Functional Concepts in C++

Rose H. Abdul Rauf, Ulrich Berger, Anton Setzer*

Department of Computer Science, University of Wales Swansea
Email: rosehafsah@yahoo.co.uk, {u.berger,a.g.setzer}@swansea.ac.uk

Abstract

We describe a parser-translator program that translates typed λ-terms into C++ classes so as to integrate functional concepts. We prove the correctness of the translation of λ-terms into C++ with respect to a denotational semantics using Kripke-style logical relations. We introduce a general technique for introducing lazy evaluation into C++, and illustrate it by carrying out in C++ the example of computing the Fibonacci numbers efficiently using infinite streams and lazy evaluation. Finally, we show how merge higher-order λ-terms with imperative C++ code.

1 INTRODUCTION

C++ is a general purpose language that supports object oriented programming as well as procedural and generic programming, but unfortunately not directly functional programming. We have developed a parser-translator program that translates typed λ-term into C++ statements so as to integrate functional concepts. This translated code uses the object oriented approach of programming that involves creation of classes for the λ-term. That a translated λ-term is an element of the translation of a function type is achieved by using inheritance.

The paper is organised as follows: First, we introduce the translation and discuss how the translated code is executed including a description of the memory allocation (Sect. 2). The correctness of our implementation is proved with respect to the usual (set-theoretic) denotational semantics of the simply typed λ-calculus on the one hand, and a mathematical model of a suffi ciently large fragment of C++ on the other hand. The proof is based on a Kripke-style logical relation between the C++ class and the denotational model (Sect. 3). In Sect. 4 we introduce a general technique for introducing lazy evaluation into C++ by introducing a data type of lazy elements of type *A*. Finally, in Sect. 5 we discuss a few general features arising if we allow λ-terms in C++ to have side-effects.

Related work. Several researchers [Kis98], [Läu95] have discovered that C++ can be used for functional programming by representing higher order functions using classes. Our representation is based on similar ideas. There are other approaches that have made C++ a language that can be used for functional programming such as the FC++ library [MS00] (a very elaborate approach) as well as FACT! [Str] (extensive use of templates and overloading) and [Kis98] (creating macros that allow creation of single macro-closure in C++). The advantages of our

*Supported by EPSRC grant GR/S30450/01.

solution are that it is very simple, it uses classes and inheritance in an essential way, it can be used for implementing λ-terms with side-effects, and, most importantly, we have a formal correctness proof.

The approach of using denotational semantics and logical relations in a proof of the correctness of programs has been used before by Plotkin [Plo77] and others. The method of logical relations can be traced back at least to Tait [Tai67] and has been used for various purposes (e.g. Jung and Tiuryn [JT93], Statman [Sta85] and Plotkin [Plo80]). To our knowledge the verification of the implementation of the λ-calculus in C++ using logical relations is new.

Lazy evaluation in C++ has been studied in the literature (see [Sch00], [MS00], [Kel97]). To our knowledge all implementations are restricted to lazy lists, whereas we introduce a general type of lazy elements of an arbitrary type.

2 TRANSLATION OF TYPED λ-TERMS INTO C++

In this section we describe how to translate simply typed λ-terms into C++ using the object-oriented concepts of classes and inheritance.

The *simply typed λ-calculus* over the base type of integers with constants for arithmetic functions, *λ-calculus* for short, is given as follows: *Types* are built from the base type, Int, and and function types, $A \to B$. *Terms* are of the form x (variables), n (numerals $\in \mathrm{N} = \{0,1,2,,\ldots\}$), $\lambda x^A r$ (abstraction), $r\ s$ (application), $f[r_1,\ldots,r_n]$ (function application). In the last case f is taken from a set F of names of number-theoretic C++ functions. A *context* is a finite set of pairs $\Gamma = x_1 : A_1,\ldots,x_n : A_n$ (all x_i distinct) which is, as usual, identified with a finite map. We let Type, Var, Term, Context denote the set of types, variables, terms and contexts, respectively. The *typing rules* are as expected:

$$\Gamma, x : A \vdash x : A \qquad \Gamma \vdash n : \mathrm{Int}$$

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x^A r : A \to B} \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B}$$

$$\frac{\Gamma \vdash r_1 : \mathrm{Int} \ldots \Gamma \vdash r_k : \mathrm{Int}}{\Gamma \vdash f[r_1,\ldots,r_k] : \mathrm{Int}} \quad (f \text{ a } k\text{-ary arithmetic function})$$

Rather than explaining in general how a λ-term is translated into its equivalent C++ statements, we describe the translation of the example term

$$t = (\lambda f^{\mathrm{Int}\to\mathrm{Int}} \lambda x^{\mathrm{Int}}.f\ (f\ x))\ (\lambda x^{\mathrm{Int}}.x+2)\ 3 \ \ .$$

The concrete ASCII notation for this term (i.e. the input for the parser program) is

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

For each function type we first define an abstract class with a virtual operator that will be overloaded in the definition of the λ-term. The type itself is the type of pointers to an object of this class. In the class names the letters `C` and `D` are used to represent opening and closing brackets and an underscore stands for an arrow. Hence `Cint_intD` means Int $\to$ Int. The type of t, $(\mathrm{Int} \to \mathrm{Int}) \to (\mathrm{Int} \to \mathrm{Int})$, is represented in stages:[1]

```
class Cint_intD_aux
{ public : virtual int operator() (int x) = 0; };

typedef Cint_intD_aux*  Cint_intD;

class CCint_intD_Cint_intDD_aux
{  public : virtual Cint_intD operator()
                             (Cint_intD x) = 0; };

typedef CCint_intD_Cint_intDD_aux*
        CCint_intD_Cint_intDD;
```

The term t is translated in stages as well. The subterm $\lambda x^{\mathrm{Int}}.f\ (f\ x)$ 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 $\lambda f^{\mathrm{Int}\to\mathrm{Int}}\lambda x^{\mathrm{Int}}.f(fx)$ is translated as follows:

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

The other λ-term, $\lambda x^{\mathrm{Int}}.2+x$, is translated in a similar way:

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

The defining equation for t will be finally translated into the expression

[1]In order to obtain fast compilation of the translated code, we do not use templates in our machine-generated code. This will as well ease the correctness proof to be given in Sect. 3. If one defines λ-types by hand it is of course easier to define a general C++-template for the class corresponding to the type $X \to Y$.

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

When evaluating the expression t, first the application of lambda0 to lambda2 is evaluated. For this first instances l0, l2 of the classes lambda0 and lambda2 are created. Then the operator()-method of l0 is called. That call will create an instance l1 of lambda1, with the instance variable f set to l2. The result of application of lambda0 to lambda2 is l1.

The next step in the evaluation of t is to evaluate 3, which evaluates to 3, and then to call the operator() method of l1. This will first make a call to the operator method of f, which is bound to l2, and apply it to 3. This will evaluate to 5. Then it will call the operator method of f again, which is still bound to l2, and apply it to the result 5. The result returned is 7.

We see that the evaluation of the expression above follows the call-by-value evaluation strategy. Note that l0, l1, l2 were created on the heap, but have not been deleted afterwards. The deletion of l0, l1 and l2 relies on the use of a garbage collected version of C++, otherwise we could use smart pointers in order to enforce their deletion.

3 PROOF OF CORRECTNESS

We sketch how to prove the correctness of our C++ implementation of the λ-calculus. By "correctness" we mean that every closed term r of type Int is evaluated by our implementation to a numeral which coincides with the *value* of r. The value of a term can be defined either *operationally* as the normal form w.r.t. β-reduction, $(\lambda x^A r)s \longrightarrow_\beta r[s/x]$, and function reduction, $f[n_1,\ldots,n_k] \longrightarrow_f n$ (n the value of f at $n_1,\ldots,n_k$), or, equivalently, *denotationally* as the natural value in a suitable domain of functionals of finite types. Since our calculus doesn't allow for recursive definitions, the details of the operational and denotational semantics do not matter: Operationally, any sufficiently complete reduction strategy (call-by-value, call-by-name, full normalisation) will do, and denotationally, any Cartesian closed category containing the type of integers can be used. For our purposes it will be most convenient to work with a denotational model. For simplicity we take the naive set-theoretic hierarchy D of functionals of finite types over the integers (setting $\mathrm{N} = \{0,1,2,\ldots\}$ and $X \to Y = \{f \mid f : X \to Y\}$):

$$\mathrm{D}(\mathrm{Int}) = \mathrm{N}, \quad \mathrm{D}(A \to B) = \mathrm{D}(A) \to \mathrm{D}(B), \quad D = \bigcup_{A \in \mathrm{Type}} \mathrm{D}(A)$$

A *functional environment* is a mapping $\xi : \mathrm{Var} \to \mathrm{D}$. FEnv denotes 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) $[\![f[\vec{r}]]\!] = [\![f]\!]([\![\vec{r}]\!]\xi)$

where in the last clause $[\![f]\!]$ is the number-theoretic function denoted by f. Our implementation of the λ-calculus is modelled in a similar way as e.g. in [ASS85] using functions eval and apply. In order to model the C++ implementation as truthfully as possible, we will introduce a simplified model of C++, in which we will make the pointer structures for the classes and objects explicit. The functions eval and apply modify these pointer structures via side effects.

In our model all classes will have instance variables, one constructor, and one method corresponding to the operator() method. The constructor has one argument for each instance variable, and will set the instance variables to these arguments. No other code is performed. The method has one argument, and the body consists of an applicative term, where applicative terms are simplified C++ expressions in our model. So, a class is given by a context representing its instance variables, the abstracted variable of the method and its type, and an applicative term.

Applicative terms will be numbers, variables, function terms applied to applicative terms, the application of one applicative term to another applicative term (which corresponds to the method call in case the first applicative term is an object), or a constructor applied to applicative terms.

When a constructor call of a class is evaluated, its arguments are first evaluated. Then, memory for the instance variables of this class will be allocated on the heap, and these instance variables will be set to the evaluated arguments. The address to this memory location is the result returned by evaluating this constructor call. The only other possible result of the evaluation of an applicative term is a number, so values are addresses or numbers.

The data sets associated with our model of C++ classes are defined as follows (letting $X+Y$ and $X \times Y$ denote the disjoint sum and Cartesian product of X and Y, X^* the set of finite lists of elements in X and $X \to_{\text{fin}} Y$ the set of finite maps from X to Y):

Addr	=	a set of numbers denoting addresses of classes on the heap
Constr	=	a set of strings denoting constructors, i.e. class names
Val	=	$\mathrm{N} + \mathrm{Addr}$
F	=	a set of names for arithmetic C++ functions
App	=	$\mathrm{N} + \mathrm{Var} + \mathrm{F} \times \mathrm{App}^* + \mathrm{App} \times \mathrm{App} + \mathrm{Constr} \times \mathrm{App}^*$
Context	=	$\mathrm{Var} \to_{\text{fin}} \mathrm{Type}$
Class	=	$\mathrm{Context} \times \mathrm{Var} \times \mathrm{Type} \times \mathrm{App}$
VEnv	=	$\mathrm{Var} \to_{\text{fin}} \mathrm{Val}$
Heap	=	$\mathrm{Addr} \to_{\text{fin}} \mathrm{Constr} \times \mathrm{Val}^*$
CEnv	=	$\mathrm{Constr} \to_{\text{fin}} \mathrm{Class}$

We write applicative terms ($\in$ App) that are neither numbers nor variables as $f[\vec{a}]$, $a\;b$ and $c[\vec{a}]$. Classes ($\in$ Class) are written as $(\Gamma; x : A; r)$.

The fact that the parsing function as well as the functions eval and apply have side effects on the classes and the heap can be conveniently expressed using the familiar *state monad*

$$\mathrm{M}_X(Y) := X \to Y \times X$$

Elements of $\mathrm{M}_X(Y)$ are usually called *actions* and can be viewed as elements of Y that may depend on a current state $x \in X$ and also may change the current state. Monads are a category-theoretic concept whose computational significance was discovered by Moggi [Mog91]. The functionalities of the parsing function P and the operations eval and apply can now be written as

$$\begin{aligned}
\mathrm{P} :&\quad \mathrm{Context} \to \mathrm{Term} \to \mathrm{M}_{\mathrm{CEnv}}(\mathrm{App}) \\
\mathrm{eval} :&\quad \mathrm{CEnv} \to \mathrm{VEnv} \to \mathrm{App} \to \mathrm{M}_{\mathrm{Heap}}(\mathrm{Val}) \\
\mathrm{apply} :&\quad \mathrm{CEnv} \to \mathrm{Val} \to \mathrm{Val} \to \mathrm{M}_{\mathrm{Heap}}(\mathrm{Val})
\end{aligned}$$

Hence, parsing has a side effect on the class environment, while eval and apply have side effects on the heap.

Strictly speaking, the definitions above (and below) are not quite accurate since, for example, applicative terms ($\in$ App) are untyped, but are supposed to represent valid (and hence typed) C++ expressions. A corresponding typing discipline could easily be introduced, but would probably be more distracting than insightful. Similarly, actions as well as the functions P, eval and apply should be modelled by partial rather than total functions. For example, in order for an expression of the form apply $C\;v\;w$ to make sense the value, v, must not be a numeral, n, but an address, h, in the heap (pointing to the C++ implementation of a λ-abstraction). The defining equation for apply reflects the assumption that this is the case. Similar soundness assumptions are tacitly made elsewhere. In case these assumptions are violated it is assumed that the computation results in an error. Again, the error cases could be modelled by a suitable monad, and one could show that, if we refer to correctly typed C++ expressions, this error never occurs. For the sake of brevity, we refrain from carrying this out. There is as well another reason for partiality, namely the fact that C++ programs can be recursive and therefore evaluation of C++ programs, even if correctly typed, might not terminate. However, it is a consequence of Theorem 3.1 that the evaluation of parsed λ-terms terminates and does not result in an error.

Also note that the use of monads for describing the C++ implementation is just a notational convenience. We do not make any assumptions of monadic structures to exist in C++.

We use the following standard monadic notation (roughly following Haskell syntax): Suppose $e_1 : \mathrm{M}_X(Y_1), \ldots, e_{k+1} : \mathrm{M}_X(Y_{k+1})$ are actions where e_i may depend on $y_1 : Y_1, \ldots, y_{i-1} : Y_{i-1}$. Then

$$\mathrm{do}\{y_1 \leftarrow e_1 \;;\; \ldots \;;\; y_k \leftarrow e_k \;;\; e_{k+1}\} : \mathrm{M}_X(Y_{k+1})$$

is the action that maps any state $x_0 : X$ to (y_{k+1}, x_{k+1}) where $(y_i, x_i) = e_i\ x_{i-1}$, for $i = 1, \dots, k+1$. If y_i does not occur in any e_j with $j > i$, then we suppress "$y_i \leftarrow$". We also allow let-expressions with pattern matching within a do-construct. Furthermore, we use

$$\begin{array}{ll}
\mathrm{return} : Y \to \mathrm{M}_X(Y) & \mathrm{return}\ y\ x = (y, x) \\
\mathrm{mapM} : (Z \to \mathrm{M}_X(Y)) \to Z^* \to \mathrm{M}_X(Y^*) & \mathrm{mapM}\ f\ \vec{a} = \mathrm{do}\{y_1 \leftarrow f\ a_1\ ;\ \dots \\
 & \quad \dots\ ;\ y_k \leftarrow f\ a_k\ ;\ \mathrm{return}\ (y_1, \dots, y_k)\} \\
\mathrm{get} : \mathrm{M}_X(X), & \mathrm{get}\ x = (x, x) \\
\mathrm{put} : X \to \mathrm{M}_X(\{*\}) & \mathrm{put}\ x\ x' = (*, x)
\end{array}$$

With these notations the definitions of P, eval and apply read as follows (we use a function fresh with the property that if $m : X \to_{\mathrm{fin}} Y$, where X is infinite, then $\mathrm{fresh}(m) \in X \setminus \mathrm{dom}(m)$):

$$\begin{array}{rcl}
\mathrm{P}\,\Gamma\,u & = & \mathrm{return}\ u,\ \text{if } u \text{ is a numeral or a variable} \\
\mathrm{P}\,\Gamma\ f[\vec{r}] & = & \mathrm{do}\{\vec{a} \leftarrow \mathrm{mapM}\ (\mathrm{P}\,\Gamma)\ \vec{r}\ ;\ \mathrm{return}\ f[\vec{a}]\} \\
\mathrm{P}\,\Gamma\ (r\ s) & = & \mathrm{do}\{(a, b) \leftarrow \mathrm{mapM}\ (\mathrm{P}\,\Gamma)\ (r, s)\ ;\ \mathrm{return}\ (a\ b)\} \\
\mathrm{P}\,\Gamma\ (\lambda x^A.r) & = & \mathrm{do}\{a \leftarrow \mathrm{P}\,\Gamma[x \mapsto A]\ r\ ;\ C \leftarrow \mathrm{get}\ ;\ \mathrm{let}\ c = \mathrm{fresh}(C)\ ; \\
 & & \quad \mathrm{put}(C[c \mapsto (\Gamma; x : A; a)])\ ;\ \mathrm{return}(c[\mathrm{dom}(\Gamma)])\}
\end{array}$$

$$\begin{array}{rcl}
\mathrm{eval}\ C\ \eta\ n & = & \mathrm{return}\ n \\
\mathrm{eval}\ C\ \eta\ x & = & \mathrm{return}\ (\eta\ x) \\
\mathrm{eval}\ C\ \eta\ f[\vec{a}] & = & \mathrm{do}\{\vec{n} \leftarrow \mathrm{mapM}\ (\mathrm{eval}\ C\ \eta)\ \vec{a}\ ;\ \mathrm{return}\ [\![f]\!](\vec{n})\} \\
\mathrm{eval}\ C\ \eta\ (a\ b) & = & \mathrm{do}\{(v, w) \leftarrow \mathrm{mapM}(\mathrm{eval}\ C\ \eta)\ (a, b)\ ;\ \mathrm{apply}\ C\ v\ w\} \\
\mathrm{eval}\ C\ \eta\ c[\vec{a}] & = & \mathrm{do}\{\vec{v} \leftarrow \mathrm{mapM}\ (\mathrm{eval}\ C\ \eta)\ \vec{a}\ ;\ H \leftarrow \mathrm{get}\ ;\ \mathrm{let}\ h = \mathrm{fresh}(H)\ ; \\
 & & \quad \mathrm{put}(H[h \mapsto (c, \vec{v})])\ ;\ \mathrm{return}(h)\} \\
\mathrm{apply}\ C\ h\ v & = & \mathrm{do}\{H \leftarrow \mathrm{get}\ ;\ \mathrm{let}\ (c, \vec{w}) = H\ h\ ;\ \mathrm{let}\ (\vec{y} : \vec{B}; x : A; a) = C\ c\ ; \\
 & & \quad \mathrm{eval}\ C\ [\vec{y}, x \mapsto \vec{w}, v]\ a\}
\end{array}$$

The correctness proof of the translated code is based on a Kripke-style relation between the C++ representation of a term ($\in \mathrm{Val} \times \mathrm{Heap}$) 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, $A \to B$, extensions of the heap and the class environment have to be taken into account, this definition has some similarity with Kripke models. The relation

$$\sim_A^C \subseteq (\mathrm{Val} \times \mathrm{Heap}) \times \mathrm{D}(A),$$

where $A \in \mathrm{Type}, C \in \mathrm{CEnv}$, is defined by recursion on A as follows:

$$\begin{array}{rcl}
(v, H) \sim_{\mathrm{Int}}^C n & :\Longleftrightarrow & v = n \\
(v, H) \sim_{A \to B}^C f & :\Longleftrightarrow & \forall C \subseteq C', \forall H \subseteq H', \forall (w, d) \in \mathrm{Val} \times \mathrm{D}(A) : \\
 & & (w, H') \sim_A^{C'} d \Longrightarrow \mathrm{apply}\ C'\ v\ w\ H' \sim_B^{C'} f(d)
\end{array}$$

We also set $(\eta, H) \sim^{C}_{\Gamma} \xi :\Longleftrightarrow \forall x \in \mathrm{dom}(\Gamma)(\eta(x), H) \sim^{C}_{\Gamma(x)} \xi(x)$.

The main result below corresponds to the usual ”Fundamental Lemma” or ”Adequacy Theorem” for logical relations:

Theorem 3.1 *Assume* $\eta : \mathrm{VEnv}$, $\xi : \mathrm{FEnv}$, $\Gamma \vdash r : A$. *Assume* $\xi : \Gamma$, $\mathrm{P}\,\Gamma\, r\, C = (a, C')$, $C' \subseteq C''$, $(\eta, H) \sim^{C''}_{\Gamma} \xi$ *, and* $H \subseteq H'$. *Then* $\mathrm{eval}\, C''\, \eta\, a\, H' \sim^{C''}_{A} [\![r]\!]\xi$.

The theorem can be proven by induction on the typing judgement $\Gamma \vdash r : A$. Due to limited space we omit details.

Corollary 3.2 (Correctness of the implementation) *Assume* $\vdash r : \mathrm{Int}$, $\mathrm{P}\; \emptyset\; r\; \emptyset = (a, C)$ *and* $C \subseteq C'$. *Then for any heap* H *and environment* η *we have* $\mathrm{eval}\, C'\, \eta\, a\, H = ([\![r]\!], H')$ *for some* $H' \supseteq H$.

4 LAZY EVALUATION IN C++

Haskell is famous for its programming techniques using infinite lists. A well-known example are the Fibonacci numbers, which are computed efficiently by using the following code:

```
fib = 1:1:(zipWith (+) fib (tail fib))
```

This example requires that we have infinite streams of natural numbers, and relies heavily on lazy evaluation. We will show how to translate this code into efficient C++ code. This requires that we are able to deal with lazy evaluation.

The standard technique for replacing call-by-value by call-by-name is to replace types A by $() \to A$ where $()$ is the empty type (i.e. `void`). This delays execution, but does not cater for the reuse of evaluated expressions, so that an expression is only evaluated once. In order to obtain this, we will define a new type `Lazy(A)`. This type delays evaluation of an element of type A in such a way that, if needed, the evaluation is carried out – however, only once. Once the value is computed, the result is stored in a variable for later reuse. The definition is as follows (we make use of the extended C++ syntax introduced in Sect. 2, especially `r ^^ t` for functional application, \ for λ, and `A -> B` for the type of functions from `A` to `B`):

```
template<typename X> class lazy{
  bool is_evaluated;
  union {X       result;
         () -> X compute_function;};
public:
  lazy(()-> X compute_function){
    is_evaluated = false;
    this->compute_function = compute_function;};
  X eval(){
    if (not is_evaluated){
            result = compute_function ^^ ();
            is_evaluated = true;};
    return result;};};
#define Lazy(X) lazy<X>*
```

Using this class we can now easily define lazy streams of natural numbers (lazy lists, i.e. possibly terminating streams, can be defined similarly, but require the usual technique based on the composite design pattern for formalising algebraic data types as classes by introducing a main class for the main type which has subclasses for each constructor, each of which stores the arguments of the constructor)

```
template<typename X>class lazy_stream{
public: Lazy(X) head;
        Lazy(lazy_stream<X>*) tail;
        ... Constructor as usual ... }
#define Lazy_Stream(X) lazy_stream<X>*
```

In order to deal with the example of the Fibonacci numbers, one needs to define the operators used in the above mentioned definition of `fib`:

- `lazy_cons<X>` computes the cons-operation on streams:

  ```
  Lazy(Lazy_Stream(X)) lazy_cons<X>
          (Lazy(X) head, Lazy(Lazy_Stream(X)) tail)
  ```

- `lazy_tail<X>` computes the tail of a stream lazily:

  ```
  Lazy(Lazy_Stream(X)) lazy_tail<X>
                       (Lazy(Lazy_Stream(X)) s)
  ```

- `lazy_zip_with<X>` computes the usual zip_with function (i.e. zip_with$(f,[a,b,..],[c,d,..]) = [f\ a\ c, f\ b\ d, \ldots]$):

  ```
  Lazy(Lazy_Stream(X)) lazy_zip_with<X>
     (X -> X -> X f,
      Lazy(Lazy_Stream(X)) s0,
      Lazy(Lazy_Stream(X)) s1)
  ```

The definition of these operation is straightforwards, once one has introduced a few combinators for dealing with `Lazy(X)`.

Now we can define the stream of Fibonacci numbers as follows (`plus` is $\lambda x,y.x+y$, `one_lazy` is the numeral 1 converted into an element of `Lazy(int)`, `create_lazy` transforms elements of type `()->A` into `Lazy(A)`, and `eval` evaluates an element of type `Lazy(A)` to an element of type `A`):

```
()-><Lazy_Stream(int)> fib_aux =
 \() x. Lazy_Stream(int)
        eval(
         lazy_cons(
           one_lazy,
           lazy_cons(
```

```
                  one_lazy,
                  lazy_zip_with(
                    plus,
                    create_lazy(this),
                    lazy_tail(create_lazy(this))))));
Lazy_Stream(int) fib = eval(create_lazy(fib_aux))
```

Note that here we were using the keyword `this` in the definition of `fib_aux`. This is how a recursive call should be written. If we instead put `fib_aux` here, C++ will, when instantiating `fib_aux`, first instantiate `fib_aux` as an empty class, and then use this value when evaluating the right hand side. Only when using `this` we obtain a truely recursive definition.

When evaluated, one sees that the nth element of `fib` obtains fib(n) and this computation is the efficient computation in which previous calls of fib(k) are memorized. If one replaces `Lazy(X)` by `() -> X`, one obtains an implementation of the Fibonacci numbers, which computes the correct results. However, since that implementation doesn't memorize values, it has exponential running time and on our laptop we were not able to compute fib(25).

Generalization. The above technique can easily be generalized to general algebraic types, in fact to all class structures avilable in C++. If one replaces in a tree structure all types by lazy types, then only a trunk of the tree structure is evaluated and kept in memory, namely the trunk which has been used already by any function accessing this structure.

5 COMBINING FUNCTIONAL AND IMPERATIVE PROGRAMMING

When combining functional and imperative programming we obtain more than just the disjoint union of both constructs. The translation can be extended to λ-terms which execute imperative C++-code with side effects. Assume

```
class Student{
public: int student_number;
        Student(int x){student_number = x;};}
```

Assume some standard (lazy or non-lazy – in fact the full range of C++ implementations of lists is possible) implementation `list<X>` of lists of type `X`. Then it is easy to define the map functional

```
template<class X,class Y> list<Y>*
                          map (X->Y f, list<X>* l);
```

which takes a function and a list and returns the result of applying each element to this list.

Now, if we extend the above language by allowing λ-terms to have arbitrary sequences of expressions, we can define

```
S->() print_Student_Number
  = \(Student s).() {cout >> Student.student >> endl;
                      return ();};
```

If we map this function to an element of `list` $<$`Student`$*>*$, then it will print out all the students, without any need for using an explicit loop. Similarly we can define functions, which update student numbers in a list of students, etc, using functional style programming in C++.

Having λ-terms with side effects seems to be an intereresting feature in programming and is not covered by usual implementations of λ-terms in C++, since those implementations require the body of a λ-term to be an expression – a sequence of statements is not allowed.

6 CONCLUSION

In this paper we showed how to introduce functional concepts into C++ in a provably correct way. The modelling and the correctness proof used monadic concepts as well as logical relations. We also showed how to integrate lazy evaluation and infinite structures into C++ and gave examples indicating how to model higher-order functions with side effects. This work lends itself to a number of extensions, for example, the integration of recursive higher-order functions, polymorphic and dependent type systems as well as the combination of larger parts of C++ with the λ-calculus. The accurate description of these extensions will require more sophisticated, e.g. domain-theoretic constructions and a more systematic mathematical modelling of C++. We believe that if our approach is extended to cover full C++, one obtains a language in which the worlds of functional and object-oriented programming are merged, and that we will see many examples, where the combination of both language concepts will result in interesting new programming techniques.

REFERENCES

[ASS85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs*. MIT Press, 1985.

[JT93] A. Jung and J. Tiuryn. A new characterization of lambda definability. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 245–257. Springer Verlag, 1993.

[Kel97] R. M. Keller. The Polya C++ Library. Version 2.0. Available via `http://www.cs.hmc.edu/ keller/Polya/`, 1997.

[Kis98] O. Kiselyov. Functional style in C++: Closures, late binding, and lambda abstractions. In *ICFP '98: Proceedings of the third ACM SIGPLAN International conference on Functional programming*, page 337, New York, NY, USA, 1998. ACM Press.

[KL05] O. Kiselyov and R. Lämmel. Haskell's overlooked object system. Draft. Submitted for journal publication. Online since 30 Sep. 2004. Full version released 10 September 2005, 2005.

[Läu95] K. Läufer. A framework for higher-order functions in C++. In *COOTS*, 1995.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(3):55–2, 1991.

[MS00] B. McNamara and Y. Smaragdakis. Functional programming in C++. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 118–129, New York, NY, USA, 2000. ACM Press.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Plo80] G. D. Plotkin. Lambda definability in the full type hierarchy. In R. Hindley and J. Seldin, editors, *To H.B. Curry: Essays in Combinatory Logic, lambda calculus and Formalisms*, pages 363 – 373. Academic Press, 1980.

[Pol81] W. Polak. Program verification based on denotation semantics. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 149–158, New York, NY, USA, 1981. ACM Press.

[Sch00] S. Schupp. Lazy lists in C++. *SIGPLAN Not.*, 35(6):47–54, 2000.

[Set03] A. Setzer. Java as a functional programming language. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs: International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. Selected Papers.*, pages 279 – 298. LNCS 2646, 2003.

[SS71] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[Sta85] R. Statman. Logical relations and the typed lambda-calculus. *Inf. and Control*, 65:85 – 97, 1985.

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[Str] J. Striegnitz. FACT! – the functional side of C++.
http://www.fz-juelich.de/zam/FACT.

[Tai67] W. W. Tait. Intensional interpretation of functionals of finite type. *J. Symbolic Logic*, 32:198 – 212, 1967.

[Win93] G. Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.