Chapter 1

Functional Concepts in C++

Rose H. Abdul Rauf[1], Ulrich Berger[2], Anton Setzer[2 3]

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 a Kripke-style logical relation. We also 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.

1.1 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-terms into C++ statements so as to integrate functional concepts. The translated code uses the object oriented approach of programming that involves the creation of classes for the λ-term. By using inheritance, we achieve that the translation of a λ-abstraction is an element of a function type.

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. 1.2). The correctness of our implementation is proved with respect to the usual (set-theoretic) denotational semantics of the simply typed λ-calculus and a mathematical model of a sufficiently large fragment of C++. The proof is based on a Kripke-style logical relation between C++ values and denotational values (Sect. 1.3). In Sect. 1.4 we introduce a general technique for introducing lazy

[1]Faculty of Information System and Quantitative Science, University of Technology MARA, 40450 Shah Alam, Selangor D.E., Malaysia; Email: `hafsah@tmsk.uitm.edu.my`

[2]Department of Computer Science, University of Wales Swansea, Singleton Park, Swansea SA2 8PP, UK; Email: `{u.berger,a.g.setzer}@swansea.ac.uk`

[3]Supported by EPSRC grant GR/S30450/01

evaluation into C++ by introducing a data type of lazy elements of an arbitrary C++ type.

Related work. Several researchers [7], [8] 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 [9] (a very elaborate approach) as well as FACT! [19] (extensive use of templates and overloading) and [7] (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, 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 for proving program correctness has been used before by Plotkin [12], Reynolds [14] and many others. The method of logical relations can be traced back at least to Tait [20] and has been used for various purposes, for example, for proving normalization (Tait [20]), computational adequacy (Plotkin [12]) and completeness (Jung and Tiuryn [5], Statman [18], Plotkin [13]). To our knowledge the verification of the implementation of the λ-calculus in C++ (and related object-oriented languages) using logical relations is new.

There are other fragments of object-oriented languages in the literature which are used to prove the correctness of programs. A well-known example is Featherweight Java ([4]). The model for this language avoids the use of a heap, since methods do not modify instance variables. In contrast, our model of C++ does make use of a heap and is therefore closer to the actual implementation of C++. Although our fragment of C++ does not allow for methods with side effects, it could easily be extended this way and then used to verify programs in C++ using side effects. This could be used, for instance, to prove the efficiency of the `Lazy` construct introduced in Sect. 1.4.

Lazy evaluation in C++ has been studied extensively in the literature (see e.g. [15], [9], [6]). To our knowledge, all implementations are restricted to lazy lists, whereas we introduce a general type of lazy elements of an arbitrary type, which not only corresponds to call-by-name (which is usually achieved by replacing a type A by $() \to A$), but also guarantees that elements are only evaluated once, as required by true lazy evaluation. Note as well that there is no need to add a new delay construct to C++ since our implementation of laziness makes use of the existing language of C++ only.

1.2 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*, *λ-calculus* for short, is given as follows: We assume a set $\mathsf{basetype}$ of base types $\rho, \sigma, \ldots$, and a set F of *basic functions* from base types to base types. Any native C++ type can be used as a base type, and any native C++ function without side effects can be used as a basic function. [4]

Types are elements of basetype, and function types $A \to B$ (if A, B are types). *Terms* are of the form x (variables), $\lambda x^A r$ (abstraction), $r\, s$ (application), $f[r_1, \ldots, r_n]$ (function application; $f \in \mathsf{F}$),[5] where r, s, r_i are terms. We treat constants, c, as basic functions with no arguments, and write in this case c instead of $c[\,]$.

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 usual:

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

$$\frac{\Gamma \vdash r_1 : \sigma_1 \ldots \Gamma \vdash r_k : \sigma_k}{\Gamma \vdash f[r_1, \ldots, r_k] : \rho} \quad (f \text{ a basic function of type } \sigma_1 \to \cdots \to \sigma_k \to \rho)$$

We now show how to translate λ-terms into C++. The translation is essentially the same as that given by the parsing function P introduced in Sect. 1.3 – the only difference is that P will be developed in an abstract setting, whereas the translation given in the following generates genuine C++ code.

Our translation generates new identifiers, which we need to disambiguate; in order for this to work, we restrict ourselves to the translation of finitely many λ-terms and types at a time. We first define an identifier $\mathsf{name}(a)$: String for finitely many a : Type. Here String is the set of strings.

- If A is a native C++-type, $\mathsf{name}(A)$ is a C++ identifier obtained from A. This is A, if A is already an identifier, and the result of removing blanks and modifying symbols not allowed in identifiers (e.g. replacing $*$ by x), in case A is a compound type like `long int` or $*A$.[6]

- $\mathsf{name}(A \to B) :=$"`C`"$*\mathsf{name}(A)*$"`_`"$*\mathsf{name}(B)*$"`D`", where $*$ means concatenation. Here `C` stands for an open bracket, `D` for a closing bracket, and `_` for the arrow in this identifier. By using these symbols we obtain valid C++-identifiers.[7]

---

[4] The translation given below makes sense as well for functions with side effects, including those which affect instance variables of the classes used. However, in this case we would go beyond the simply typed λ-calculus, and could not use the simple denotational semantics of the λ-calculus in order to express the correctness of the translation.

[5] Note that we do not have any product types and that native C++-functions are not necessarily objects – they can even be constants such as integers – therefore $f[r_1, \ldots, r_k]$ cannot be subsumed by the rule for $r\, s$.

[6] This modification might result in name clashes, in which case one adds some string like _n for some integer n in order to disambiguate the names. Since we are translating only finitely many λ-types at any time, this way of avoiding name clashes is always possible.

[7] Again, we might need to disambiguate the identifiers as it was done for native C++ types.

For instance $\mathsf{name}(\mathtt{int} \to \mathtt{int}) =$"`Cint_intD`", $\mathsf{name}((\mathtt{int} \to \mathtt{int}) \to \mathtt{int}) =$ "`CCint_intD_intD`". In the following, we write `CA_BD` instead of $\mathsf{name}(A \to B)$ and `CA_BD_aux` instead of $\mathsf{name}(A \to B)*$"`_aux`" (this type will be introduced below), similarly for other types.

For every $A \in \mathsf{Type}$ we introduce a series of class definitions, after which $\mathsf{name}(A)$ is a valid C++ type (assuming class definitions for any native C++ type used):

- For native C++-types the sequence of class definitions is empty.
- The sequence of class definitions for $A \to B$ consists of the class definitions of A, the class definitions of B not contained in the class definitions of A and additionally

  ```
  class CA_BD_aux{
    public: virtual B operator () (A x)=0;};
  typedef CA_BD_aux * CA_BD;
  ```

  So, `CA_BD_aux` is a class with one virtual method used as application, which maps an element of type `A` to an element of type `B`. `CA_BD` is a pointer to an element of this class.

Now we define for every λ-term t a sequence of C++-class definitions and a C++-term t^{C++}, s.t. if $t : A$, then t^{C++} is of type $\mathsf{name}(A)$. [8]

- If x is a variable, then the class definitions for introducing x are empty and $x^{C++} := x$.
- Assume $A = E \to F$, $t = \lambda x^A.r$. Assume the free variables of t are of type $x_1 : A_1, \ldots, x_n : A_n$ and that `t` is a new identifier. Assume $\mathsf{name}(A_i) =$ `Ai`, `xi` is the C++-representation for x_i, $\mathsf{name}(E) =$ `E`, $\mathsf{name}(F) =$ `F`, and $r^{C++} =$ `r`. The class definition for t consists of the class definition for r together with

  ```
  class t : CE_FD_aux{
    public:
      A1 x1;
      ...
      An xn;
      t(A1 x1,A2 x2, ... , An xn){
           this->x1 = x1;
           ...
           this->xn = xn;}
      virtual F operator () (E x){
            return r;};};
  ```

[8]Strictly speaking, t^{C++} depends on the choice of identifiers for λ-types and C++-classes representing λ-terms. When defining the parse function P in Sect. 1.3, this will be made explicit by having the dependency of this function on the context Γ and the class environment C. Since in our abstract setting λ-types are represented by themselves, P does not depend on the choice of identifiers for those types.

$t^{C++} :=$ `new t(x1,...,xn)`.

- Assume $t = r\, s$. Then the class definitions of t consist of the class definitions for r, and the class definitions for s (where the class definitions corresponding to λ-abstractions occurring in both r and s need to be introduced only once).[9] Furthermore $t^{C++} := (*(r^{C++}))(s^{C++})$.

- Assume $t = f[r_1, \ldots, r_n]$. Then the class definitions for t are the class definitions for r_i (again class definitions for λ-terms occurring more than once need only to be introduced once). Furthermore, $t^{C++} := \mathtt{f}(r_1^{C++}, \ldots, r_n^{C++})$.

Note that a λ-abstraction is interpreted as a function of its free variables in the form ($\mathtt{new\,t}(\mathtt{x_1}, \ldots, \mathtt{x_n})$). Hence, the evaluation of a λ-abstraction in an environment for the free variables is similar to a "closure" in implementations of functional programming languages.

We have developed a program which parses λ-terms and translates them into C++. Our intention is to upgrade this to an extension of the language of C++ by λ-types and -terms together with a parser program which translates this extended language into native C++. For this purpose we introduce a syntax for representing λ-types and -terms in C++. We use functional style notation rather than overloading existing C++-notation, since we belief that this will improve readability and acceptability of our approach by functional programmers. In our extended language, we write `A –> B` for the function type $\mathtt{A} \to \mathtt{B}$, `r ^^s` for the application of `r` to `s`,[10]and `\A x.B s` for λx^A.`s`, if $s : B$. (If s is a term starting with λ, `B` will be omitted). For instance, the term

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

is written in our extended C++ syntax as

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

We will use this extended syntax in our C++ implementation of lazy data structures (Sect. 1.4).

As an example, we show how the translation program transforms the term t above into native C++ code. We begin with the class definitions for the λ-types:

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

typedef Cint_intD_aux*  Cint_intD;
```

[9]A λ-abstraction is represented as a new instance of its corresponding class. Even if the classes for two occurrences of the same λ-abstraction coincide, for each occurrence a new instance is created. Therefore there is no problem, if a variable occurs as the same name, but with different referential meaning in two identical λ-expressions.

[10]Note that we cannot $\mathtt{r}(\mathtt{s})$ here, since this notation will not translate into application, but into $(*\,\mathtt{r})(\mathtt{s})$.

```
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 class definition for $t_1 := \lambda x^{\mathsf{int}}.f\ (f\ x)$ is

```
class t1 : public Cint_intD_aux{
 public :Cint_intD f;
 t1( Cint_intD f)  {   this-> f = f;};
 virtual int operator () (int x)
 { return (*(f))((*(f))(x)); };
};
```

and $t_1^{C++} = \texttt{new t1(f)}$. The class definitions for $t_0 := \lambda f^{\mathsf{int}\to\mathsf{int}}\lambda x^{\mathsf{int}}.f(fx)$ and $t_2 := \lambda x^{\mathsf{int}}.2+x$ (using identifiers `t0`, `t2`) are as follows:

```
class t0 : public CCint_intD_Cint_intDD_aux{
 public :
 t0( ) { };
 virtual Cint_intD operator () (Cint_intD f)
 { return new  t1( f); }
};

 class t2 : public Cint_intD_aux{
 public :
 t2( ) { };
 virtual int operator () (int x)
 { return x + 2; };
};
```

Finally

$$t^{C++} := (*((*(\texttt{ new t0( )}))(\texttt{ new t2( )})))(3);$$

When evaluating the expression t^{C++}, first the application of `t0` to `t2` is evaluated. To this end, instances `l0`, `l2` of the classes `t0` and `t2` are created first. Then the `operator()` method of `l0` is called. This call creates an instance `l1` of `t1`, with the instance variable `f` set to `l2`. The result of applying `t0` to `t2` is `l1`.

The next step in the evaluation of t^{C++} is to evaluate `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.[11] Note that `l0`, `l1`, `l2` were created on the heap, but have

[11] Note that this computation causes some overhead, since for every subterm of the form $\lambda x.r$ a new object is created, which is in many cases used once, and can be thrown away

not been deleted afterwards. The deletion of `l0`, `l1` and `l2` relies on the use of a garbage collected version of C++, alternatively we could use smart pointers in order to enforce their deletion.

1.3 PROOF OF CORRECTNESS

We now prove the correctness of our C++ implementation of the λ-calculus. For notational simplicity we restrict ourselves to the base type `int` of integers. 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 does not 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 is most convenient to work with a denotational model, for example, the naive set-theoretic hierarchy D of functionals of finite types over the integers[12] (setting $\mathsf{Z} = \{\ldots,-2,-1,0,1,2,\ldots\}$ and $X \to Y = \{f \mid f : X \to Y\}$):

$$\mathsf{D}(\mathsf{int}) = \mathsf{Z}, \quad \mathsf{D}(A \to B) = \mathsf{D}(A) \to \mathsf{D}(B), \quad \mathsf{D} = \bigcup_{A \in \mathsf{Type}} \mathsf{D}(A)$$

A *functional environment* is a mapping $\xi : \mathsf{Var} \to \mathsf{D}$. FEnv denotes the set of all functional environments. If Γ is a context, then $\xi : \Gamma$ means $\forall x \in \mathsf{dom}(\Gamma).\xi(x) \in \mathsf{D}(\Gamma(x))$.

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

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 [1] using functions `eval` and `apply`, but, in order to model the C++ implementation as

afterwards. One could optimize this, however at the price of having a much more complicated translation, and therefore a much more complex correctness proof of the translation.

[12] Recursion can be interpreted in a domain-theoretic model [12].

truthfully as possible, we make the pointer structures for the classes and objects explicit by letting the functions eval and apply modify these pointer structures via side effects.

We model only the fragment of C++ that we used in Section 1.2 to translate the simply typed λ-calculus into C++. Hence we assume that classes have instance variables, one constructor, and one method corresponding to the `operator()` method. The constructor has one argument for each instance variable and sets 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 are 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 is allocated on the heap, and these instance variables are 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. Hence, 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 $X \to_{\mathsf{fin}} Y$ 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	$=$	$\mathsf{Z}+\mathsf{Addr}$
F	$=$	a set of names for arithmetic C++ functions
App	$=$	$\mathsf{Z}+\mathsf{Var}+\mathsf{F}\times\mathsf{App}^*+\mathsf{App}\times\mathsf{App}+\mathsf{Constr}\times\mathsf{App}^*$
$\mathsf{Context}$	$=$	$\mathsf{Var}\to_{\mathsf{fin}}\mathsf{Type}$
Class	$=$	$\mathsf{Context}\times\mathsf{Var}\times\mathsf{Type}\times\mathsf{App}$
VEnv	$=$	$\mathsf{Var}\to_{\mathsf{fin}}\mathsf{Val}$
Heap	$=$	$\mathsf{Addr}\to_{\mathsf{fin}}\mathsf{Constr}\times\mathsf{Val}^*$
CEnv	$=$	$\mathsf{Constr}\to_{\mathsf{fin}}\mathsf{Class}$

Applicative terms ($\in \mathsf{App}$), which we write as n, x, $f[a_1,\dots,a_n]$, $(a\ b)$ and $c[a_1,\dots,a_n]$, correspond to the C++ constructs $\mathtt{n}$, $\mathtt{x}$, $\mathtt{f}(\mathtt{a}_1,\dots,\mathtt{a}_n)$, $(*\,(\mathtt{a}))(\mathtt{b})$ and $\mathtt{new}\ \mathtt{c}(\mathtt{a}_1,\dots,\mathtt{a}_n)$, while classes ($\in \mathsf{Class}$), written in the form $(\Gamma;x:A;b)$ with $\Gamma = x_1:A_1,\dots,x_n:A_n$, correspond to a C++ class definition of the form

```
class c : CA_BD_aux{
  public: A1 x1;
          ...
          An xn;
  c(A1 x1,A2 x2, ... , An xn){
```

```
                this->x1 = x1;
                ...
                this->xn = xn;}
  virtual B operator () (A x){
               return b;};};
```

The type B is omitted in $(\Gamma; x : A; b)$ since it can be derived, and the class name c is associated with the class through the class environment CEnv.

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 a *partial state monad* (the object part of which is)

$$\mathsf{M}_X(Y) := X \xrightarrow{\sim} Y \times X$$

where $X \xrightarrow{\sim} Y \times X$ is the set of partial functions from X to $Y \times X$. Elements of $\mathsf{M}_X(Y)$ are 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 [10]. We need to work with partial instead of total functions because the operations eval and apply defined below do not yield defined results in general. We will however prove that for inputs generated by translating well-typed λ-terms the results will always be defined.

Using the monad terminology the functionalities of the parsing function P and the operations eval and apply can now be written as

$$\begin{array}{rl} \mathsf{P}: & \mathsf{Context} \to \mathsf{Term} \to \mathsf{M}_{\mathsf{CEnv}}(\mathsf{App}) \\ \mathsf{eval}: & \mathsf{CEnv} \to \mathsf{VEnv} \to \mathsf{App} \to \mathsf{M}_{\mathsf{Heap}}(\mathsf{Val}) \\ \mathsf{apply}: & \mathsf{CEnv} \to \mathsf{Val} \to \mathsf{Val} \to \mathsf{M}_{\mathsf{Heap}}(\mathsf{Val}) \end{array}$$

Hence, parsing has a side effect on the class environment, while eval and apply have side effects on the heap. The function P corresponds to the assignment $t \mapsto t^{C++}$ introduced in Section 1.2, but makes the dependencies on the context and the class environment explicit.

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

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

is the action that maps any state $x_0 : X$ to (y_{k+1}, x_{k+1}) where $(y_i, x_i) \simeq e_i \; x_{i-1}$, for $i = 1, \ldots, k+1$ ($\simeq$ denotes the usual "partial equality"). We also allow let-expressions with pattern matching within a do-construct (with the obvious meaning). We adopt the convention that computations are "strict", i.e. the result of a computation is undefined if one of its parts is. Furthermore, we use the standard monadic notations

$$\begin{array}{lcll} \mathsf{return} & : & Y \to \mathsf{M}_X(Y) & \mathsf{return}\; y\, x = (y, x) \\ \mathsf{mapM} & : & (Z \to \mathsf{M}_X(Y)) \to Z^* & \mathsf{mapM}\; f\, \vec{a} = \mathsf{do}\{y_1 \leftarrow f\; a_1 \, ; \, \ldots \\ & & \quad \to \mathsf{M}_X(Y^*) & \quad \ldots \, ; \, y_k \leftarrow f\; a_k \, ; \, \mathsf{return}\; (y_1, \ldots, y_k)\} \end{array}$$

as well as

$$\begin{array}{lll}\mathsf{read} : X \to \mathsf{M}_{X\to_{\mathsf{fin}}Y}(Y), & \mathsf{read}\ x\ m \simeq (m\ x, m) & \\ \mathsf{add} : Y \to \mathsf{M}_{X\to_{\mathsf{fin}}Y}(X), & \mathsf{add}\ y\ m \simeq (x, m[x \mapsto y]) & \text{where } x = \mathsf{fresh}(m)\end{array}$$

Here, fresh is a function with the property that if $m : X \to_{\mathsf{fin}} Y$, then $\mathsf{fresh}(m) \in X \setminus \mathsf{dom}(m)$ [13]. With these notations the definitions of P, eval and apply read as follows

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

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

where $\emptyset$ is the undefined action, i.e. the partial function with empty domain[14].

Lemma 1.1. *(1)* $\mathsf{P}\ \Gamma\ r$ *is total and if* $\mathsf{P}\ \Gamma\ r\ C = (a, C')$*, then* $C \subseteq C'$*.*

(2) If $\mathsf{eval}\ C\ \eta\ a\ H = (v, H')$*, then* $H \subseteq H'$*.*

(3) If $\mathsf{apply}\ C\ v\ w\ H = (v', H')$*, then* $H \subseteq H'$*.*

Proof. Property (1) is direct by induction on the term r. Properties (2) and (3) can be proved by a straightforward simultaneous induction on the definitions of eval and apply, i.e. by "fixed point induction" [21]. □

Due to the complexity of C++ it would be a major task, which would require much more man power than was available in our project, to formally prove that our mathematical model, given by eval and apply, coincides with the operational semantics of C++.[15] (Note that other models of fragments of object-oriented languages in the literature face the same problem and their correctness w.r.t. real

[13] In our applications X will be a space of addresses which we assume to be infinite, i.e. we assume that the allocation of a new address is always possible.

[14] It would be more appropriate to let $\mathsf{apply}\ C\ n\ v$ result in an error, but, for simplicity, we identify errors with non-termination.

[15] The formalisation of the semantics of Java in [17] was a major project, and still this book excludes some features of Java like inner classes. Note that C++ is much more complex than Java.

languages is therefore usually not shown.) However, when going through the definitions we observe that the evaluation function eval is indeed defined in accordance with the expected behaviour of C++: An integer n is evaluated by itself, and a variable is evaluated by returning its value in the current environment η. The application of a native C++ function to arguments $a_1,\ldots,a_n$ is carried out by first evaluating $a_1,\ldots,a_n$ in sequence, and then applying the function f to those arguments. $(a\ b)$ corresponds in C++ to the construct $(*\ (a))(b)$. First a and b are evaluated. Because of type correctness, a must be an element of the type of pointers to a class, and the value of a will therefore be an address on the heap. On the heap the information about the class used and the values of the instance variables of that class are stored. Then $(*\ (a))(b)$ is computed by evaluating the body of the method of the class in the environment where the instance variables have the values as stored on the heap, and the abstracted variable has the result of evaluating b. This is what is computed by $\mathsf{eval}\ C\ \eta\ (a\ b)$ (which makes use of the auxiliary function apply). The expression $c[\vec{a}]$, which stands for the C++ expression $\mathtt{new\ c(a_0,\ldots,a_n)}$, is evaluated by first computing $\mathtt{a_0},\ldots,\mathtt{a_n}$ in sequence. Then new storage on the heap is allocated. Note that in our simplified setting, the constructor of $\mathtt{c}$ simply assigns to the instance variables the values of $\mathtt{a_0},\ldots,\mathtt{a_n}$. Consequently, the intended behaviour of C++ is that it stores on the heap the information about the class used and the result of evaluating $\mathtt{a_0},\ldots,\mathtt{a_n}$, which is what is carried out by eval.

The formal correctness proof for the translated code with respect to our mathematical model of (a fragment of) C++ is based on a Kripke-style logical relation between a C++ value ($\in \mathsf{Val}\times\mathsf{Heap}$) and a denotational value ($\in \mathsf{D}(A)$). The relation is indexed by the class environment C and the type A of the term. The relation

$$\sim^C_A \subseteq (\mathsf{Val}\times\mathsf{Heap})\times\mathsf{D}(A),$$

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

$$\begin{array}{rcl}
(v,H)\sim^C_{\mathsf{int}} n & :\Longleftrightarrow & v=n\\
(v,H)\sim^C_{A\to B} f & :\Longleftrightarrow & \forall C'\supseteq C, \forall H'\supseteq H, \forall(w,d)\in\mathsf{Val}\times\mathsf{D}(A):\\
 & & (w,H')\sim^{C'}_A d \Longrightarrow \mathsf{apply}\ C'\ v\ w\ H' \sim^{C'}_B f(d)
\end{array}$$

Note that the formula $\mathsf{apply}\ C'\ v\ w\ H' \sim^{C'}_B f(d)$ above states that $\mathsf{apply}\ C'\ v\ w\ H'$ is defined and the result is in relation $\sim^{C'}_B$ with $f(d)$. The formula $\mathsf{eval}\ C''\ \eta\ a\ H' \sim^{C''}_A [\![r]\!]\xi$ in the theorem below is to be understood in a similar way. We also set

$$(\eta,H)\sim^C_\Gamma \xi :\Leftrightarrow \mathsf{dom}(\Gamma)\subseteq\mathsf{dom}(\eta)\cap\mathsf{dom}(\xi)\wedge\forall x\in\mathsf{dom}(\Gamma)(\eta(x),H)\sim^C_{\Gamma(x)}\xi(x)$$

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

Theorem 1.2. *Assume $\Gamma\vdash r:A$, $\mathsf{P}\ \Gamma\ r\ C=(a,C')$, $C''\supset C'$ and $\xi:\Gamma$. Then for all $(\eta,H)\in\mathsf{VEnv}\times\mathsf{Heap}$:*

$$(\eta,H)\sim^{C''}_\Gamma \xi \Longrightarrow \mathsf{eval}\ C''\ \eta\ a\ H \sim^{C''}_A [\![r]\!]\xi.$$

Proof. The proof is by induction on the typing judgement $\Gamma \vdash r : A$. In the proof we will use the properties (1) and (2) of Lemma 1.1 as well as the following property of the relation $\sim_A^C$, which is clear by definition, and which in the following will be referred to as "monotonicity":

If $(v,H) \sim_A^C d$ and $H \subseteq H'$ and $C \subseteq C'$, then $(v,H') \sim_A^{C'} d$.

We now consider the four possible cases of how $\Gamma \vdash r : A$ can be derived.

$\Gamma, x : A \vdash x : A$. We have $\mathsf{P}(\Gamma, x : A)\, x\, C = \mathsf{return}\, x\, C = (x, C)$. Assume $C' \supseteq C$, $\xi : (\Gamma, x : A)$ and $(\eta, H) \sim_{\Gamma,x:A}^{C'} \xi$. We need to show $\mathsf{eval}\, C'\, \eta\, x\, H \sim_A^{C'} [\![x]\!]\xi$. Since $\mathsf{eval}\, C'\, \eta\, x\, H = (\eta(x), H)$ and $[\![x]\!]\xi = \xi(x) \in \mathsf{D}(A)$, and the assumption $(\eta, H) \sim_{\Gamma,x:A}^{C'} \xi$ entails $(\eta(x), H) \sim_A^{C'} \xi(x)$, we are done.

$\Gamma \vdash \lambda x^A r : A \to B$, derived from $\Gamma, x : A \vdash r : B$. $\mathsf{P}\, \Gamma\, \lambda x^A r\, C = (c[\mathsf{dom}(\Gamma)], C')$ where $\mathsf{P}(\Gamma, x : A)\, r\, C = (a, C_1)$, with $C_1 \supseteq C$ by (1), $c = \mathsf{fresh}(C_1)$ and $C' = C_1[c \mapsto (\Gamma; x : A; a)]) \supseteq C$. Assume $C'' \supseteq C'$, $\xi : \Gamma$ and $(\eta, H) \sim_\Gamma^{C''} \xi$. We need to show $\mathsf{eval}\, C''\, \eta\, c[\mathsf{dom}(\Gamma)]\, H \sim_{A\to B}^{C''} [\![\lambda x^A r]\!]\xi$. We have $\mathsf{eval}\, C''\, \eta\, c[\mathsf{dom}(\Gamma)]\, H = (h, H_1)$ where $\vec{v} = \mathsf{map}\, \eta\, \mathsf{dom}(\Gamma)$ (the usual map function), $h = \mathsf{fresh}(H)$ and $H_1 = H[h \mapsto (c, \vec{v})])$. In view of the definition of $\sim_{A\to B}^{C''}$ we assume $C''' \supseteq C''$, $H' \supseteq H_1$ and $(w, H') \sim_A^{C'''} d$. We need to show $\mathsf{apply}\, C'''\, h\, w\, H' \sim_B^{C'''} [\![r]\!]\xi[x \mapsto d]$. Clearly, $\mathsf{apply}\, C'''\, h\, w\, H' = \mathsf{eval}\, C'''\eta_1\, a\, H'$ where $\eta_1 = [\mathsf{dom}(\Gamma), x \mapsto \vec{v}, w]$. Furthermore, $(\eta_1, H') \sim_{\Gamma,x:A}^{C'''} \xi[x \mapsto d]$, by the assumptions $(\eta, H) \sim_\Gamma^{C''} \xi$ and $(w, H') \sim_A^{C'''} d$ and monotonicity. Using the induction hypothesis we obtain $\mathsf{eval}\, C'''\, \eta_1\, a\, H' \sim_B^{C'''} [\![r]\!]\xi[x \mapsto d]$ since $\mathsf{P}(\Gamma, x : A)\, r\, C = (a, C_1)$ and $C_1 \supseteq C'''$.

$\Gamma \vdash r\, s : B$, derived from $\Gamma \vdash r : A \to B$ and $\Gamma \vdash s : A$. By (1), $\mathsf{P}\, \Gamma\, r\, C = (a, C_1)$ with $C \subseteq C_1$ and $\mathsf{P}\, \Gamma\, s\, C_1 = (b, C_2)$ with $C_1 \subseteq C_2$. Therefore, $\mathsf{P}\, \Gamma\, (r\, s)\, C = (a\, b, C_2)$. Assume $C' \supseteq C_2$, $\xi : \Gamma$ and $(\eta, H) \sim_\Gamma^{C'} \xi$. We need to show $\mathsf{eval}\, C'\, \eta\, (a\, b)\, H \sim_A^{C'} [\![r\, s]\!]\xi$. By induction hypothesis and (2), $\mathsf{eval}\, C'\, \eta\, a\, H = (v, H_1)$ for some $H_1 \supseteq H$ with $(v, H_1) \sim_{A\to B}^{C'} [\![r]\!]\xi$ and, using monotonicity, $\mathsf{eval}\, C'\, \eta\, a\, H_1 = (w, H_2)$ for some $H_2 \supseteq H_1$ with $(w, H_2) \sim_A^{C'} [\![s]\!]\xi$. Hence, $\mathsf{apply}\, C'\, v\, w\, H_2 \sim_B^{C'} [\![r]\!]\xi([\![s]\!]\xi)$ and we are done, since $\mathsf{eval}\, C'\, \eta\, (a\, b)\, H = \mathsf{apply}\, C'\, v\, w\, H_2$ and $[\![r\, s]\!]\xi = [\![r]\!]\xi([\![s]\!]\xi)$.

$\Gamma \vdash f[r_1, \ldots, r_k] : \mathsf{int}$, derived from $\Gamma \vdash r_i : \mathsf{int}$, $i = 1, \ldots, k$. By (1), $\mathsf{P}\, \Gamma\, r_1\, C = (a_1, C_1)$ with $C \subseteq C_1$ and $\mathsf{P}\, \Gamma\, s\, C_{i+1} = (a_{i+1}, C_{i+1})$ with $C_i \subseteq C_{i+1}$, $i = 1, \ldots, k-1$. Hence, $\mathsf{P}\, \Gamma\, f[\vec{r}]\, C = (f[\vec{a}], C_k)$. Assume $C' \supseteq C_k$, $\xi : \Gamma$ and $(\eta, H) \sim_\Gamma^{C'} \xi$. We need to show $\mathsf{eval}\, C'\, \eta\, f[\vec{r}]\, H \sim_{\mathsf{int}}^{C'} [\![f[\vec{r}]]\!]\xi$. By induction hypothesis and (2), $\mathsf{eval}\, C'\, \eta\, a_1\, H = (n_1, H_1)$ for some $n_1 \in \mathsf{Z}$ and $H_1 \supseteq H$ with $(n_1, H_1) \sim_{\mathsf{int}}^{C'} [\![r_1]\!]\xi$, i.e. $n_1 = [\![r_1]\!]\xi$. Similarly, using monotonicity and (2), for $i = 1, \ldots, k-1$ we have $\mathsf{eval}\, C'\, \eta\, a_{i+1}\, H_i = (n_{i+1}, H_{i+1})$ for some $n_{i+1} \in \mathsf{Z}$ and $H_{i+1} \supseteq H_i$ with $n_{i+1} = [\![r_{i+1}]\!]\xi$. It follows $\mathsf{eval}\, C'\, \eta\, f[\vec{r}]\, H = ([\![f]\!][\vec{n}], H_k) = ([\![f[\vec{r}]]\!]\xi, H_k)$. $\square$

Corollary 1.3 (Correctness of the implementation). *Assume* $\vdash r : \mathsf{int}$ *and let* $C \in \mathsf{CEnv}$. *Then* $\mathsf{P}\, \emptyset\, r\, C = (a, C')$ *for some* $C' \supseteq C$. *Furthermore, for any heap* H, *any environment* η *and any* $C'' \supseteq C'$ *we have*

$$\mathsf{eval}\, C''\, \eta\, a\, H = (n, H')$$

where n *is the value of* r *and* H' *is some extension of* H.

Remark. The proof of Theorem 1.2 is rather "low level" since it mentions and manipulates the class environment and the heap explicitely. It would be desirable, in particular with regard to a formalisation in a proof assistant, to lift the proof to the same abstract monadic level at which the functions P, eval and apply are defined. A framework for carrying this out might be provided by suitable versions of Moggi's Computational λ–Calculus, Pitts' Evaluation Logic [11] and special logical relations for monads [2].

1.4 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 show how to translate this code into efficient C++ code, by using lazy evaluation. The full code for the following example, in which we have translated λ-types and -terms into original C++, is available from [16].

The standard technique for replacing call-by-value by call-by-name is to delay evaluation by replacing types A by $() \to A$ where $()$ is the empty type (i.e. `void`). However, according to the slogan "lazy evaluation = call-by-name+sharing" (which we learnt from G. Hutton [3]) lazy evaluation means more than delaying evaluation: it means as well that a term is evaluated only once. In order to obtain this, we 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. Note that this is a general definition, which is not restricted to lazy streams.The definition is as follows (we use the extended C++ syntax for λ-terms and -types introduced in Sect. 1.2, esp. `r ^^t` for application, \ for λ, and –> for →):

```
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>*
```

Note that without support by the extended syntax the code above would be much longer and considerably more complicated.

Using the class `lazy` 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>*
```

We define an operation which takes a function of type $() \to \mathtt{X}$ and returns the corresponding element of type $\mathtt{Lazy}(\mathtt{X})$:

```
template<typename X> Lazy(X) create_lazy
     (()-> X compute_function){
                 return new lazy<X>(compute_function);};
```

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_lazy<X>` computes the cons-operation on streams and returns lazily a lazy stream:

  ```
  template<typename X>Lazy(Lazy_Stream(X)) lazy_cons_lazy
               (Lazy(X) head,
                Lazy(Lazy_Stream(X)) tail){
    return create_lazy
                 (\ () x.new lazy_stream<X>(head,tail))};};
  ```

- `lazy_tail<X>` computes the tail of a stream lazily (we define here only its type):

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

- `lazy_zip_with<X>` computes the usual zip_with function (i.e. $\mathsf{zip\_with}(f,[a,b,..],[c,d,..]) = [f\ a\ c, f\ b\ d, \ldots]$; we define here only its type):

  ```
  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 `lazy_tail` and `lazy_zip_with` is straightforward, 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_lazy(
           one_lazy,
           lazy_cons_lazy(
             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` computes to $\mathsf{fib}(n)$ and this computation is the efficient one in which previous calls of $\mathsf{fib}(k)$ are memoized. Replacing `Lazy(X)` by $() \text{–>}$ `X`, results in an implementation of the Fibonacci numbers which is still correct, but requires exponential space since memoization is lost (on our laptop we were not able to compute $\mathsf{fib}(25)$).

Generalization. The above technique can easily be generalized to arbitrary algebraic types, in fact to all class structures available in C++. If, for example, 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.

1.5 CONCLUSION

In this paper we have shown 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 have shown how to integrate lazy evaluation and infinite structures into C++.

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 would require more sophisticated, e.g. domain-theoretic constructions and a more systematic mathematical modelling of C++. It would be as well be interesting to expand our fragment of C++ in order to deal with side effects. This would allow for instance to *prove* that our lazy construct actually gives rise to an efficient implementation of the Fibonacci function.

We believe that if our approach is extended to cover full C++, we obtain 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 (e.g. the use of λ-terms with side-effects) will result in interesting new programming techniques.

REFERENCES

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

[2] J. Goubault-Larrecq, S. Lasota, and D. Nowak. Logical relations for monadic types. In Julian C. Bradfield, editor, *Proceedings of the 16th International Workshop on Computer Science Logic (CSL'02)*, volume 2471 of *Lecture Notes in Computer Science*, pages 553–568, Edinburgh, Scotland, UK, September 2002. Springer.

[3] G. Hutton. *Programming in Haskell*. Cambridge University Press, Cambridge, UK, 2006.

[4] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[5] 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, 1993.

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

[7] 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.

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

[9] 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.

[10] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[11] A. M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer, 1991.

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

[13] 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.

[14] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP'83*, pages 513–523. North-Holland, 1983.

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

[16] A. Setzer. Lazy evaluation in C++. `http://www.cs.swan.ac.uk/~csetzer/articles/additionalMaterial/lazyEvaluationCplusplus.cpp`, 2006.

[17] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine – Definition, Verification, Validation.* Springer, 2001.

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

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

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

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