arguments.The body of a λ-abstraction is associated with an applicative term ($\in$App, below). The complete list of syntactic sets associated with the C++ classes is as follows:

- **Addr = Int**
  These are addresses of classes or variables on the heap.
- **Constr = String**
  Constructors are names of classes.
- **Val = Int + Addr**
  A value is either an integer or an address of a class or variables.
- **App = Int + Var +** $\mathcal{F}\times$ **list(App) + App** $\times$ **App + Constr** $\times$ **list(App)**
  Applicative terms representing bodies of λ-abstraction.
- **Abst = Var** $\times$ **Typ** $\times$ **Context** $\times$ **App**
  Abstractions consist of the variables and the type bound to the abstraction, and the context which is the list of variables and their types and the application.
- **Env = list(Var** $\times$ **Val)**
  An environment is the list of variables and their values.
- **Heap = list(Addr** $\times$ **Constr** $\times$ **list(Val))**
  The heap consists of list of addreses of constructors and their lists of values of the variables.
- **Class = list(Constr** $\times$ **Abst)**
  The class environment consists of list of constructors and their abstractions.

We assume that every $f \in \mathcal{F}$ is given by a side effect free C++ function

a) **The evaluation of λ-terms in C++**
When a λ-term is executed, a class address of the application of the λ-term is created on the heap and, with respect to the environment, a λ-term is evaluated to the value and an extended heap. This extended heap contains the address of

the value that has been evaluated for the λ-terms. Thus, the functionality of the evaluation function (**eval**) is :

$$\mathbf{eval} : \mathbf{Heap} \to \mathbf{Env} \to \mathbf{App} \to \mathbf{Val} \times \mathbf{Heap}$$

For a function application, where a λ-term is applied to another λ-term, the heap, which contains the class address of the two terms with the two values evaluated from the two terms, will evaluate to a value and an extended heap. Thus, the functionality of the application function (**apply**) is :

$$\mathbf{apply} : \mathbf{Heap} \to \mathbf{Val} \to \mathbf{Val} \to \mathbf{Val} \times \mathbf{Heap}$$

In the definition of the function **eval** and **apply** we fix some C:Class.
In presenting the evaluation rules we will follow the convention that :

- n ranges over numbers **N**
- x ranges over variables **Var**
- a, b range over application **App**
- v, w range over values **Val**
- k ranges over addresses **Addr**
- H ranges over **Heap**
- c ranges over constructors **Constr**
- C ranges over **Class**
- A, B range over **Typ**
- η ranges over **Env**

The metavariables we use to range over the syntactic categories can be primed or subscripted. For example, H, H', H'', H_k stand for heaps, C, C', C'' stand for classes and v_1, v' stand for values.

The rules for the evaluation of the λ-terms are as follows:

i) **Evaluation of a λ-term where application is a number:**

$$\mathrm{eval}\, H\, \eta\, n = (n, H)$$

ii) **Evaluation of a λ-term where application is a variable:**

$$\mathrm{eval}\, H\, \eta\, x = (\eta(x), H)$$

iii) **Evaluation of a λ-term where application is a function with a list of arguments:**

$$\mathrm{eval}\, H\, \eta\, f[\boldsymbol{a}] = ([\![f]\!](\boldsymbol{n}), H_k)$$

where $\boldsymbol{a} = a_1, \ldots, a_k, \boldsymbol{n} = n_1, \ldots, n_k$ and $\mathrm{eval}^*\, H\, \eta\, \boldsymbol{a} = (\boldsymbol{n}, H_k)$.
Here we define $\mathrm{eval}^*\, H\, \eta\, \boldsymbol{a} = (\boldsymbol{n}, H_k)$ if eval $H \eta\, a_1 = (n_1, H_1), \ldots,$ eval $H_{k-1}\, \eta\, a_k = (n_k, H_k)$. H_k is not changed by f because $f \in \mathcal{F}$ has no side effect.

iv) **Evaluation of a λ-term where the application is the application of one term to the other:**

$$\mathrm{eval}\, H\, \eta\, (a\ \ b) = \mathrm{apply}\, H'' v\, w = (v', H''')$$

where $\mathrm{eval}\, H\, \eta\, a = (v, H'),\ \mathrm{eval}\, H'\, \eta\, b = (w, H'')$.
The definition of **apply** in detail is shown as follows :

$$\mathrm{apply\ H}\, k\, v = \mathrm{eval}\, H\, [x, \boldsymbol{y} \mapsto v, \boldsymbol{w}]a$$

where $H(k) = (c, \boldsymbol{w})$, $C(c) = (x : A; \boldsymbol{y} : \boldsymbol{B}; a)$ (assuming $c \in \mathrm{dom}(C)$).

v) **Evaluation of a λ-term where the application is a constructor with a list of arguments:**

$$\mathrm{eval}\, H\, \eta\, c[\boldsymbol{a}] = (k, H'[k \mapsto c[\boldsymbol{v}]]) \quad (k \in \mathbf{Addr}, v \in \mathbf{Val})$$

where eval* $H\, \eta\, \boldsymbol{a} = (\boldsymbol{v}, H')$ and $k =$ new H' (new H' is an address not in $\mathrm{dom}(H')$).

In all other cases for the application, it is termed invalid and an error will be returned.

Lemma 1. *1. eval* $H\, \eta\, a = (v, H') \Longrightarrow H \subseteq H'$
2. apply $H\, v\, w = (v', H') \Longrightarrow H \subseteq H'$
*3. eval** $H\, \eta\, \boldsymbol{a} = (\boldsymbol{n}, H') \Longrightarrow H \subseteq H'$

The proof for Lemma 1 is by induction on the definition of **eval** and **apply**.

Note that, since **eval** and **apply** depend on C:Class, the true signatures of eval and apply are as follows :
eval : Class→Heap→Env→App→Val×Heap
apply : Class→Heap→Val→Val→Val×Heap
We write $\mathbf{eval}_C\, H\, \eta\, a$ and $\mathbf{apply}_C\, H\, v\, w$ if the argument C:Class is to be made explicit.

b) **The Parsing of a λ Term**

Traditionally, a λ-term that is input is parsed as a long string which will undergo several steps of parsing to get the translated code. The parsing will create classes for the λ-term where in the case of a complex λ-term it will create several levels of classes where the class of an upper level is an extension of the lower level class. In order to simplify things and to concentrate on the most important aspects of the problem we assume that the input is given as an abstract term rather than a string. The parsing from a string to a term is a traditional parsing problem which is of no interest here. What is interesting here is the process of creating a system of C++ classes that represents a λ-term.

In order to give a recursive description of this process, we must assume that the term in question is not the first term being parsed, but other terms (or subterms) have been parsed before having created a system of classes. Furthermore, if the term has free variables, then the types of these variables must be fixed by an appropriate context. Therefore, the parser **P** has the following functionality :

$$\mathbf{P} : \mathbf{Class} {\rightarrow} \mathbf{Context} {\rightarrow} \mathbf{Term} \rightarrow \mathbf{App} {\times} \mathbf{Class}$$

The rules for the parsing are as follows :

i) **Parsing when the term is a number:** $\mathrm{P}_C \mathit{\Gamma} n = (n, C)$
ii) **Parsing when the term is a variable:** $\mathrm{P}_C \mathit{\Gamma} x = (x, C)$
iii) **Parsing when the term is a function with a list of arguments:**

$$\mathrm{P}_C \mathit{\Gamma} f[\boldsymbol{r}] = (f[\boldsymbol{a}], C')$$

where $\mathrm{P}^*{}_C \mathit{\Gamma} \boldsymbol{r} = (\boldsymbol{a}, C')$ and P* is defined in a smilar way as eval*.
iv) **Parsing of an application:** $\mathrm{P}_C \mathit{\Gamma}(r\ \ s) = (a\ \ b, C'')$
where $\mathrm{P}_C \mathit{\Gamma} r = (a, C'),\ \mathrm{P}_{C'} \mathit{\Gamma} s = (b, C'')$
v) **Parsing of a λ abstraction :** $\mathrm{P}_C \mathit{\Gamma}(\lambda x : A.r) = (c[\boldsymbol{y}], C'[c \mapsto (x : A; \mathit{\Gamma}; a)])$
where $\boldsymbol{y} = \mathrm{dom}(\mathit{\Gamma}), \mathrm{P}_C \mathit{\Gamma}[x \mapsto A] r = (a, C'),$ and $c =$ new C'
meaning that c is a name of a class that is "new" i.e. has not been used before.

Remark: We only generate $c[\boldsymbol{x}] \in$ App with $\boldsymbol{x} \in$ list(Var) and not $c[\boldsymbol{a}]$ with arbitary $\boldsymbol{a} \in$ list(App)

Lemma 2. *i)* $P_C \mathit{\Gamma} r = (a, C') \Longrightarrow C \subseteq C'$
ii) $P^*{}_C \mathit{\Gamma} \boldsymbol{r} = (\boldsymbol{a}, C') \Longrightarrow C \subseteq C'$

The proof for Lemma 2 is by induction on r respectively $\boldsymbol{r}$.

3.4 The correctness of the translated code

The correctness proof of the translated code is based on a Kripke-style relation between the C++ representation of the term ($\in$ Val × Heap) and its denotational value ($\in \mathrm{D}(A)$). The relation is indexed by the class environment C and the type A of the term. Since in the case of an arrow type, $A \rightarrow B$, extensions of H and C have to be taken into account, this definition has some similarity with Kripke models. The relation

$$\sim^C_A \subseteq (\mathrm{Val} \times \mathrm{Heap}) \times \mathrm{D}(A) \text{ where } A \in \text{ Typ }, C \in \mathrm{Class}$$

is defined by recursion on A as follows:

$$\begin{aligned} (v, H) \sim^C_{\mathrm{Int}} n &:\Longleftrightarrow v = n \\ (v, H) \sim^C_{A \rightarrow B} f &:\Longleftrightarrow \forall C \subseteq C', \forall H \subseteq H', \forall (w, d) \in \mathrm{Val} \times \mathrm{D}(A) : \\ &\qquad (w, H') \sim^{C'}_A d \Longrightarrow \mathrm{apply}_{C'} H' v w \sim^{C'}_B f(d) \end{aligned}$$

We also set $(\eta, H) \sim^C_{\mathit{\Gamma}} \xi := \forall x \in \text{ dom } \mathit{\Gamma}(\eta(x), H) \sim^C_{\mathit{\Gamma}(x)} \xi(x) \in \mathrm{D}(\mathit{\Gamma}(x))$.

Lemma 3.

$$(v, H) \sim_A^C d, C \subseteq C', H \subseteq H' \Longrightarrow (v, H') \sim_A^{C'} d$$

The proof for Lemma 3 is by induction on A.

Our main theorem, which corresponds to the usual "Fundamental Lemma" or "Adequacy Theorem" for logical relations, reads as follows:

Theorem 1. *If* $\eta : Env, \xi : FEnv, \Gamma \vdash r : A, \xi : \Gamma, P_C \Gamma r = (a, C'), C' \subseteq C'', (\eta, H) \sim_\Gamma^{C''} \xi$, *and* $H \subseteq H'$, *then* $eval_{C''} H' \eta a \sim_A^{C''} [\![r]\!] \xi$.

The theorem can be proved by an induction on the typing judgement $\Gamma \vdash r : A$ using the Lemma 1-3 above. Due to limited space we omit details.

For a closed term r , we define $\mathrm{P}r = \mathrm{P}_\emptyset \emptyset r$.

Corollary 1 ((Correctness of the implementation)). *If* $\vdash r : Int, Pr = (a, C), C \subseteq C'$, *then for any heap* H, $eval_{C'}\, H\, \eta\, a = ([\![r]\!], H')$ *for some* $H' \supseteq H$.

4 Conclusion

The aim of this paper was to introduce a new approach of integrating functional programming into C++ and to show a method of proving the correctness of the translation code produced by denotational semantics and logical relation. In the past, several researches [3],[4] discovered that C++ can be used for functional programming by representing first class functions and higher order functions using classes, and by this technique we produced the translated code. There are other approaches that have made C++ a language that can be used for functional programming such as FC++ library [5] (a very elaborate approach), FACT! [13] (extensive use of templates and overloading) and [3] (creating macros that allow creation of single macro-closure in C++). The advantages of our solution are that it is very simple, it uses classes and inheritance in an essential way and, most importantly, we have a formal correctness proof.

In addition to the mathematical proof given in this paper, the correctness of the translated code produced by the parser-translator program has been verified by testing it with several types of λ-term from simple to complex ones.

Future work. This work can be extended by integrating lazy constructors (infinite structures) and lazy evaluation, having terms with side effect and integrating recursive higher order functions [1].

References

1. Abdul Rauf R.H., Berger U., Setzer A.: Functional Concepts in C++, To appear in Proceedings of TFP 2006, http://www.cs.nott.ac.uk/∼nhn/TFP2006.

2. Jung A., Tiuryn J.: A New Characterization of Lambda Definability. Typed Lambda Calculus and Applications, 1993.
3. Kiselyov O.: Functional Style in C++ : Closures, Late Binding, and Lambda Abstraction. Poster presentation, Int. Conf. on Functional Programming, 1998.
4. Läufer K.: A Framework For Higher Order functions in C++. Proc. Conf. Object Oriented Technologies(COOTS), Monterey, C.A., June 1995.
5. McNamara B., Smaragdakis Y.: Functional Programming in C++. ICFP '00, Montreal Canada,ACM Press, 2000.
6. Plotkin G. D.: Lambda Definability in the Full Type Hierarchy. To H.B. Curry; Essays on Combinatoric Logic, Lambda Calculus and Formalism., J.P.Seldin , J.R. Hindley, eds., 363-373, 1980.
7. Plotkin G. D.: LCF Considered As a Programming Language. Theoretical Computer Science, 5:223-255, 1977.
8. Polak W.: Program Verification Based On Denotational Semantics. Proceedings of the 8^{th} ACM, SIGPLAN-SIGACT Symposium on Principles Of Programing Analysis. ACM Press, Jan. 1981.
9. Scott, D., Strachey, C.: Mathematical Semantics For Computer Language. Tech. Monograph PRG-6, Programming Research Group, University Of Oxford, 1971.
10. Setzer A.: Java as a Functional Programming Language. Types for Proofs and Programs: International Workshop, Types 2002, Berg en Dal,April 24-28,2002. Selected Papers, Geuver H., Wiedijk F., eds, 279-298, LNCS 2646, 2003.
11. Statman R.: Logical Relation and the Typed λ Calculus. Information and Control, 65:85-97, 1985.
12. Stoy, J: Denotational Semantics - The Scot-Strachey Approach To Language Theory. MIT Press, Cambride, 1977.
13. Striegnitz J. : FACT!-The Functional Side of C++. http://www.fz-juelich.de/zam/FACT.
14. Tait W.: Intentional Intrepretation of Funtional of Finite Type I. Journal Of Symbolic Logic, 32(2):198-212, 1967.
15. Winskel, G. : The Formal Semantics Of Programming Languages : an Introduction. Massachusetts Institute Of Technology, 1993.