# A Provably Correct Translation of the $\lambda$-Calculus into a Mathematical Model of C++

Rose H. Abdul Rauf[*], Ulrich Berger, Anton Setzer[‡]

November 9, 2006

**Abstract**

We introduce a translation of the simply typed $\lambda$-calculus into C++, and give a mathematical proof of the correctness of this translation. For this purpose we develop a suitable fragment of C++ together with a denotational semantics. We then introduce a formal translation of the $\lambda$-calculus into this fragment, and show that this translation is correct with respect to the denotational semantics and complete. We introduce a mathematical model for the evaluation of programs of this fragment, and show that the evaluation computes the correct result with respect to this semantics.

## 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 $\lambda$-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 $\lambda$-term. By using inheritance, we achieve that the translation of a $\lambda$-abstraction is an element of a function type.

In this article, we do not only present this translation, but give as well a mathematical proof that it is correct. For this purpose we introduce a suitable fragment of C++ with a precise denotational semantics. We give a formal translation of $\lambda$-terms into this fragment and show that it preserves this semantics. We will show as well completeness, i.e. essentially all programs in this fragment of C++ can be obtained by translating terms of the $\lambda$-calculus. We develop a mathematical model for the evaluation of programs in this model, and show that this evaluation is correct with respect to the denotational semantics. This shows that our translation results in C++ programs which are evaluated correctly in our mathematical model of C++.

We hope that our model of a fragment of C++ which includes a formal model of the heap, will have applications which go beyond the translation of

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

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

[‡]Supported by EPSRC grant GR/S30450/01.

the typed $\lambda$-calculus. We expect that extensions of this model can be used to verify formally the correctness of more complex C++ programs, including programs with side effects.

**Organisation of the paper.** In Sect. 2 we introduce the typed $\lambda$-calculus together with a standard denotational semantics. In Sect. 3, we present our parser-translator program, which translates $\lambda$-terms into the full language of C++. We discuss how the translated code is executed including a description of the memory allocation. Since it is not feasible (at least for our group) to prove the correctness of our translation with respect to the full operational semantics of the very complex language C++, we develop in Sect. 4 a small fragment of C++ into which we can translate the typed $\lambda$-calculus. We introduce as well the evaluation of applicative terms in this language. In Sect. 5 we give a formal translation of the typed $\lambda$-calculus into this fragment. We introduce a denotational semantics of the fragment of C++ and show correctness and completeness of the formal translation: The translation respects the denotational semantics, and essentially all applicative terms can be obtained by translating suitable typed $\lambda$-terms (completeness). In Sect. 6 we show that the evaluation of $\lambda$-terms is correct with respect to the denotational semantics. This proof makes use of a Kripke-style logical relation. We conclude with an overall result, namely that if we translate a $\lambda$-term into the fragment of C++ and evaluate it, we obtain the correct result with respect to the denotational semantics.

**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++). What is different in our paper is that we develop a mathematical model of a fragment of C++, and that we formally prove the correctness of our translation.

The approach of using denotational semantics and logical relations for proving program correctness has been used before by Plotkin [Plo77], Reynolds [Rey83] and many 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 $\lambda$-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 ([IPW99]). 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++ with side effects.

**Acknowledgements:** We would like to thank the referees of earlier versions of this paper for valuable comments.

## 1.1 General Notations

**Notation 1.1 (Finite maps)** *By $X \to_{\mathsf{fin}} Y$ we denote the set of finite maps from the set $X$ to the set $Y$, that is, the set of functions $f\colon \mathsf{dom}(f) \to Y$ where $\mathsf{dom}(f)$ is a finite subset of $X$. If $f \in X \to_{\mathsf{fin}} Y$ and $(x, y) \in X \times Y$, then $f[x \mapsto y]$ denotes the finite function with domain $\mathsf{dom}(f) \cup x$ that sends $x$ to $y$ and any other $x' \in \mathsf{dom} f$ to $f(x')$. A list $x_1 : y_1, \ldots, x_n : y_n$, where the $x_i$ are distinct elements of $X$ and the $y_i$ are in $Y$, denotes an element of $X \to_{\mathsf{fin}} Y$ in the obvious way. Furthermore, $f, x : y := f[x \mapsto y]$.*

In this article we observe a strict naming convention: Once a group of letters has been used to range over a certain entity (e.g. $A$, $B$ (but not $C$) range over types), letters in that group (possibly with sub- or superscripts) will always denote instances of that entity. There will be only two exceptions: $x, y, z$ may denote elements of unspecified sets $X, Y, Z$ as well as variables (Definition 2.2), and $f$ ranges over unspecified functions as well as basic C++ functions 2.1. For the reader's convenience there is a complete table of notations in Sect. 8 at the end of this article

# 2 The Typed $\lambda$-Calculus

We introduce a version of the typed $\lambda$-calculus based on base types, which are native C++-types, and basic functions, which are native C++-functions.

**Assumption 2.1** *(a) We fix a set $\mathsf{basetype}$ of base types $\rho, \sigma, \ldots$. One specific base type is the type of integers $\mathsf{int}$.*

*(b) We fix a set $\mathsf{F}$ of names for* basic functions $f : (\rho_1, \ldots, \rho_k) \to \sigma$.

*(c) We view functions of arity 0 as constants and denote them by the letter $n$.*

*(d) Let $[\![\rho]\!]$ denote the set of elements of base type $\rho$. In case of $\mathsf{int}$, $[\![\mathsf{int}]\!]$ is the set of integers.*

*(e) Let $[\![f]\!] : [\![\rho_1]\!] \times \ldots \times [\![\rho_k]\!] \to [\![\sigma]\!]$ be the function denoted by $f \in \mathsf{F}$. Especially we assume that a constant (0-ary function) $n$, which stands for the integer $n$, is interpreted by itself (i.e. by $n$).*

Any native C++ type can be used as a base type, and any native C++ functions without side effects (including constants) can be used as basic functions [1].

**Definition 2.2 (Simply typed $\lambda$-calculus with basic functions)**

*(a) We fix a set $\mathsf{Var}$ of variables $x, y, z, \ldots$.*

[1]The translation given below makes sense as well fore 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 $\lambda$-calculus, and could not use the simple denotational semantics of the $\lambda$-calculus in order to express the correctness of the translation.

*(b) The sets of types, contexts and $\lambda$-terms are defined as follows:*

$$\begin{array}{llll} \textit{Types:} & \mathsf{Type} \ni A, B, E, F & ::= & \rho \mid A \to B \\ \textit{Contexts:} & \mathsf{Context} \ni \Gamma, \Delta & ::= & \mathsf{Var} \to_{\mathsf{fin}} \mathsf{Type} \\ \lambda\textit{-terms:} & \mathsf{Term} \ni r, s, t & := & x \mid f[r_1, \ldots, r_k] \mid r\ s \mid \lambda x^A.r \end{array}$$

*(c) The relation $\Gamma \vdash r : A$ is inductively defined by the following standard* typing rules*:*

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

$$\frac{\Gamma \vdash r_1 : \sigma_1 \ldots \Gamma \vdash r_k : \sigma_k}{\Gamma \vdash f[r_1, \ldots, r_k] : \rho} \quad (f : (\rho_1, \ldots, \rho_k) \to \sigma)$$

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 the rule for $f[r_1, \ldots, r_k]$ is not subsumed by the rule for $r\ s$.

The simply type $\lambda$-calculus has a well-known operational semantics defined by $\beta$-reduction, $(\lambda x^A r)s \to_\beta r[s/x]$, and function reduction, $f[n_1, \ldots, n_k] \to_f [\![f]\!](n_1, \ldots, n_k)$. But there is also an equivalent denotational semantics which, for our purposes, will be more convenient to work with. Since our calculus does not allow for recursive definitions, it is possible to interpret types and terms in a naive set-theoretic hierarchy $\mathsf{D}$ of functionals of finite types over the base types:

**Definition 2.3** *For $A \in \mathsf{Type}$ we define the set $\mathsf{D}(A)$ of functionals of finite types over $A$ by induction on $A$:*

$$\begin{array}{rcl} \mathsf{D}(\rho) & := & [\![\rho]\!] \\ \mathsf{D}(A \to B) & := & \textit{the set of functions from } \mathsf{D}(A) \textit{ to } \mathsf{D}(B) \\ \mathsf{D} & := & \bigcup_{A \in \mathsf{Type}} \mathsf{D}(A) \end{array}$$

*Semantic values (elements of $\mathsf{D}(A)$) are denoted by $d$.*

**Definition 2.4** *(a) A* functional environment *is a mapping $\xi : \mathsf{Var} \to \mathsf{D}$.* $\mathsf{FEnv}$ *denotes the set of all functional environments.*

*(b) If $\Gamma$ is a context, then $\xi : \Gamma$ means $\forall x \in \mathsf{dom}(\Gamma).\xi(x) \in \mathsf{D}(\Gamma(x))$.*

**Definition 2.5 (Denotational semantics of the simply typed $\lambda$-calculus)** *For every typed $\lambda$-term $\Gamma \vdash r : A$ and every functional environment $\xi : \Gamma$ the denotational value $[\![r]\!]\xi \in \mathsf{D}(A)$ is defined by*

$$\begin{array}{rcl} [\![x]\!]\xi & = & \xi(x) \\ [\![f[r_1, \ldots, r_k]]\!]\xi & = & [\![f]\!]([\![r_1]\!]\xi, \ldots, [\![r_k]\!]\xi) \\ [\![r\ s]\!]\xi & = & [\![r]\!]\xi([\![s]\!]\xi) \\ [\![\lambda x^A.r]\!]\xi & = & \lambda\!\!\lambda d \in \mathsf{D}(A).[\![r]\!]\xi[x \mapsto d] \end{array}$$

## 3 Translation of Typed $\lambda$-Terms into C++

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

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

- If $\rho$ is a native C++-type, $\mathsf{name}(\rho)$ is a C++ identifier obtained from $\rho$. This is $\rho$, if $\rho$ is already an identifier, and the result of removing blanks and modifying symbols not allowed in identifiers (e.g. replacing $*$ by $x$), in case $\rho$ is a compound type like `long int` or $*\,\rho$. [2]
- $\mathsf{name}(A \to B) :=$"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.[3]

For instance $\mathsf{name}(\texttt{int} \to \texttt{int}) =$"`Cint_intD`", $\mathsf{name}((\texttt{int} \to \texttt{int}) \to \texttt{int}) =$ "`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`" (that 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,[4] which maps an element of type `A` to an element of type `B`. `CA_BD` is a pointer to an element of this class. The body of this method will then be the body of the function to be invoked when applied to its arguments.

Now we define for every $\lambda$-term $r$ a sequence of C++-class definitions and a C++-term $r^{C++}$, s.t. if $r : A$, then $r^{C++}$ is of type $\mathsf{name}(A)$. [5]

[2] 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 $\lambda$-types at any time, this way of avoiding name clashes is always possible.

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

[4] In C++, if an object `o` has a method `B operator () (A x)`, invocation of this method is written like application, i.e. as `o(s)`. Note however that Java objects correspond in C++ to pointers to C++-objects. A pointer `o'` to an object `o` has first to be dereferenced, written as `(* o')`, and therefore `o'` applied to `s` is written as `(* o')(s)`. Note that, when creating an object using `new`, we obtain a pointer to an object.

[5] Strictly speaking, $r^{C++}$ depends on the choice of identifiers for $\lambda$-types and C++-classes representing $\lambda$-terms. When defining the parse function $\mathsf{P}$ in Sect. 6, this will be made explicit

- If $x$ is a variable, then the class definitions for introducing $x$ are empty and $x^{C++} := x$.

- Let $t = \lambda x^A.r$ be of type $A \to B$. Assume the free variables of $t$ are of type $x_1 : A_1, \ldots, x_k : A_k$ and that `t` is a new identifier. Assume $\mathsf{name}(A_i) = $ `Ai`, `xi` is the C++-representation for $x_i$, $\mathsf{name}(A) = $ `A`, $\mathsf{name}(B) = $ `B`, and $r^{C++} = $ `r`. The class definition for $t$ consists of the class definition for $r$ together with

  ```
  class t : CA_BD_aux{
    public:
      A1 x1;
      ...
      An xk;
      t(A1 x1,A2 x2, ... , Ak xk){
           this->x1 = x1;
           ...
           this->xk = xk;}
      virtual B operator () (A x){
            return r;};};
  ```

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

  Therefore the class definition of `t` has instance variables `xi` of type `Ai`. The constructors has one argument for each variable `xi` and sets the instance variable `xi` to the value of that argument. The class has one method `operator()` with one argument `x` of type `A`. When invoked, the body of this method `r`, which is the translation of the body of the $\lambda$-term, is evaluated in the environment mapping `x` to the value of the argument of the method, and `xi` to the value of this instance variable. Note that no other variables are visible in the body of this method, since this environment might differ between when an object of this class was created and when it is used. That is the reason why one needs to copy first, when creating an object of such a class, the environment into some instance variables.[6]

  When applying an object of this class to an element, the body of the $\lambda$-term is invoked. The $\lambda$-term is translated into the statement which creates a new object with the instance variables set to the value they have in the current environment.

- 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 $\lambda$-abstractions occurring in both $r$ and $s$ need only to be introduced once).[7] Furthermore $t^{C++} := (*(r^{C++}))(s^{C++})$.

by having the dependency of this function on the context $\Gamma$ and the class environment $C$. Since in our abstract setting $\lambda$-types are represented by themselves, $\mathsf{P}$ does not depend on the choice of identifiers for those types.

[6] In C++ there are no inner classes as they occur in Java, which allow references to the current environment.

[7] A $\lambda$-abstraction is represented as a new instance of its corresponding class. Even if the classes for two occurrences of the same $\lambda$-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 $\lambda$-expressions.

So $t$ is interpreted as the result of applying the translation of $r$ to the translation of $s$.

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

  So $t$ is interpreted as the result of applying the native C++ function `f` to the translations of $r_i$.

Note that a $\lambda$-abstraction is interpreted as a function of its free variables in the form $(\mathtt{new\,t}(\mathtt{x_1}, \ldots, \mathtt{x_k}))$. Hence, the evaluation of a $\lambda$-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 $\lambda$-terms and translates them into the full language of C++. Our intention is to upgrade this to an extension of the language of C++ by $\lambda$-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 $\lambda$-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` $\rightarrow$> `B` for the function type `A` $\rightarrow$ `B`, `r ^^s` for the application of `r` to `s`[8], and \\`A x.B s` for $\lambda x^A$.`s` if $s : B$. (If $s$ is a term starting with $\lambda$, `B` will be omitted). For instance, the term

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

is written in our extended C++ syntax as

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

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 $\lambda$-types:

```
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 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)); };
};
```

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

and $t_1^{C++} =$ `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.[9] 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++, alternatively we could use smart pointers in order to enforce their deletion.

## 4 Modelling a Fragment of C++

In this section we construct a mathematical model of a fragment of C++ that contains the code created by the translation of $\lambda$-terms described in the previous section. We model the execution of C++ code by functions eval and apply, similar to the modelling of the $\lambda$-calculus in [ASS85]. However, in order to model the C++ implementation as truthfully as possible, we differ from [ASS85] by making the pointer structures for the classes and objects explicit and letting the functions eval and apply modify these pointer structures via side effects.

When we investigate what was needed from C++ in order to translate simply typed $\lambda$-terms, we see that the classes obtained have instance variables, one constructor, and one method corresponding to the `operator()` method. The

[9]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 only once, and can be thrown away afterwards. One could optimise this, however at the price of having a much more complicated translation, and therefore a much more complex correctness proof of the translation.

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 a simplified C++ expression. Here simplified C++ expressions are the C++ expressions occurring in the translation process, which were all translations of $\lambda$-terms. Simplified C++-expressions are variables, native C++ functions applied to simplified C++ expressions, the application of one simplified C++ expression to another simplified C++ expression (which corresponds to the method call in case the first applicative term is an object), and the construct `new` applied to a constructor and simplified C++ expressions.

We develop a language which formulates this fragment of C++. In this language, a class is be given by a context representing its instance variables, the abstracted variable of the method and its type, and an applicative term. Applicative terms (which correspond to the simplified C++ expressions above) are variables, native C++ functions applied to applicative terms (where C++ functions with no arguments are constants), 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 (which corresponds to the `new`-construct).

This fragment could easily be extended in order to cover modification of instance variables and method calls in the body of a method, the possibility of having several methods, and other C++ constructs.

**Definition 4.1 (Applicative terms, classes, class environments)**

*Let* $\mathsf{Constr}$ *be an infinite set of* constructors *(i.e. class names), denoted by* $c$.

| | | | |
|---|---|---|---|
| *Applicative terms:* | $\mathsf{App} \ni a, b$ | $::=$ | $x \mid f[a_1, \ldots, a_k] \mid a\ b \mid c(a_1, \ldots, a_k)$ |
| *Classes:* | $\mathsf{Class}$ | $::=$ | $(\Gamma; x : A; b)$ |
| *Class environments:* | $\mathsf{CEnv} \ni C$ | $::=$ | $\mathsf{Constr} \rightarrow_{\mathsf{fin}} \mathsf{Class}$ |

Applicative terms ($\in \mathsf{App}$) correspond to the C++ constructs `x`, $\mathtt{f}[\mathsf{a}_1, ..., \mathsf{a}_k]$, $(*\ (\mathsf{a}))(\mathsf{b})$ and `new` $\mathsf{c}(\mathsf{a}_1, \ldots, \mathsf{a}_k)$. A class $(\Gamma; x : A; b) \in \mathsf{Class}$, where $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, corresponds to a C++ class definition of the form

```
class c : CA_BD_aux{
  public: A1 x1;
          ...
          An xk;
  c(A1 x1,A2 x2, ... , Ak xk){
                this->x1 = x1;
                ...
                this->xk = xk;}
  virtual B operator () (A x){
               return b;};};
```

Note that 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 $\mathsf{CEnv}$.

**Definition 4.2** *We define the free variables* $\mathrm{FV}(a)$ *of an applicative term* $a \in \mathsf{App}$ *as follows:* $\mathrm{FV}(x) = \{x\}$, $\mathrm{FV}(f[a_1, \ldots, a_k]) := \mathrm{FV}(a_1) \cup \cdots \cup \mathrm{FV}(a_k)$, $\mathrm{FV}(a\ b) := \mathrm{FV}(a) \cup \mathrm{FV}(b)$, $\mathrm{FV}(c(a_1, \ldots, a_k)) := \mathrm{FV}(a_1) \cup \cdots \cup \mathrm{FV}(a_k)$.

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.

Note as well that on the heap we store elements of the form $c(\vec{v})$, which can be represented as elements of $\mathsf{Constr} \times \mathsf{Val}^*$ (here $\mathsf{Val}$ is the set of values). Therefore we can model the heap as a finite function from addresses to $\mathsf{Constr} \times \mathsf{Val}^*$.

**Definition 4.3 (Values, closures, heaps, and value environments)**

*Let* $\mathsf{Addr}$ *be an infinite set of* addresses *denoted by* $h$.
*Let* $n$ *range over C++ constants, that is, elements of basic C++ types;*

| | | | |
|---|---|---|---|
| *Values:* | $\mathsf{Val} \ni v, w$ | $::=$ | $n \mid h$ |
| *Closures:* | $\mathsf{Constr} \times \mathsf{Val}^*$ | $::=$ | $(c, v_1, \ldots, v_k)$ |
| *Heaps:* | $\mathsf{Heap} \ni H$ | $::=$ | $\mathsf{Addr} \to_{\mathsf{fin}} (\mathsf{Constr} \times \mathsf{Val}^*)$ |
| *Value environments* | $\mathsf{VEnv} \ni \eta$ | $::=$ | $\mathsf{Var} \to_{\mathsf{fin}} \mathsf{Val}$ |

*Note that* $n$ *denotes both constants and elements of basic C++-types. Since constants are to be interpreted by themselves, this doesn't cause any confusion.*

The functions $\mathsf{eval}$ and $\mathsf{apply}$ defined below have side effects on the heap. This fact can be conveniently expressed using monads.

**Definition 4.4 (State monad)**

*The* partial state monad *for a given sets* $X$ *(of states) is the functor* $\mathsf{M}_X\colon \mathsf{Set} \to \mathsf{Set}$ *(the object part of which is)*

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

*where* $X \overset{\sim}{\to} Y \times X$ *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. So, an element of $\mathsf{M}_X(Y)$ is a partial function which, depending on the current state returns a result and a new state (or fails). Monads are a category-theoretic concept whose computational significance was discovered by Moggi [Mog91].

We need to work with partial instead of total functions because the operations $\mathsf{eval}$ and $\mathsf{apply}$ defined below do not yield defined results in general. We will however prove that for inputs that can be typed the results will always be defined.

**Notation 4.5** ($\mathsf{do}, \mathsf{return}, \mathsf{mapM}, \mathsf{read}, \mathsf{add}$) *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* $y_0 : Y$ *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"). The intuitive idea is that the* $\mathsf{do}$*-expression is computed by evaluating* $e_0, \ldots, e_{k+1}$ *in sequence, where*

*$e_i$ can make use of the result $y_j$ returned by $e_j$ ($j < i$). The result returned is that of $e_{k+1}$, and the computation of each $e_i$ might change the state.*

*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*

$$\begin{array}{ll}
\mathsf{return} : Y \to \mathsf{M}_X(Y) & \mathsf{return}\ y\ x = (y, x) \\
\mathsf{mapM} : (Z \to \mathsf{M}_X(Y)) \to Z^* \to \mathsf{M}_X(Y^*) & \mathsf{mapM}\ f\ \vec{z} = \mathsf{do}\{y_1 \leftarrow f\ z_1\ ;\ \ldots \\
& \quad \ldots\ ;\ y_k \leftarrow f\ z_k\ ;\ \mathsf{return}\ (\vec{y})\}
\end{array}$$

*as well as*

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

*Here,* fresh *is a function with the property that if $m : X \to_{\mathsf{fin}} Y$, then $\mathsf{fresh}(m) \in X \setminus \mathsf{dom}(m)$* [10].

**Definition 4.6** *We define functions*

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

*by mutual recursion as follows (in Sect.6 we will omit the argument $C$, since it will be a global parameter):*

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

*where $\emptyset$ is the undefined action, i.e. the partial function with empty domain*[11].

**Lemma 4.7** *(1) If* eval *$C\ \eta\ a\ H = (v, H')$, then $H \subseteq H'$.*

*(2) If* apply *$C\ v\ w\ H = (v', H')$, then $H \subseteq H'$.*

**Proof.** Straightforward simultaneous induction on the definitions of eval and apply, i.e. by "fixed point induction" [Win93]. **q.e.d.**

Due to the complexity of C++ it would be a major task, which would require much more man power than was available in our research group, to formally prove that our mathematical model, given by eval and apply, coincides with

[10]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.

[11]It would be more appropriate to let apply $C\ n\ v$ result in a finite error, but, for simplicity, we identify errors with non-termination.

the operational semantics of C++.[12] (Note that other models of fragments of object-oriented languages in the literature face the same problem and their correctness w.r.t. real languages is therefore usually not shown.) However, when going through the definitions we observe that the evaluation function $\mathsf{eval}$ is indeed defined in accordance with the expected behaviour of C++:

- A variable is evaluated by returning its value in the current environment $\eta$.
- The application of a native C++ function to arguments $a_1, \ldots, a_k$ is carried out by first evaluating $a_1, \ldots, a_k$ in sequence, and then applying the function $f$ to those arguments.
- Note that constants are special cases of functions with arity 0, and therefore constants are evaluated by themselves.
- $(a\ b)$ 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 $\mathsf{eval}\ \eta\ (a\ b)$ (which makes use of the auxiliary function $\mathsf{apply}$).
- The expression $c(\vec{a})$, which stands for the C++ expression `new` $\mathsf{c}(\mathsf{a}_0, \ldots, \mathsf{a}_\mathsf{k})$, is evaluated by first computing $\mathsf{a}_0, \ldots, \mathsf{a}_\mathsf{k}$ in sequence. Then new storage on the heap is allocated. Note that in our simplified setting, the constructor of $\mathsf{c}$ simply assigns to the instance variables the values of $\mathsf{a}_0, \ldots, \mathsf{a}_\mathsf{k}$. Consequently, the intended behaviour of C++ is that it stores on the heap the information about the class used and the result of evaluating $\mathsf{a}_0, \ldots, \mathsf{a}_\mathsf{k}$, which is what is carried out by $\mathsf{eval}$.

## 5 Formal Translation of Typed $\lambda$-Terms and its Correctness

Despite of the fact that we could describe only informally the connection of our mathematical model with the actual implementation of C++, we will be able to prove formally that the model as well as the translation of $\lambda$-terms described in Section 3 are correct in the following sense: As we did for $\lambda$-terms, we will define for C++ terms, $a \in \mathsf{App}$, a typing relation, $\Gamma \vdash a : A$, and a denotational semantics, $[\![\, a \,]\!]^H \xi \in \mathsf{D}(A)$. Similarly, we will define for values, $v \in \mathsf{Val}$, a relation $H \vdash v : A$ and a semantics $[\![\, v \,]\!]^H \in \mathsf{D}(A)$ (all these definitions will depend on a class environment $C \in \mathsf{CEnv}$). Our main results will be the correctness of the translation function, $\mathsf{P}$ (see below), and the evaluation function, $\mathsf{eval}$, with

[12]The formalisation of the semantics of Java in [SSB01] 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.

respect to these typing relations and denotational semantics (Theorems 5.6 and 6.9).

In this section we carry out the first step, namely the introduction of the parsing relation and a proof that it is correct and complete. In the next section we will show that the evaluation of applicative terms is correct as well and obtain the correctness of our implementation.

## Definition of the Parse Function $\mathsf{P}$

We are going to define a formal analogue to the translation of $\lambda$-terms described in Section 3. We use the monadic notation from Section 4.

**Definition 5.1 (Definition of the Parse Function $\mathsf{P}$)** *We define a function*

$$\mathsf{P} : \quad \mathsf{Context} \to \mathsf{Term} \to \mathsf{M}_{\mathsf{CEnv}}(\mathsf{App})$$

*by recursion on terms as follows:*

$$\begin{aligned} \mathsf{P}\,\Gamma\,x &= \mathsf{return}\;x, \;\; \textit{if } x \textit{ is a variable} \\ \mathsf{P}\,\Gamma\,f[\vec{r}] &= \mathsf{do}\{\vec{a} \leftarrow \mathsf{mapM}\;(\mathsf{P}\,\Gamma)\;\vec{r}\,;\;\mathsf{return}\;f[\vec{a}]\} \\ \mathsf{P}\,\Gamma\,(r\;s) &= \mathsf{do}\{(a,b) \leftarrow \mathsf{mapM}\;(\mathsf{P}\,\Gamma)\;(r,s)\;;\;\mathsf{return}\;(a\;b)\} \\ \mathsf{P}\,\Gamma\,(\lambda x^A.r) &= \mathsf{do}\{a \leftarrow \mathsf{P}\;(\Gamma, x : A)\;r\;;\;c \leftarrow \mathsf{add}(\Gamma; x : A; a)\;;\;\mathsf{return}\;c(\mathsf{dom}(\Gamma))\} \end{aligned}$$

*Hence, the translation has a side effect on the class environment.*

**Lemma 5.2** $\mathsf{P}\,\Gamma\,r$ *is total and if* $\mathsf{P}\,\Gamma\,r\,C = (a, C')$*, then* $C \subseteq C'$*.*

**Proof.** Induction on the term $r$. **q.e.d.**

## Typing and denotational semantics of applicative terms

**Definition 5.3 (Typing of Applicative Terms)** *We define inductively a typing relation* $C; \Gamma \vdash a : A$ *(where we sometimes write* $\Gamma \vdash a : A$ *instead, if* $C$ *is a global fixed parameter):*

$$C; \Gamma, x : A \vdash x : A$$

$$\frac{\begin{array}{c} f \in \mathsf{F},\;\; f : (\rho_1, \dots, \rho_k) \to \sigma \\ C; \Gamma \vdash a_i : A_i \quad (i = 1, \dots, k) \end{array}}{C; \Gamma \vdash f[a_1, \dots, a_k] : B}$$

$$\frac{C; \Gamma \vdash a : A \to B \qquad C; \Gamma \vdash b : A}{C; \Gamma \vdash a\;b : B}$$

$$\frac{\begin{array}{c} C(c) = (\Delta; x : A; a) \quad \Delta = x_1 : A_1, \dots, x_k : A_k \\ C; \Delta, x : A \vdash a : B \\ C; \Gamma \vdash a_i : A_i \quad (i = 1, \dots, k) \end{array}}{C; \Gamma \vdash c(a_1, \dots, a_k) : A \to B}$$

**Definition 5.4 (Denotational semantics of applicative terms)** *If $C;\Gamma \vdash a : A$, then for every functional environment $\xi \in \mathsf{FEnv}$ such that $\xi : \Gamma$ we define $[\![ a ]\!]^C \xi \in \mathsf{D}(A)$ (we write $[\![ a ]\!]\xi$ if $C$ is a fixed global parameter):*

$$\begin{array}{rcl} [\![ x ]\!]^C\xi & := & \xi(x) \\ [\![ f[a_1,\dots,a_k] ]\!]^C\xi & := & [\![ f ]\!]([\![ a_1 ]\!]^C\xi,\dots,[\![ a_k ]\!]^C\xi) \\ [\![ a\, b ]\!]^C\xi & := & [\![ a ]\!]^C\xi([\![ b ]\!]^C\xi) \\ [\![ c(a_1,\dots,a_k) ]\!]^C\xi & := & \lambda\!\!\lambda d \in \mathsf{D}(A).[\![ a ]\!]^C\xi'[x \mapsto d] \end{array}$$

*where in the last clause it is assumed that we have $C(c) = (\Delta; x : A; a)$ with $C; \Delta, x : A \vdash a : B$, $\Delta = x_1 : A_1,\dots,x_k : A_k$ and $C;\Gamma \vdash a_i : A_i$ $(i = 1,\dots,k)$, and $\xi'$ is defined by $\xi'(x_i) := [\![ a_i ]\!]^C\xi$ for $i = 1,\dots,k$.*

**Lemma 5.5** *(a) If $C \subseteq C'$, $\Gamma \subseteq \Gamma'$, $C;\Gamma \vdash a : A$, then $C';\Gamma' \vdash A$.*

*(b) If $C;\Gamma \vdash a : A$, $C;\Gamma \vdash a : A'$, then $A = A'$.*

*(c) If $C \subseteq C'$, $\Gamma \subseteq \Gamma'$, $\xi \subseteq \xi'$, $C;\Gamma \vdash a : A$, and $\xi : \Gamma$, $\xi' : \Gamma'$, then $[\![ a ]\!]^C\xi = [\![ a ]\!]^{C'}\xi'$.*

**Proof:** Straightforward.

## Correctness of the Parse Function $\mathsf{P}$

**Theorem 5.6** *If $r$ is a $\lambda$-term, $\Gamma \vdash r : A$, then $\mathsf{P}\ \Gamma\ r\ C \simeq (a, C')$ for some $C', a$ s.t. $C';\Gamma \vdash a : A$ and for all $\xi : \Gamma$ we have $[\![ r ]\!]\xi = [\![ a ]\!]^{C'}\xi$.*

**Proof:** Induction on the derivation of $\Gamma \vdash r : A$. The only interesting case is $r = \lambda x^B.r'$, where we have $\Gamma, x : B \vdash r' : A$. Assume $\Gamma = x_1 : A_1,\dots,x_k : A_k$ and let $\vec{x} := x_1,\dots,x_k$. By induction hypothesis $\mathsf{P}\ (\Gamma, x : B)\ r'\ C = (a', C_1)$ for some $C_1, a'$ s.t. $C_1;\Gamma, x : B \vdash a' : A$, and for $d \in \mathsf{D}(A)$ we have $[\![ r' ]\!]\xi[x \mapsto d] = [\![ a' ]\!]^{C_1}\xi[x \mapsto d]$. Then $\mathsf{P}\ \Gamma\ r\ C = (c(\vec{x}), C_2)$, where $C_2 := C_1[c \mapsto (\Gamma; x : B; a')]$, for some fresh $c$. $C_2;\Gamma \vdash x_i : A_i$, and by monotonicity $C_2;\Gamma, x : B \vdash a' : A$, therefore $C_2;\Gamma \vdash c(\vec{x}) : B \to A$. Furthermore, $[\![ x_i ]\!]^{C_2}\xi = \xi(x_i)$, therefore $\xi' \subseteq \xi$ for the $\xi'$ as in the definition of $[\![ c(\vec{x}) ]\!]^{C_2}\xi$. Therefore $[\![ c(\vec{x}) ]\!]^{C_2}\xi = \lambda\!\!\lambda d.[\![ a ]\!]^{C_2}\xi'[x \mapsto d] = \lambda\!\!\lambda d.[\![ a ]\!]^{C}\xi[x \mapsto d] = \lambda\!\!\lambda d.[\![ r' ]\!]\xi[x \mapsto d] = [\![ r ]\!]\xi$. **q.e.d.**

## Completeness of the Parse Function $\mathsf{P}$

In addition to the correctness of the translation function $\mathsf{P}$ we show the opposite direction, namely completeness: the translated versions of typed $\lambda$-terms are already essentially all typed elements of $\mathsf{App}$. The only restriction is that constructors are only applied to variables, and that they are applied to all variables in the context, independently of whether the variables occur in the body of the class or not.

**Definition 5.7** *Let $C;\Gamma \vdash' a : A$ be defined by the same rules as for $C;\Gamma \vdash a : A$, except for the rule of deriving $C;\Gamma \vdash c(a_1,\dots,a_k) : A \to B$, which is replaced by the following:*

$$\frac{C(c) = (\Gamma; x : A; a) \quad \Gamma = x_1 : A_1, \ldots, x_k : A_k \qquad C; \Gamma, x : A \vdash' a : B}{C; \Gamma \vdash' c(x_1, \ldots, x_k) : A \to B}$$

**Remark 5.8** *In Theorem 5.6 we have as well* $C'; \Gamma \vdash' a : A$.

We have no control over the choice of class names (constructors) introduced by the parse function. So a class term will in general only be reached by the parse function up to renaming of class names. Furthermore, if in a $\lambda$-term there exist the same $\lambda$-term twice as a subterm, the parse function will assign different class names to each occurrence of it. (One could improve the parse function so that this doesn't take place.) Therefore, if we want to obtain an element $a$ of $\mathsf{App}$ by parsing a $\lambda$-term $r$, it might be that in the parsed $\lambda$-term $a'$ there are two different constructors which correspond to the same constructor in $a$. So we obtain an element of $\mathsf{App}$ by parsing a $\lambda$-term only up to renaming and possibly identification of class names. The following definition of a class homomorphism makes this explicit:

**Definition 5.9** *(a) Let* $\theta : \mathsf{Constr} \to_{\mathsf{fin}} \mathsf{Constr}$. *Then* $\theta(a)$ *is defined if each constructor occurring in* $a$ *is an element of* $\mathsf{dom}(\theta)$. *If* $\theta(a)$ *is defined, then* $\theta(a)$ *is the result of replacing each occurrence of* $c \in \mathsf{Constr}$ *in* $a$ *by* $\theta(c)$. *Furthermore, for* $(\Gamma; x : B; a) \in \mathsf{Class}$ *we define* $\theta(\Gamma; x : B; a) :\simeq (\Gamma; x : B; \theta(a))$.

*(b) Let* $C, C' \in \mathsf{CEnv}$. $\theta : \mathsf{dom}(C') \to \mathsf{dom}(C)$ *is a* $\mathsf{CEnv}$*-homomorphism, if* $\forall c \in \mathsf{dom}(C').\theta(C'(c)) = C(\theta(c))$.

**Theorem 5.10 (Completeness of the Parse Function $\mathsf{P}$)** *Assume* $C; \Gamma \vdash' a : A$, *and* $C' \in \mathsf{CEnv}$. *Then there exists a* $\lambda$*-term* $r$, *a* $\mathsf{CEnv}$*-homomorphism* $\theta : C'' \to C$, *and an* $a' \in \mathsf{App}$ *s.t. the following holds*

$$\begin{array}{rcl} \Gamma & \vdash & r : A \ , \\ \mathsf{P}\,\Gamma\, r\, C' & = & (a', C' \cup C'') \ , \\ C' \cup C''; \Gamma & \vdash & a' : A \ , \\ \theta(a') & = & a \ . \end{array}$$

**Proof:** Induction on $C; \Gamma \vdash' a : A$.

Case $\Gamma = x_1 : A_1, \ldots, x_k : A_k$, $a = c(x_1, \ldots, x_k)$, $A = A' \to B'$, $C(c) = (\Gamma; x : A'; a')$, $C; \Gamma, x : A' \vdash a' : B'$. Assuming $C'$ we find by induction hypothesis $C_0$, a $\mathsf{CEnv}$-homomorphism $\theta' : C_0 \to C$, a $\lambda$-term $r'$, $a'' \in \mathsf{App}$ s.t. $\mathsf{P}\ (\Gamma, x : A')\ r'\ C' = (a'', C' \cup C_0)$ and $\theta'(a'') = a'$. Then $\mathsf{P}\ \Gamma\ (\lambda x^B.r')\ C' = (c_0(x_1, \ldots, x_k), C' \cup C_0[c_0 \mapsto (\Gamma; x : B; a'')])$ for some fresh $c_0$. Let $a' := c_0(x_1, \ldots, x_k)$, $C'' := C_0[c_0 \mapsto (\Gamma; x : B; a'')]$, $\theta := \theta'[c_0 \mapsto c]$, $r := \lambda x^B.r'$.

The other cases are straightforward. **q.e.d.**

## 6 Correctness of the Evaluation of Applicative Terms

In this Section, except for the main theorem 6.10 at the end, the class environment $C$ will not change. We will therefore omit this parameter in all notations (including $\mathsf{apply}$, $\mathsf{eval}$).

## Typing, Semantics, and Semantic Typing of Values

**Definition 6.1 (Typing of Values)** *The typing relation $H \vdash v : A$ is defined inductively by the following rules:*

*If $\rho$ is a native C++ type and $n \in [\![ \rho ]\!]$, then*

$$H \vdash n : \rho$$

*If $h \in \mathsf{Addr}$, $H(h) = (c, v_1, \ldots, v_k)$, $C(c) = (\Delta; x : A; a)$*
*$\Delta = x_1 : A_1, \ldots, x_k : A_k$ then*

$$\frac{\begin{array}{c}\Delta, x : A \vdash a : B \\ H \vdash v_i : A_i \quad (i = 1, \ldots, k)\end{array}}{H \vdash h : A \to B}$$

**Definition 6.2 (Denotational Semantics of Values)** *(a) Assuming $H \vdash v : A$ we define the denotational semantics of $v$, $[\![ v ]\!]^H \in \mathsf{D}(A)$, by recursion on the derivation of $H \vdash v : A$:*

$$\begin{array}{rcl}[\![ n ]\!]^H & := & n \\ [\![ h ]\!]^H & := & \lambda\!\!\lambda d \in \mathsf{D}(A).[\![ a ]\!]\xi[x \mapsto d]\end{array}$$

*where $H(h) = (c, v_1, \ldots, v_k)$, $C(c) = (\Delta; x : A; a)$, $\Delta = x_1 : A_1, \ldots, x_k : A_k$, $\Delta, x : A \vdash a : B$, $H \vdash v_i : A_i$ and $\xi(x_i) := [\![ v_i ]\!]^H$ $(i = 1, \ldots, k)$.*

*If $C$ needs to be mentioned we write $[\![ v ]\!]^{C,H}$ instead of $[\![ v ]\!]^H$.*

*(b) If $\eta \in \mathsf{FEnv}$, $\eta : \Gamma$, we define $[\![ \eta ]\!]^H \in \mathsf{VEnv}$ by $[\![ \eta ]\!]^H := \lambda\!\!\lambda x \in \mathsf{dom}(\eta).[\![ \eta(x) ]\!]^H$.*

**Lemma 6.3** *(a) If $H \vdash v : A$ and $H \subseteq H'$, then $H' \vdash v : A$ and $[\![ v ]\!]^H = [\![ v ]\!]^{H'}$.*

*(b) If $H \vdash v : A$ and $H \vdash v : A'$, then $A = A'$.*

The next definition is motivated by the following consideration: We want to show that $\mathsf{eval}\ \eta\ a\ H$ is defined, whenever we have $\Gamma \vdash a : A$, and if the result is $(v, H')$, then $[\![ a ]\!] = [\![ v ]\!]^{H'}$ (more precisely we have a dependency on an environment $\eta$). The problem is that $[\![ a\ \ b ]\!] = [\![ a ]\!]([\![ b ]\!])$, whereas $\mathsf{eval}\ \eta\ (a\ b)\ H :\simeq \mathsf{apply}\ v\ w\ H''$ where $v$, $w$ are obtained by applying $\mathsf{eval}$ to $a$ and $b$, and $H''$ is the heap obtained when evaluating $a$ and $b$. Even so in the proof of the correctness of $\mathsf{eval}$ we might know by induction hypothesis (for some suitable heap $H'''$) that $[\![ a ]\!] = [\![ v ]\!]^{H'''}$ and $[\![ b ]\!] = [\![ w ]\!]^{H'''}$, we will not be able to conclude that $[\![ a ]\!]([\![ b ]\!]) = [\![ v' ]\!]^{H'''}$, if $\mathsf{apply}\ v\ w\ H'' \simeq (v', H''')$, unless we know for $a$ already that it respects $\mathsf{apply}$. If $a = c(a_1, \ldots, a_n)$, this is no problem, but $a$ might be a variable or an application term $a'\ b'$.

So, when proving the correctness for an applicative term of type $A \to B$, we need to show that it respects $\mathsf{apply}$ as well. We can achieve this if we know that $\mathsf{apply}$ is only applied to terms which respect $\mathsf{apply}$ themselves. We also need to assume that all variables have this property as well. The correct condition is expressed by using the following Kripke-style logical relation $H \models v \sim d : A$

between a C++ value $v \in \mathsf{Val}$ and a denotational value ($d \in \mathsf{D}(A)$). This relation, which depends on a class environment $C$, a heap $H$, and the type $A$, can be viewed as a semantic analogue of the (proof-theoretic) typing relation.

**Definition 6.4** *(a) We define a relation $H \models v \sim d : A$ by recursion on the type $A$.*

*In order to increase readability we will use the following abbreviations: $\models \mathsf{apply}\ v\ w\ H \sim d : A$ is shorthand for*

$$\exists v', H'\ (\mathsf{apply}\ v\ w\ H \simeq (v', H') \land H' \models v' \sim d : A).$$

*Similarly, $\models \mathsf{eval}\ \eta\ a\ H \sim d : A$ stands for*

$$\exists v', H'\ (\mathsf{eval}\ \eta\ a\ H \simeq (v', H') \land H' \models v' \sim d : A).$$

*(Note that $\models \mathsf{apply}\ v\ w\ H \sim d : A$ implies $\mathsf{apply}\ v\ w\ H$ is defined):*

$$\begin{array}{rcl} H \models v \sim n : \rho & :\Longleftrightarrow & v = n \in [\![ \rho ]\!] \\ H \models v \sim f : A \to B & :\Longleftrightarrow & H \vdash v : A \to B \land [\![ v ]\!]^H = f \\ & & \land \forall H' \supseteq H, \forall (w,d) \in \mathsf{Val} \times \mathsf{D}(A). \\ & & \quad H' \models w \sim d : A \\ & & \quad \Longrightarrow \models \mathsf{apply}\ v\ w\ H' \sim f(d) : B \end{array}$$

*(b) We also define*

$$\begin{array}{l} H \models \eta \sim \xi : \Gamma :\Longleftrightarrow \mathsf{dom}(\Gamma) \subseteq \mathsf{dom}(\eta) \cap \mathsf{dom}(\xi) \land \\ \qquad\qquad\qquad\qquad \forall x \in \mathsf{dom}(\Gamma) H \models \eta(x) \sim \xi(x) : \Gamma(x) \end{array}$$

**Lemma 6.5** *If $H \models v \sim d : A$ and $H \subseteq H'$ then $H' \models v \sim d : A$.*

**Proof.** Easy by the definition and Lemma 6.3 (a). **q.e.d.**

**Lemma 6.6** *If $H \models v \sim d : A$, then $H \vdash v : A$. Hence, if $H \models \eta \sim \xi : \Gamma$, then $H \vdash \eta : \Gamma$.*

**Proof.** Clear, by definition. **q.e.d.**

## Proof of Correctness of eval

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

**Lemma 6.7** *Assume $\Gamma \vdash a : A$. Then for all $\eta$, $\xi$ we have*

$$H \models \eta \sim \xi : \Gamma \Longrightarrow \models \mathsf{eval}\ \eta\ a\ H \sim [\![a]\!]\xi : A\ .$$

**Proof.** The proof is by induction on the typing judgement $\Gamma \vdash a : A$. In the proof we will use the properties (1) and (2) of Lemma 4.7 and Lemma 6.5. We refer to the latter one as "monotonicity". We now consider the four possible cases of how $\Gamma \vdash a : A$ can be derived.

$\Gamma, x : A \vdash x : A$. Assume $\xi : (\Gamma, x : A)$ and $H \models \eta \sim \xi : (\Gamma, x : A)$. We need to show $\models \mathsf{eval}\ \eta\ x\ H \sim [\![x]\!]\xi : A$. We have $\mathsf{eval}\ \eta\ x\ H = (\eta(x), H)$ and $[\![x]\!]\xi =$

$\xi(x) \in \mathsf{D}(A)$. Furthermore, $H \models \eta \sim \xi : (\Gamma, x : A)$ entails $H \models \eta(x) \sim \xi(x) : A$, and therefore the assertion follows.

$\Gamma \vdash c(\vec{a}) : A \to B$, derived from $C(c) = (\Delta; x : A; a)$, where $\Delta = \vec{y} : \vec{A}$, $\Gamma \vdash \vec{a} : \vec{A}$, and $\Gamma, x : A \vdash a : B$. Assume $\xi : \Gamma$ and $H \models \eta \sim \xi : \Gamma$.

We need to show $\models \mathsf{eval}\ \eta\ c(\vec{a})\ H \sim [\![ c(\vec{a}) ]\!]\xi : A \to B$. By induction hypothesis for $\Gamma \vdash a_i : A_i$ and monotonicity we get that $\mathsf{mapM}\ (\mathsf{eval}\ \eta)\ \vec{a}\ H = (\vec{v}, H_1)$ for some $H_1$, $\vec{v}$ s.t. $H_1 \models v_i \sim [\![ a_i ]\!]\xi : A_i$ Therefore $\mathsf{eval}\ \eta\ c(\vec{a})\ H = (h, H')$, where $H' = H_1[h \mapsto (c, \vec{v})])$ with $h = \mathsf{fresh}(H_1)$. We need to show $H' \models h \sim [\![ c(\vec{a}) ]\!]\xi : A \to B$, which is a conjunction of three statements (i), (ii), (iii):

(i) We need to show $H' \vdash h : A \to B$, which follows, since $H'(h) = (c, \vec{v})$, $C(c) = (\Delta; x : A; a)$, $\Gamma, x : A \vdash a : B$ and $H' \vdash v_i : A_i$, where the last statement follows by $H_1 \models v_i \sim [\![ a_i ]\!]\xi : A_i$.

(ii) We need to show $[\![ h ]\!]^{H'} = [\![ c(\vec{a}) ]\!]\xi$:

$$\begin{array}{rcl} [\![ h ]\!]^{H'} & = & \lambda\!\!\lambda d \in \mathsf{D}(A).[\![ a ]\!][\![ \eta_0 ]\!]^{H'}[x \mapsto d] \ , \\ [\![ c(\vec{a}) ]\!]\xi & = & \lambda\!\!\lambda d \in \mathsf{D}(A).[\![ a ]\!]\xi_0[x \mapsto d] \end{array}$$

where $\eta_0 := [\vec{y} \mapsto \vec{v}]$, $[\![ \eta_0 ]\!]^{H'} := \lambda\!\!\lambda y \in \mathsf{dom}(\eta_0).[\![ \eta_0(y) ]\!]^{H'}$, and $\xi_0 = [\vec{y} \mapsto [\![ \vec{a} ]\!]\xi]$. By $H_1 \models v_i \sim [\![ a_i ]\!]\xi : A_i$ and monotonicity we have $H' \models \eta_0 \sim \xi_0 : \Delta$. Therefore $[\![ \eta_0(y_i) ]\!]^{H'} = \xi_0(y_i)$ and we are done.

(iii) Assume $H'' \supseteq H'$ and $H'' \models w \sim d : A$. We need to show

$$\models \mathsf{apply}\ h\ w\ H'' \sim ([\![ c(\vec{a}) ]\!]\xi)(d) : B \ .$$

First,

$$\begin{array}{rcl} \mathsf{apply}\ h\ w\ H'' & \simeq & \mathsf{eval}\ \eta_0[x \mapsto w]\ a\ H'' \ , \\ ([\![ c(\vec{a}) ]\!]\xi)(d) & = & [\![ a ]\!]\xi_0[x \mapsto d] \ . \end{array}$$

By $H' \models \eta_0 \sim \xi_0 : \Delta$, $H'' \models w \sim d : A$, and monotonicity we obtain $H'' \models \eta_0[x \mapsto w] \sim \xi_0[x \mapsto d] : (\Delta, x : A)$. Using the induction hypothesis we obtain $\models \mathsf{eval}\ \eta_0[x \mapsto w]\ a\ H'' \sim [\![a]\!]\xi_0[x \mapsto d] : B$.

$\Gamma \vdash a\ b : B$, derived from $\Gamma \vdash a : A \to B$ and $\Gamma \vdash b : A$. Assume $\xi : \Gamma$ and $H \models \eta \sim \xi : \Gamma$. We need to show $\models \mathsf{eval}\ \eta\ (a\ b)\ H \sim [\![a\ b]\!]\xi : A$. By induction hypothesis and (1), $\mathsf{eval}\ \eta\ a\ H = (v, H_1)$ for some $H_1 \supseteq H$ with $H_1 \models v \sim [\![a]\!]\xi : A \to B$ and, using monotonicity, $\mathsf{eval}\ \eta\ b\ H_1 = (w, H_2)$ for some $H_2 \supseteq H_1$ with $H_2 \models w \sim [\![b]\!]\xi : A$. By the definition of $H_1 \models v \sim [\![ a ]\!]\xi : A \to B$ we obtain $\models \mathsf{apply}\ v\ w\ H_2 \sim [\![a]\!]\xi([\![b]\!]\xi) : B$ and we are done, since $\mathsf{eval}\ \eta\ (a\ b)\ H \simeq \mathsf{apply}\ v\ w\ H_2$ and $[\![a\ b]\!]\xi = [\![a]\!]\xi([\![b]\!]\xi)$.

$\Gamma \vdash f[a_1, \ldots, a_k] : B$, derived from $\Gamma \vdash a_i : A_i$, $i = 1, \ldots, k$, where $f : (A_1, \ldots, A_n) \to B$. Assume $\xi : \Gamma$ and $H \models \eta \sim \xi : \Gamma$. We need to show $\models \mathsf{eval}\ \eta\ f[\vec{a}]\ H \sim [\![f[\vec{a}]]\!]\xi : B$. By induction hypothesis and (1), $\mathsf{eval}\ \eta\ a_1\ H = (n_1, H_1)$ for some $n_1 \in [\![ A_1 ]\!]$ and $H_1 \supseteq H$ with $H_1 \models n_1 \sim [\![a_1]\!]\xi : A_1$, especially $n_1 = [\![a_1]\!]\xi$. Similarly, using monotonicity and (1), for $i = 1, \ldots, k-1$ we have $\mathsf{eval}\ \eta\ a_{i+1}\ H_i = (n_{i+1}, H_{i+1})$ for some $n_{i+1} \in [\![ A_i ]\!]$ and $H_{i+1} \supseteq H_i$ with $n_{i+1} = [\![a_{i+1}]\!]\xi$. It follows $\mathsf{eval}\ \eta\ f[\vec{a}]\ H = ([\![f]\!](\vec{n}), H_k) = ([\![f[\vec{a}]]\!]\xi, H_k)$. **q.e.d.**

**Lemma 6.8** *If $H \vdash v : A$ then $H \models v \sim [\![ v ]\!]^H : A$.*

**Proof.** Induction on $H \vdash v : A$.

The case $A \in \mathsf{basetype}$ is trivial.

Case $H \vdash h : A \to B$, derived from $H(h) = (c, \vec{v})$, $C(c) = (\vec{x} : \vec{A}; x : B; a)$, $\vec{x} : \vec{A}, x : A \vdash a : B$, $H \vdash \vec{v} : \vec{A}$. We need to show that for $H' \supseteq H$, $w, d$ s.t. $H' \models w \sim d : A$ we have $\models \mathsf{apply}\ h\ w\ H' \sim [\![ h ]\!]^H(d) : B$, so assume $H', w, d$ as stated.

Let $\eta := [\vec{x} \mapsto \vec{v}]$, $\xi := [\![ \eta ]\!]^H$. Then $[\![ h ]\!]^H(d) = [\![ h ]\!]^{H'}(d) = [\![ a ]\!]\xi[x \mapsto d]$. By induction hypothesis $H \models v_i \sim [\![ v_i ]\!]^H : A_i$, and by $H' \models w \sim d : A$ we get therefore $H' \models \eta[x \mapsto w] \sim \xi[x \mapsto d] : \Gamma$. Furthermore,
$\mathsf{apply}\ h\ w\ H' \simeq \mathsf{eval}\ (\eta[x \mapsto w])\ a\ H'$. By Lemma 6.7 $\models \mathsf{eval}\ (\eta[x \mapsto w])\ a\ H' \sim [\![a]\!]\xi[x \mapsto d] : B$, which proves the assertion

**Theorem 6.9 (Correctness of** eval**)** *If $\Gamma \vdash a : A$, $H \vdash \eta : \Gamma$, then there exists $H', v$ s.t. $\mathsf{eval}\ \eta\ a\ H \simeq (v, H')$, $H' \vdash v : A$ and $[\![ a ]\!]([\![ \eta ]\!]^{H'}) = [\![ v ]\!]^{H'}$.*

**Proof:** Immediate by Lemma 6.7 and 6.8.

## Main Theorem

**Theorem 6.10 (Overall Correctness)** *Assume $\vdash r : A$ 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 $\eta$ and any $C'' \supseteq C'$ we have*

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

*for some $H'$, $v$ s.t. $[\![ r ]\!]\emptyset = [\![ v ]\!]^{C,H'}$.*

*Especially, in case $A = \mathsf{int}$ we have that $r \to_\beta n$ for some $n$, and therefore $[\![ r ]\!]\emptyset = n$ and*

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

**Proof.** By Theorem 5.6 and 6.9.

**Remark.** The proof of Theorem 6.7 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 [Pit91] and special logical relations for monads [GLN02].

## 7 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 denotational semantics and logical relations.

This work lends itself to a number of extensions, for example, the integration of recursive higher-order functions, polymorphic and dependent type systems, the integration lazy evaluation and infinite structures as well as the combination of larger parts of C++ with the $\lambda$-calculus. The accurate description of these extensions would require more sophisticated, e.g. domain-theoretic constructions. 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 $\lambda$-terms with side-effects) will result in interesting new programming techniques.

The remarkable fact that it is possible to have a denotational semantics at a description level where pointers are manipulated explicitely entails that the well-known benefits of denotational semantics, extensionality and compositionality, are still available at that level. This has already paid off in this paper where we were able to give a short and concise correctness proof for our C++ fragment using the denotational semantics (instead of a complicated operational argument). More benefits are to be expected when it comes to verifying programs written in this C++ fragment or in one of the future extensions mentioned above.

## 8 List of Identifiers and Notations

In the following we list the notations used in this article, together with the place where they are introduced. 2.1 (a).

| | | |
|---|---|---|
| Finite functions | $X \to_{\mathsf{fin}} Y$ | Def. 1.1 |
| Extension of fin. fns. | $g[x \mapsto y]$ or $f, x : y$ | Def. 1.1 |
| Explicitly given fin. fns. | $x_1 : y_1, \ldots, x_k : y_k$ | Def. 1.1 |
| Base types | $\mathsf{basetype} \ni \rho, \sigma$ | Ass. 2.1 (a) |
| Base type $\mathsf{int}$ | $\mathsf{int}$ | Ass. 2.1 (a) |
| Basic functions | $\mathsf{F} \ni f$ | Ass. 2.1 (b) |
| Denot. sem. of base type | $[\![ \rho ]\!]$ | Ass. 2.1 (d) |
| Denot. sem. of basic fn. | $[\![ f ]\!]$ | Ass. 2.1 (e) |
| Constants (of any base type) | $n$ | Ass. 2.1 (c) |
| Variables | $\mathsf{Var} \ni x, y, z$ | Def. 2.2 (a) |
| Types | $\mathsf{Type} \ni A, B$ | Def. 2.2 (b) |
| Contexts | $\mathsf{Context} \ni \Gamma, \Delta$ | Def. 2.2 (b) |
| Extension of Contexts | $\Gamma, x : A$ | Def. 2.2 (b) |
| $\lambda$-terms | $\mathsf{Term} \ni r, s, t$ | Def. 2.2 (b) |
| Typed $\lambda$-terms | $\Gamma \vdash r : A$ | Def. 2.2 (b) |
| Functionals of type $A$ | $\mathsf{D}(A) \ni d$ | Def. 2.3 |
| Functional environment | $\mathsf{FEnv} \ni \xi$ | Def. 2.4 (a) |
| Typed contexts | $\xi : \Gamma$ | Def. 2.4 (b) |
| Denot. sem. of $\lambda$-terms | $[\![ r ]\!]\xi$ | Def. 2.5 |
| Name of a type | $\mathsf{name}(A)$ | Sect. 3 |
| Translation of $\lambda$-terms | $t^{C++}$ | Sect. 3 |
| Constructors or classnames | $\mathsf{Constr} \ni c$ | Def. 4.1 |
| Applicative terms | $\mathsf{App} \ni a, b$ | Def. 4.1 |
| Classes | $\mathsf{Class} \ni (\Gamma; x : A; b)$ | Def. 4.1 |
| Class environments | $\mathsf{CEnv} = \mathsf{Constr} \ni C$ | Def. 4.1 |
| Free variables of $a$ | $\mathrm{FV}(a)$ | Def. 4.2 |
| Heap addresses | $\mathsf{Addr} \ni h$ | Def. 4.3 |
| C++ constants | $n$ | Def. 4.3 |

| | | |
|---|---|---|
| Values | $\mathsf{Val} \ni v, w$ | Def. 4.3 |
| Closures | $\mathsf{Constr} \times \mathsf{Val}^* \ni (c, v_1, \dots, v_k)$ | Def. 4.3 |
| Heaps | $\mathsf{Heap} = \mathsf{Addr} \ni H$ | Def. 4.3 |
| Value environments | $\mathsf{VEnv} \ni \eta$ | Def. 4.3 |
| State Monad | $\mathsf{M}_X(Y)$ | Def. 4.4 |
| Monadic notations | $\mathsf{do}, \mathsf{return}, \mathsf{mapM}, \mathsf{read}, \mathsf{add}$ | Not. 4.5 |
| Fresh element | $\mathsf{fresh}(X)$ | Not. 4.5 |
| Eval and Apply | $\mathsf{eval}, \mathsf{apply}$ | Def. 4.6 |
| Parse function | $\mathsf{P}$ | Def. 5.1 |
| Typed applicative terms | $C; \Gamma \vdash a : A$ | Def. 5.3 |
| Denot. sem. of appl. terms | $[\![ a ]\!]^C \xi$ or $[\![ a ]\!] \xi$ | Def. 5.4 |
| Variant of above | $C; \Gamma \vdash' a : A$ | Def. 5.7 |
| $\mathsf{CEnv}$-homomorphism | $\theta$ | Def. 5.9 (b) |
| $\theta$ applied to $r$ | $\theta(r)$ | Def. 5.9 (a) |
| $\theta$ applied to a class | $\theta(\Gamma; x : B; a)$ | Def. 5.9 (a) |
| Typed values | $H \vdash v : A$ | Def. 6.1 |
| Denot. sem. of values | $[\![ v ]\!]^H$, $[\![ v ]\!]^{C,H}$ | Def. 6.2 (a) |
| Denot. sem. of value envs. | $[\![ \eta ]\!]^H$ | Def. 6.2 (b) |
| Semantic typing of values | $H \models v \sim d : A$ | Def. 6.4 |
| Special not. for the above | $\models \mathsf{apply}\ v\ w\ H \sim d : A$ | |
| | $\models \mathsf{eval}\ \eta\ a\ H \sim d : A$ | Def. 6.4 |

## References

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

[GLN02] 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.

[IPW99] 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.

[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, 1993.

[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.

[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(1):55–92, 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.

[Pit91] A. M. Pitts. Evaluation logic. In G. Birtwistle, editor, *IVth Higher Order Workshop, Banff 1990*, Workshops in Computing, pages 162–189. Springer, 1991.

[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.

[Rey83] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP'83*, pages 513–523. North-Holland, 1983.

[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine – Definition, Verification, Validation.* Springer, 2001.

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

[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. *Journal of Symbolic Logic*, 32:198 – 212, 1967.

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